INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI[®]

AN EFFICIENT HARDWARE IMPLEMENTATION OF A RECONFIGURABLE TURBO ENCODER/ DECODER EMULATOR

By

Henry Wong, B.Eng. (Electrical) with High Distinction,
Carleton University, Ottawa, Ontario, Canada

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

Ottawa-Carleton Institute for Electrical Engineering
Department of Electronics
Carleton University
Ottawa, Ontario, Canada

Copyright © 2005 by Henry Wong

Library and
Archives Canada

Bibliothèque et
Archives Canada

0-494-06808-6

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

Abstract

An Efficient Hardware Implementation of a Reconfigurable Turbo Encoder/Decoder Emulator

Henry Wong

Department of Electronics, Carleton University,

Ottawa, Ontario, Canada

2005

This work presents the design and implementation of a reconfigurable Turbo encoder/decoder emulator built on a Xilinx FPGA development platform. It deploys efficient hardware design to improve performance and it is also reconfigurable. It allows real time selection of total frame number, the termination requirement, and the number of iterations, beside a choice of puncturing and channel reliability estimation. Other settings that require recompilation for reconfiguration include frame sizes, code rate and the number of trellis states. A simulation of 1000 one Mbit-frames requires only 6 minutes to complete versus hours or days for a software simulation on a PC.

The Turbo decoder itself has a data throughput of 2 Mbps for 8 iterations with a maximum clock frequency of 64MHz and occupies approximately 6200 configurable logic blocks of a Xilinx Virtex II FPGA. It uses the novel fixed windowing technique to reduce memory requirement, while maintaining good decoding performance.

In the loving memory of my father, who inspired me to become an engineer.

His smile will never fade, his great personality
and work ethic will always be remembered.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Professor John Knight, for his guidance and support throughout this research. His amazing knowledge on logic design has shed much light on the hardware optimization.

I am deeply indebted to the hardware manager Nagui Mikhail for his countless help on obtaining the Xilinx development platform and setting up the lab environment.

I would like also to take this opportunity to acknowledge Professor Ian Marsland for numerous insightful discussions about the Channel Coding theory.

I am grateful to the members of VLSI group for their support; in particular to Gordon Allan for his time to introduce me to Turbo decoder in the early stages and Fred Ma who helped me to resolve some Matlab issues.

I would like to express my appreciation to Winnie Ye for her boundless encouragement and inspiration to help me focus on achieving the goal.

I would also like to thank my parents, my sister and brother-in-law who always support me with their unconditional love.

Finally, I acknowledge Carleton University and OGS-ST for their financial support during this work.

Table of Contents

Abstract.....	i
Acknowledgements.....	iii
Table of Contents.....	iv
List of Figure	vii
List of Table.....	ix
Glossary	x
Acronyms.....	xiv
Chapter 1: Introductions	1
1.1 History of Error Control Coding.....	2
1.2 Classification of Error Control Codes.....	3
1.3 Convolutional Codes.....	5
1.3.1 Convolutional Encoder.....	6
1.3.2 Viterbi Decoder.....	8
1.3.3 Recursive Systematic Codes.....	10
1.3.4 Component Encoder with RSC.....	11
1.3.5 Turbo Decoder	12
1.3.6 Soft Output Viterbi Algorithm (SOVA).....	14
1.4 Thesis Objectives	15
1.5 Thesis Outline	16
Chapter 2: Maximum A-Posteriori Probability Decoding With Iterations	18
2.1 Log Likelihood Ratio Calculation.....	18
2.2 A-Posteriori Data Calculation for Iterations	20
2.3 Trellis Decoding Using the MAP Algorithm	22
Chapter 3: Soft Input Soft Output Decoders.....	25
3.1 MAP Algorithm	25
3.2 Max-Log-MAP Algorithm.....	29
3.3 Log-MAP Algorithm.....	32
Chapter 4: Turbo Decoder With Log-Map Algorithm Demonstration	35
4.1 Sample Γ_{S^t} Calculations.....	36
4.2 Sample A_{S^t} Calculations	37
4.3 Sample B_{S^t} Calculations	39
4.4 Trellis Termination.....	40
4.5 Sample LLR and A-Posteriori Data Calculations	41
4.6 Interleaver	42
4.7 Iterations.....	43
Chapter 5: Simulated Decoding Performance of the Turbo Decoder	45
5.1 Number of Iterations	45
5.2 Channel Reliability Factor	46
5.3 Generator Polynomial	47
5.4 Code Puncturing.....	49
5.5 Interleaver Function	50

5.6	Frame Size.....	51
5.7	Decoding Algorithm	52
5.8	Trellis Termination.....	53
Chapter 6: Hardware Design and Implementation		54
6.1	Design Platform Description.....	54
6.2	Variable Quantization.....	55
6.3	Turbo Decoder Emulator (TDE) Design.....	56
6.3.1	TDE Controller Design.....	58
6.3.2	User Interface	59
6.4	Address Generator (Interleaver) Design	60
6.5	Data Generator Design.....	63
6.5.1	Data Generator Controller Design.....	64
6.5.2	LFSR Random Input Generator.....	65
6.5.3	Component Encoder	65
6.5.4	Channel Data Mapper.....	66
6.5.5	Address Generator Utilization.....	67
6.5.6	Data Generator Memory Map.....	67
6.6	Turbo Decoder Design	69
6.6.1	Turbo Decoder Controller Design	69
6.6.2	Address Generator Utilization	70
6.6.3	Data Generator Memory Map.....	71
6.7	Log-MAP Decoder Design	73
6.7.1	Fixed Windowing Technique.....	74
6.7.2	Log-MAP Decoder Controller Design	76
6.7.3	Branch Metric Calculation Unit Design	76
6.7.4	Path Metric Calculation Unit Design.....	78
6.7.5	A-Posteriori Data and LLR Calculation Unit Design.....	81
Chapter 7: Emulator Performance and Conclusions.....		83
7.1	Turbo Decoder Simulation Improvements.....	83
7.1.1	Reduction of Simulation Time.....	83
7.1.2	Runtime Verification.....	84
7.1.3	Portability to Hardware Implementation	84
7.2	Turbo Decoder Hardware Specification and Benchmarking	84
7.2.1	Special Hardware Features	84
7.2.2	Other Hardware Implementation Techniques.....	86
7.2.3	TDE Performance Data	86
7.3	Construction of a Matlab Model.....	88
7.4	Research Extension	88
List of References		89
Appendix A: Verilog Codes		92
A.1	Top Level Emulator.....	92
A.2	Data Generator	103
A.3	Component Encoder.....	106
A.4	RSC Encoder.....	106
A.5	Address Generator and Interleaver	106
A.6	Turbo Decoder	110
A.7	Log-MAP Decoder.....	114
A.8	Branch Metrics Calculation Units.....	118
A.9	Path Metrics Calculation Units	119

A.10	LLR and A-Posteriori Data Calculation Unit	122
Appendix B:	Matlab Codes.....	126
B.1	Top Level Simulator.....	126
B.2	Turbo Decoder	127
B.3	Log-MAP Decoder.....	129
B.4	Path Metric Unit.....	131
B.5	LLR and A-Posteriori Data Calculation Unit	131
Vita.....		132

List of Figures

FIGURE 1.1.	Encoders and decoders in a typical communication system.....	4
FIGURE 1.2.	Explanation of terms for convolutional codes on an example RSC encoder.....	6
FIGURE 1.3.	Convolutional encoder	7
FIGURE 1.4.	Trellis construction of the G7 G5 convolutional encoder	7
FIGURE 1.5.	Viterbi decoder trellis decoding	8
FIGURE 1.6.	Recursive systematic codes (RSC) encoder	11
FIGURE 1.7.	Component encoder for Turbo decoder	12
FIGURE 1.8.	Error correction codes system using component encoder and Turbo decoder.....	13
FIGURE 2.1.	Decoding based on likelihood of data received	18
FIGURE 2.2.	Iterative Turbo decoder structure	21
FIGURE 2.3.	Probability decoding on trellis	22
FIGURE 3.1.	Recursive calculation of α_{A_t} and β_{A_t}	26
FIGURE 3.2.	Distribution of possible value of the received data y_t	28
FIGURE 3.3.	Numerical calculation of γ_{S_t} using the MAP algorithm	29
FIGURE 4.1.	Trellis construction for Log-MAP decoder.....	36
FIGURE 4.2.	Numerical calculation of Γ_{S_t} using Log-MAP algorithm	37
FIGURE 4.3.	Numerical calculation of A_{S_t} using Log-MAP algorithm.....	38
FIGURE 4.4.	Iterative calculations of A_{S_t} along trellis stating at state A at time $t = 0$	38
FIGURE 4.5.	Numerical Calculation of B_{S_t} using Log-MAP algorithm	39
FIGURE 4.6.	Iterative calculations of B_{S_t} along trellis from the end of the data sequence to the start, using state A as the initial state at the at time $t = 32$	40
FIGURE 4.7.	Numerical calculation of LLR $L(u_t y)$ and a-posteriori output $L_e(u_t)$ using Log-MAP algorithm	41
FIGURE 4.8.	Calculations of LLR and a-posteriori data along trellis	42
FIGURE 5.1.	Performance of Turbo decoder for different number of iterations at $L = 1024$ bits	46
FIGURE 5.2.	Performance of Turbo decoder for different L_c setting	47
FIGURE 5.3.	Performance of Turbo decoder for different generator polynomials	48
FIGURE 5.4.	Performance of Turbo decoder for different generator polynomial with different constraint length	49
FIGURE 5.5.	Performance of a Turbo decoder under code puncturing.....	50
FIGURE 5.6.	Effect of interleaver in performance of Turbo decoder.....	51
FIGURE 5.7.	Performance of Turbo decoder for different frame sizes	52
FIGURE 5.8.	Performance of Turbo decoder for different decoder algorithms	53
FIGURE 5.9.	Performance of Turbo decoder for different choices of termination	53
FIGURE 6.1.	Xilinx MicroBlaze and Multimedia demonstration board	54
FIGURE 6.2.	TDE block diagram	57
FIGURE 6.3.	State diagram of TDE.....	59
FIGURE 6.4.	User interface of the TDE through the Chipscope Pro program.....	60
FIGURE 6.5.	Block diagram of the address generator.....	61
FIGURE 6.6.	Zero state insertion circuit in LFSR.....	62
FIGURE 6.7.	Block diagram of the random address generator	62
FIGURE 6.8.	Block diagram of the data generator.....	64
FIGURE 6.9.	State diagram of the data generator controller	65
FIGURE 6.10.	Block diagram of the component encoder	66
FIGURE 6.11.	Data mapping diagram	66

FIGURE 6.12.	Block diagram of Turbo decoder	69
FIGURE 6.13.	State diagram of the Turbo decoder	70
FIGURE 6.14.	Block diagram of the Log-MAP decoder	73
FIGURE 6.15.	Comparison between sliding window and fixed windowing techniques	75
FIGURE 6.16.	State diagram of the Log-MAP decoder	76
FIGURE 6.17.	Block diagram of the branch metrics calculation unit	77
FIGURE 6.18.	Block diagram of the branch metrics bundle	78
FIGURE 6.19.	Block diagram of the path metrics calculation unit	79
FIGURE 6.20.	Block diagram of the path metrics bundle	81
FIGURE 6.21.	Block diagram of the a-posteriori data and LLR calculation unit.....	82
FIGURE 7.1.	Indication of the critical path in the TDE.	87

List of Table

TABLE 1.1.	Common terms in convolutional codes	5
TABLE 3.1.	Correction factors for Log-MAP decoder.....	33
TABLE 4.1.	Parameters of Turbo decoder example using Log-MAP algorithm.....	35
TABLE 4.2.	Parameter transitions outline	35
TABLE 4.3.	Random interleaving table for 32 bit frame.....	43
TABLE 4.4.	Deinterleaving table for random interleaving of a 32 bit frame	43
TABLE 4.5.	LLR table for the example Turbo decoder.....	44
TABLE 5.1.	Default configuration for Turbo decoder simulator.....	45
TABLE 6.1.	Data variable quantization values in order to reduce the width of the data representation.....	55
TABLE 6.2.	Default configuration for TDE	56
TABLE 6.3.	Address substitutions for trellis termination tits.....	63
TABLE 6.4.	Address generator utilization in data generator	67
TABLE 6.5.	Data generator memory map.....	68
TABLE 6.6.	Address generator utilization in Turbo decoder.....	71
TABLE 6.7.	Turbo decoder memory map.....	72
TABLE 6.8.	Look-up-table (LUT) to determine correction factor	80
TABLE 7.1.	Performance data of the TDE FPGA	86
TABLE 7.2.	Comparison of the recent Turbo decoder design specifications	87

Glossary

Term	Meaning
a-posteriori data $L(u_t)$	A-posteriori data represents the knowledge obtained by observing experimental result.
a-priori data $L_e(u_t)$	A-priori data represents the knowledge obtained by theory. In Turbo decoder, it is obtained by interleaving or de-interleaving the a-posteriori data from the other decoder.
block codes	Block codes collect data into blocks of fixed length, often bytes, which contain data and/or parity bits.
channel reliability L_c	This is used to weight the reliability of the systematic input determined by the type and level of noise in the channel.
code rate R	Rate of actual data versus the coded data, a smaller value represents higher redundancy, but decrease the actual data rate.
code word, expected $\tilde{x}_t$	This is the set of hard outputs consisting of systematic and parity one would expect to be generated by a convolutional encoder for a particular branch transition. Here, $\tilde{x}_t = (x_{st}, x_{pt})$
code word, received $\tilde{y}_t$	This is the set of soft inputs a SISO decoder uses for decoding. Here, $\tilde{y}_t = (y_{st}, y_{pt})$
component encoder (parallel concatenated encoder)	A encoder structure that forms a parallel concatenation of two RSC encoder separated by a interleaver as shown in Figure 1.7.
constraint length T	The length of the encoder shift register plus 1 (for the input), which represent the number of bits affecting the output.
convolutional codes	A group of error correction codes that correlate input data with a number of past inputs to increase the redundancy of data transmission.
correction factor f_c	A factor that improve the path metric calculation in the Log-MAP algorithm to allow it achieve the same performance as the optimal MAP algorithm.
de-interleaving	It is done by doing inverse permutation, i.e. $\pi^{-1}(i)$, which reorganized a sequence to its original order as seen by the encoder.
encoder input d_t or u_t	Input to the convolutional encoder expressed in a normal form $d_t = (0, 1)$ or in a modulated form $u_t = (1, -1)$

Term	Meaning
energy per message bit E_b	This value represent the amount of energy required to transmit a bit of the original message. Hence, $E_s = E_b \cdot R$
energy per message symbol E_s	Amount of energy carried by a coded symbol such as the parity or systematic. It is taken to be one for simplification.
frame	A frame is a unit that described a complete continues sequence of data.
frame length L	The total number of bits in the original (uncoded) data frame.
generator polynomial	An octal description of the generator polynomial. (see Figure 1.3.1 for usage on convolutional codes and Figure 1.3.4 for usage on RSC)
Hamming distances	A measure of how far apart two binary numbers are by comparing the bits in the same positions in the two numbers.
interleaver	Device that perform interleaving to increase the distance between the parity bits and allow parity bit be generated with different companions in the shift register of the encoder.
interleaving	It is done by the random permutation function $\pi(i)$, which selectively rearranges the order of a sequence of bits.
iteration	A number of cycles of message passing that happens between SISO decoders.
Jacobian logarithm	A method for expressing natural logarithms of the sum of exponential values, as the sum of the maximum exponential index plus a correction factor. (see Figure 3.22)
Likelihood	It is a conditional probability function that considers how likely a value of the original input was given the received code word.
Likelihood Ratio	The ratio of comparing the likelihood of one possible encoder input against the other.
Log Likelihood Ratio $L(u_i Y)$	A normalized comparison of the conditional probability which is centered at zero. The polarity represents the estimate of the original data.
Log-MAP	An improved version of Max Log-MAP algorithm which include a correction factor created by the Jacobian logarithm. Since the conversion is exact, its performance matches the optimal MAP algorithm.
Matlab	Software that support evaluation of complex mathematical function through programmed instructions and accessing its library of functions.

Term	Meaning
Max Log-MAP	An simplified version of the MAP algorithm that performs its operation in the Log domain to avoid multiplications. Because of the imperfectness in the approximation, its performance is sub-optimal to the MAP algorithm.
metric, backward path B_{S_t}	An accumulation of probability from the end of the message that represents the probability of being in the current state given the future received data.
metric, branch $\Gamma_{S'S_t}$	The probability of state transition given the received data and a-priori information.
metric, forward path A_{S_t}	An accumulation of probability from the beginning of the message that represent the probability of being in the current state given the past received data.
number of state n	Number of possible value that the encode shift register can have.
number of value in a code word k	Number of outputs from a convolutional encoder or a RSC encoder.
parity, expected x_{pt}	The expected value of the parity data for a particular branch transition. Here, $x_{pt} = (1, -1)$
parity, received y_{pt}	The soft value received parity. This value could be corrupted by noise.
permutation	A rearrangement of the elements of a frame of data.
primitive polynomial	a type of polynomial that allow generation of evenly spaced code words during encoding and also allows specification of a LFSR that has the maximum number of connected states.
puncture	A method to increase code rate in a encoding scheme by reducing the total amount of parity data transmitted. This can be done in a low noise environment to increase the rate of actual data transmission.
quantization	The division of a range of continuous values into a single discrete number. The technique to simplify hardware implementation when the highest accuracy is not needed.
received sequence Y	The sequence of code word received, i.e. $Y = (y_0, y_1, \dots, y_t, \dots, y_{(L-1)})$
Shannon's limit	The maximum amount of information that can be transmitted over a communication link with a specified bandwidth in the presence of noise interference.
simulation time M	The total amount of time to complete a desirable simulation specified by user.

Term	Meaning
soft value	Soft value gives the weight and polarity of a number (e.g. -0.1 and +1.9 etc.) versus the hard value which only gives the polarity of a number (e.g. -1 and +1).
state S	The value of bits in the encoder's shift register.
systematic, expected x_{st}	The expected value of the original data for a particular branch transition. Here, $x_{st} = (1, -1)$
systematic, received y_{st}	The received soft value of the original data. This value could be corrupted by noise.
termination	is used for resetting the last trellis state back to a known state (state A) by transmitting appropriate tail bits after all systematic and parity bits are transmitted.
time point t	A reference point in the timeline. It is used for indicating the order of events.
trellis	A data structure for decoding use. It is constructed by placing copies of the existing state along a timeline.
Turbo Code	Code sequence that generate by two recursive systematic code encoder working in parallel and separate by an interleaver
Turbo decoder	A decoding mechanism that decodes turbo code using iterations between SISO decoders.
Verilog	A hardware descriptive language that is more popular in North America compare to the other well known HDL called VHDL which is has more European popularity.
Viterbi decoder	A decoding mechanism that decodes convolutional codes by using a trellis to estimate the transmission data sequence.
window size W	The amount of time point included during a trellis decoding process.
windowing	A technique to reduce memory requirement by only using data within W time for decoding while having little or no loss in performance.
windowing, fixed	In back tracing, this windowing technique that first initializes the path metric at time $t + W$ and uses this accumulated value to create path metric from t to $t - W$.
windowing, sliding	In back tracing, this windowing technique that always uses path metrics initialized at time $t + W$ to create the current path metric.

Acronyms

Terms	Meaning
3G	Third-generation mobile telephone technology - a mobile technology that enable communication using voice, text, images and video.
ACS	Add-Compare-Select - the type of circuit that perform several additions and compare the sums to locate the largest/smallest value depending on the application.
ASIC	Application Specific Integrated Circuit - a custom IC developed for a targeted application.
AWGN	Additive White Gaussian Noise - a common wideband channel thermal noise impairment, on which signal-to-noise ratio (SNR) is typically based.
BER	Bit Error Rate - a figure of merit for a digital communication link. It is the fraction of bits received in error divided by the total number of bits transported.
BPSK	Binary Phase-Shift Keying - a type of modulation scheme in digital communication where the phase of the RF carrier is shifted 180 degrees in accordance with the two possible symbol {0,1} of a data bit.
CDMA	Code-Division Multiple Access - a digital cellular technology that uses spread-spectrum techniques.
CPU	Central Processing Unit - allows execution of sequential software instructions to perform tasks such as calculations.
DLL	Delay lock loop - circuits that provide zero propagation delay, low-clock skew between output clock signals throughout a device.
ECC	Error Control Codes - algorithm of decrease error in data transmission by introducing higher redundancy.
FER	Frame Error Rate - a figure of merit for a digital communication link. It is the fraction of frame received that has error divided by the total number of frame transported.
FPGA	Field Programmable Gate Arrays - a family of general-purpose logic devices that can be configured by the end user to perform many different complex logic functions. It is often used for prototyping logic hardware.
GSM	Global System for Mobile Communications - one of the leading digital cellular systems that uses time division multiplexing techniques.
HDL	Hardware Description Language - a text representation of logics implementation.
IC	Integrated Circuit - a semiconductor device that combines multiple transistors and other components and interconnects on a single piece of semiconductor material.

Terms	Meaning
ICON	Integrated Controller - a Chipscope Pro logic block that communicates with software via JTAG and controls other block within the Chipscope interface block.
ILA	Integrated Logic Analyzer - a Chipscope Pro logic block that implement logic analyzer function with internal FPGA logic and memories. Data samples are stored and shifted out from the internal memories.
JTAG	Joint Test Action Group - an standard named "IEEE Standard 1149.1" that specify how to control and monitor the pins of compliant devices on a printed circuit board.
LFSR	Linear Feedback Shift Register - shift register in which some of its outputs are connected to the input through some logic gates (typically, an exclusive-or (XOR). A wide variety of bit patterns can be generated inexpensively, including pseudo-random sequences. Can be used as a high speed counter.
LLR	Log Likelihood Ratio - output of the MAP algorithm which gives a normalized measure of the polarity and confident level of the original data estimation.
LUT	Look Up Table - logics that provide content look up to retrieve the relating data.
MAP	Maximum A-Posteriori Probability - a algorithm compares the likelihood of the possible original values of the data for error correction. It also gives the reliability of the estimation.
ML	Maximum Likelihood - a method of decoding based on recovering the transmission sequence.
RSC	Recursive Systematic Codes - convolutional encoder that uses the input and feedback(s) from taps of the shift register to generate new input for itself.
SISO	Soft Input Soft Output decoder - a type of trellis decoder that decode using soft value input and also create soft value output to indicate the polarity as well as the confident level of the output.
SNR	Signal-to-noise ratio - a fractional relationship between the power of the desired signal to the power of the noise signal. Typically refers to the additive thermal noise impairment.
SOVA	Soft Output Viterbi Algorithm - a modification of the Viterbi algorithm that allow manipulation of soft value input and generating soft decoded output
TDE	Turbo Decoder Emulator - a system that simulate a Turbo decoder operating environment, which includes generation of inputs, encoder and decoder logics and also provides functions for statistical data collection.
VHDL	VHSIC Hardware Description Language - a language of HDL

Terms	Meaning
VIO	Virtual Input / Output - a Chipscope Pro logic block that interface between software and hardware to support virtual functions such as pushing buttons and lighting LEDs in the software interface.
ZBT	Zero-Bus Turnaround SRAM - a type of synchronous-burst SRAM memory with no bus latency and fully uses available memory bus bandwidth. This is because they do not require turnaround cycles which is the idle cycles between read and write operations.

Chapter 1: Introductions

In the recent decade much research effort has been devoted to a new class of decoder for error control called the Turbo decoder. Turbo codes were initially introduced by Berrou, Glavieux and Thitimajashima from ENST¹ in 1993 [BGT1993]. Turbo codes have an excellent error correction performance. In an additive white Gaussian noise (AWGN) channel, it can come within 0.7dB of the theoretical limit i.e. (Shannon's Limit) for channel coding. Due to its excellent high noise performance, Turbo codes has been adopted into the Third Generation (3G) wireless standard.

The originally proposed Turbo decoder theoretically required implementation of a Soft Input Soft Output (SISO) decoder using the Maximum A-Posteriori Probability (MAP) algorithm [BCJR1974]. Soft value gives the polarity of a number (e.g. -1 and +1) and a weight showing the confidence given to that polarity. After receiving soft value inputs, the MAP algorithm invokes computationally intensive processes such as multiplication and exponential functions to generate soft value output. The soft output predicts the polarity of the original data as well as giving the confidence level for the prediction. A hardware implementation of the proposed Turbo decoder was not possible until further research was made in developing hardware efficient SISO decoders. Listed in the order of increasing decoding performance, Soft Output Viterbi Algorithm (SOVA) [HH1989], Max Log-MAP [KB1990] and Log-MAP [EPG1994] are more feasible SISO algorithms, which reduce the intensive computation processes. SOVA and Max Log-MAP decoders are suboptimal compared to the the MAP decoder but a Log-MAP decoder can achieve the decoding performance of a MAP decoder with much less hardware. Hence, it was selected to be the SISO decoder of our Turbo decoder design and implementation.

1. Ecole Nationale Supérieure des Télécommunications de Bretagne.

Although implementing a Log-MAP algorithm uses less hardware than the MAP algorithm, it still requires a large memory and hardware resources for its interleaving process and iterative structure. Because the Turbo decoder is mathematically complex, software simulation is usually performed to predict its performance, especially at low bit error rates (BER). These software simulations sometimes required days to complete, which slows down the design process. This research addresses this issue with an implementation of a hardware emulator using reprogrammable logic devices.

Improvements in IC technology has also benefited the growth of reprogrammable logic. The latest Field Programmable Gate Arrays (FPGAs) have high performance, and some of these devices even contain internal microprocessors and memories. In this work, the hardware emulator makes use of these FPGA innovations for the design and verification of a 2Mbps throughput Turbo decoder. The Turbo decoder itself has a number of nontypical design features, such as a random interleaver, a multi-word 2's complement adder, an improved add-compare-select (ACS) circuit and a path metric overflow protection circuit. The design also implements a Turbo decoder optimization feature called Fixed Windowing [WLW2001], [All2002].

1.1 History of Error Control Coding

In 1948, Claude Shannon published a theorem related to channel capacity [Sha1948]. The theorem describes the maximum possible rate of data C in bits per second which can be delivered without error over a communication channel, given the available bandwidth W in hertz and the signal-to-noise ratio (SNR). This rate is shown in Equation 1.1. This is called the Shannon Limit¹.

1. The Shannon limit is the smallest SNR that allows error free communication. It is also stated as -1.59 dB which is the limit for infinite bandwidth.

$$C = W \times \log_2(1 + SNR) \quad (1.1)$$

When the actual data rate is less than the channel capacity, it is possible to use the excess bandwidth to transmit error control codes and there will be some SNR at which the bit error rate can be made arbitrarily low. The SNR for the best possible code will be Shannon limit. Researchers have worked for more than half a century to invent a coding scheme that approaches Shannon's channel capacity limit. As mentioned before, the Turbo decoder introduced in 1993 by Berrou, Glavieux and Thitimajashima [BGT1993] has achieved an excellent decoding performance within 0.7dB of the Shannon Limit.

1.2 Classification of Error Control Codes

The purpose of an Error Control Code (ECC) is to increase the probability of successful data retrieval in applications such as data storage on fragile materials and data transmission under noisy conditions. Error correction is achieved by encoding the original data to allow redundancy in transmission and decode them back during retrieval. For binary data, if the original data bits are copied unchanged to consecutive locations in the code word, then it is called a systematic code. Most, but not all, codes are systematic. There are also two main families of ECC which are block codes and convolutional codes.

Block codes consist of equal length blocks of bits called code word. Often a block is a byte. Systematic block codes concatenate the original data block with parity check data created by the encoder. Because the encoder and decoder handle a block of data at each operation, block codes allow high data throughput by performing block correction over a frame. This technique is suitable for applications such as video compression and broadcasting. Hamming Codes and Reed-Solomon Codes¹ are examples of block codes. Hamming Codes, which were created by Richard

Hamming in the 1950s, represent the first serious attempt at providing a mechanism for error correction. The term Hamming distance is commonly used to describe the difference in the number of bits between two binary sequences of equal length.

Convolutional codes correlate each bit of data transmitted with its neighboring bits to generate parity. In systematic codes, a code word consists of a systematic bit and parity bit(s). This can introduce high redundancy, and largely reduces the effect of noise corruption from the environment. Because of its high noise performance, convolutional codes are suitable for application such as wireless data transmission. Viterbi decoders and Turbo decoders are two common decoders used for codes based on convolutional codes. More details about convolutional codes will be given in later sections.

By combining the convolutional codes and the block codes in a communication system, the benefit of both can be achieved [For1966]. Convolutional codes can decode data from noisy channel with soft values and present the less corrupted binary data to a block decoder for powerful hard value¹ decoding. Figure 1.1 shows a common communication system structure where an inner ECC system use convolutional codes cascaded with an outer ECC system which can be an algebraic code such as a Reed-Solomon code. The convolutional codes provide low BER outputs and the block code further recover data by correcting blocks of data and hence reduces the frame error rate.

FIGURE 1.1. Encoders and decoders in a typical communication system.

1. Reed-Solomon Codes was invented by Irving S. Reed and Gustave Solomon from MIT in 1960.

1. 1 and -1 as opposed to soft value such as 0.9 and -0.4. Corresponding to voltages received over the channel.

1.3 Convolutional Codes

Convolutional codes correlate neighborhood data to achieve high error correcting performance. There are two major classes of decoding algorithm for convolutional codes, they are introduced in the following paragraphs.

One class of decoding algorithm uses the maximum likelihood (ML) method, which estimates the most possible original data based on a single received sequence. The Viterbi decoder is the most common ML decoder. It requires less hardware, but does not perform as well as the Turbo decoder, which uses the MAP algorithm and is the class of the decoding algorithm.

The maximum a-posteriori probability (MAP) algorithm is a conditional probability based ML method. It finds the most possible data under the influence of a prior knowledge of the original data, which is called the a-priori probability. This a-priori information can be obtained by decoding a different received sequence. This sequence can be generated by reordering the original data and create a different correlation of neighborhood data. The Turbo decoder effectively receives two data streams. The a-posteriori knowledge (extra knowledge) obtained by decoding one stream are used to provide a-priori knowledge (bias) for the decoding of the other stream. Typical terms used in convolutional codes are given in Table 1.1 and example of the usage of these terms are shown in Figure 1.2.

code rate R	Ratio of real data bit rate to the encoded data bit rate.
code word	Set of data transmitted for each encoder input.
constraint length T	The length of shift register plus 1 (for the input), which represent the number of bit affecting the output.
encoder state	The value of bits in the encoder's shift register.
generator polynomial	The description of the encoder structure usually given in octal format. For example, G7 G4. The first number 7 (octal 111) represents the shift register taps ^a used for feed back ^b

TABLE 1.1. Common terms in convolutional codes

number of encoder output k	The number of symbols per code word.
number of encoder state n	Total combination of encoder state.
parity - z_p	Redundant data generated by the encoder to improve data integrity in systematic codes.
systematic - z_s	Original data bit. It may or may not be transmitted depending on the type of encoder.
recursive systematic codes (RSC)	Defined in Section 1.3.3

TABLE 1.1. Common terms in convolutional codes

- Connection to an XOR gate, a one input XOR can be simplified as a wire as shown in Figure 1.2 for the parity bit.
- Since feed back does not appear in non-recursive encoder, it is important to specify whether or not the generator polynomial is for RSC.

FIGURE 1.2. Explanation of terms for convolutional codes on an example RSC encoder

1.3.1 Convolutional Encoder

An example convolutional encoder, such as might be used with a Viterbi decoder is shown in Figure 1.3. The encoder correlates one data bit to two of its neighboring bits, and generates two code bits to increase data transmission redundancy. Even if the code bits may corrupted by background noise from the transmission channel, their neighbouring bits will provide enough information for data recovery in the decoder. This particular convolutional encoder has a generator polynomial of G7 G5. Here G7 corresponds to the upper code bit that is generated by the XOR with connections [octal111] to the input and positions of the shift register. G5 corresponds to the lower code bit which is similarly generated by the [octal101] connections to the XOR. This encoder has a code rate of $\frac{1}{2}$ and constraint length of 3.

In G7, 7 is an octal number identifying the input connection to the upper XOR.

FIGURE 1.3. Convolutional encoder

Decoding of the data from convolutional encoder is most commonly done using the concept of a trellis¹. The trellis is derived from the encoder state graph. A trellis is constructed in Figure 1.4 for the convolutional encoder in Figure 1.3 and Figure 1.5. From the state graph, all possible state transitions from one state to the others are connected by branches across time. Each trellis branch has its own expected codeword, which is written over each branch arrow. The original encoder input is indicated by a solid or dashed arrow for the branch. Details on how the trellis is used for decoding are discussed in the sections on decoding algorithms.

In the trellis. The state graph is shown vertically. Transitions are made not to the original graph but to a copy on the right which is the state graph in the next time step.

FIGURE 1.4. Trellis construction of the G7/G5 convolutional encoder

1. A data structure that mimics the interleaving structure of the support for vines or other creeping plants.

1.3.2 Viterbi Decoder

The Viterbi algorithm was invented by Andrew Viterbi¹ in 1967 [Vit1967]. A Viterbi decoder decodes data generated by a convolutional encoder. It is a type of ML decoder which recovers the data transmission sequence through observation of a single received data sequence. The hardware implementation of this algorithm consumes less power and less area than other decoding techniques of convolutional codes.

As previously mentioned, a Viterbi decoder uses the concept of a trellis for decoding. The decoder trellis should match the encoder trellis state and transitions. The trellis diagram shown in Figure 1.5 is the same as the trellis diagram of the encoder in Figure 1.3.

FIGURE 1.5. Viterbi decoder trellis decoding

1. Viterbi was one of the founders of Qualcomm which is one of the primary initiators of the wireless CDMA standard.

Consider the encoder, going from time = t to $t+4$, encoded the sequence 1 1 0 1 0 to generate transmitted code words 11 01 01 00 10. Because of error the decoder receives the transmitted signal 11 11 01 00 10. Assume the trellis was reset to state A at the start. For the two possible transitions, AA (state A at t to state A at $t+1$) and AB (state A at t to state B at $t+1$), the expected values of the branches are compared to the received input. The Hamming distance, as mentioned before, gives a measure of how far apart two binary numbers are by comparing the bits in the same positions in the two numbers. Therefore at time = t the Hamming distance between the AA branch and the AB branch to the received code word is 2 and 0 respectively as enclosed by the rectangular boxes in Figure 1.5.

The next input at time = $t+1$ is 11 and has an error, so both possible paths are at Hamming distance 1. Because there are no 11 or 00 paths leaving state B, one must choose either 10(BC branch) or 01(BD branch). At this time either transition looks equally likely.

At time = $t+2$, if one starts from D; then the branch DC gives a Hamming distance of 0. However, if one starts from C, the Hamming distance is 1 for both CA and CB branches. Thus, although at time = $t+1$ the proper path was not obvious, at time = $t+2$, the choice is clearer. We will choose a path through the trellis based on the cumulative Hamming distance called the path metric, which is the sum of the Hamming distances as one steps along a path through the trellis.

Figure 1.5 shows how the Hamming distances accumulate as one goes down various paths through the trellis diagram. At time = $t+4$, one path has the lowest cumulative Hamming distance of 1 as indicated by the hexagonal box while the others have a distance of 3 or 4. Thus the most likely path at this point is $A \Rightarrow B \Rightarrow D \Rightarrow C \Rightarrow B \Rightarrow C$ with a cumulative distance of 1, and the corresponding output data is 11010.

Viterbi decoders find the best path in the trellis to estimate the transmission sequence. When multiple paths converge into the same state, the path with the lowest accumulative Hamming distance is kept. Therefore the number of possible paths do not exceed the number of states. When the length of the trellis path increases, the winning path will become more dominant. For practical purposes when a time of 5 times longer than the constraint length T is reached, all possible trellis paths are likely to converge to the same state [Vit1998] and the encoder input bit at time $t - 5T$ can be found through back tracing. A trellis can also be constructed for soft inputs. This increases the performance of the decoder as the quantization error is reduced. A soft value Viterbi decoder is discussed later in Section 1.3.6.

1.3.3 Recursive Systematic Codes

The recommended component encoder for a Turbo decoder consists of Recursive Systematic Codes (RSC) blocks, one of which is shown as an example in Figure 1.6. It is named recursive because feedback from the shift register is used to generate the new input. It is systematic because one bit of the codeword is always the same as the RSC input. This RSC has a generator polynomial of $G_7 G_5$. The G_7 is referring to the connection [octal111] to the input XOR from the data input and feedback from shift register. The G_5 refers to the connection [octal101] to the output XOR. Since one cannot distinguish between an RSC and a normal convolutional encoders by just the generator polynomial, it is essential to explicitly mention the type of the encoder each time a generator polynomial is used. Although this RSC has a constraint length of 3, by continuously feeding back data, data are convoluted with each other over a longer time frame. Data correlation of a RSC is more than just the constraint length of a normal convolutional encoder. The RSC block of the 3G wireless standard [3GPP2001] has a constraint length of 4.

FIGURE 1.6. Recursive systematic codes (RSC) encoder

1.3.4 Component Encoder with RSC

Two major innovations with the Turbo decoder are parallel concatenated encoders and iterative decoding. An example of an encoder used for a Turbo decoder system is shown in Figure 1.7. It is commonly referred to as a component encoder because it is constructed from two or more recursive systematic codes (RSC) encoders as its components.

The interleaver¹ increase the distance between the parity bits. This is because the parity bit in the second decoder is generated by a convolution of different companions in the encoder shift register. In Figure 1.7, the code rate is 1/3 because of the two parity bits. Depending on the application, more parity bits can be added by introducing more interleavers and RSCs. More parity bits increases the data redundancy. Hence when burst errors occur, the nearby data is less likely to be corrupted. Interleavers are commonly used with error correction codes to improve error correction performance. In fact in a low noise channel including the AWGN channel, a good interleaver enables decoders to achieve a very low BER. [BMD2003]. Because random permutation is performed on the sequence, memory is needed to store different sequences to be transmitted for synchronization purpose.

1. The interleaver randomly rearranges the order of a sequence.

FIGURE 1.7. Component encoder for Turbo decoder

1.3.5 Turbo Decoder

Similar to the Viterbi decoder, the Turbo decoder also performs well under an environment with white Gaussian noise. However, it has a better error correction performance that can approach Shannon's Limit for channel coding. Hence it has been adopted into the 3G wireless standards [3GPP2001]. Turbo decoder uses the concept of maximum a-posteriori¹ probability (MAP) decoding. However, a Turbo decoder requires more hardware, and thus consumes more power than a Viterbi decoder.

Iterative² decoding process is an important innovation of Turbo decoding. A typical Turbo decoder has two trellis decoders working alternatively and use the a-posteriori knowledge generated by one decoder as the a-priori³ of the other decoder. This iterative structure allows improvement on data recovery rate.

-
1. A-posteriori probability represents those knowledge obtained by observing experimental result.
 2. The Turbo decoder get its name because its iterative structure resembles a turbo charged automotive engine.
 3. A-priori probability represents those knowledge obtained by theory. In Turbo decoder, it is obtained by interleaving or de-interleaving the a-posteriori data from the other decoder.
-

A typical Turbo decoder is shown in Figure 1.8. The decoder blocks use a trellis somewhat like the Viterbi decoder, but with soft input and soft output (SISO). The first and second decoder blocks are designed to decode the first and second RSCs in the encoder respectively. The Interleavers and the De-Interleavers perform the same permutation used in the encoder.

FIGURE 1.8. Error correction codes system using component encoder and Turbo decoder

When data arrives, it is first stored in memory. As mentioned before, Turbo decoder uses iteration for decoding. In the beginning of the first iteration, the a-priori1 data is not available, and it is set to zero. Thus, only the systematic and the parity1 data are used by decoder1 to calculate the a-posteriori1 data. A-posteriori1 data from decoder1 becomes a-priori2 data after interleaving, and it together with parity2 data and the interleaved systematic data, are used by decoder2 to calculate a-posteriori2 data. Again the de-interleaved¹ a-posteriori2 data becomes the a-priori1 data for decoder1 and the first iteration is finished. A Turbo decoder goes through several iterations before the final data output can be retrieved.

1. The de-interleaver reorder the data to its original sequence corresponding the order of data sequence encoded

There has been much literature published about the SISO decode algorithm since the introduction of the Turbo decoder. Examples of these decode algorithms are MAP, Max-Log-MAP, Log-MAP and SOVA. Descriptions of the SOVA algorithm are in Section 1.3.6 while the rest of the SISO decoders will be discussed in detail in Chapter 2 and Chapter 3.

1.3.6 Soft Output Viterbi Algorithm (SOVA)

The disadvantage of the Viterbi algorithm is that only hard symbol values such as $+1/-1$ are generated. There is no indication of the confidence level of the output. Hence it cannot be used as a low-level decoder in a Turbo decoder. Researchers have modified the Viterbi algorithm to create a soft output Viterbi algorithm (SOVA) [HH1989]. SOVA takes a-priori information when performing branch metric calculations, and it also provides the Log Likelihood Ratio (LLR) as the output. Hence SOVA, unlike Viterbi, is an MAP decoding algorithm.

When two paths merge in the trellis, SOVA discards the path that has a lower path metric just like a Viterbi decoder. However, SOVA stores the difference of the merging path metrics to represent the certainty of the decision. These path differences are stored for each state along the survivor path. When back tracing happens, the product of multiplying the hard output by the minimum path differences along the survivor path becomes the LLR approximation.

The SOVA has a lower error correction performance than the theoretically optimal MAP algorithm. However, since the SOVA was already available when the Turbo decoder was introduced, it is being considered as the first low-level SISO decoder for the hardware implementation of the Turbo decoder [BGT1993] due to its smaller hardware requirements than the MAP decoder.

1.4 Thesis Objectives

There are two main objectives in this research. The first objective is to implement a Turbo decoder hardware efficiently. Techniques such as fixed windowing are used to improve the hardware design and implementation. The Turbo decoder should allow users to have real time configuration options such as the channel reliability, while maintaining a good performance.

The second objective is to construct a hardware emulator for verification of Turbo encoder/decoder system. Traditionally, when designing a high performance communication system, a system designer implements the design in a software model to perform system verification. This system can be set up using Matlab, which allows a shorter implementation time due to the availability of many supported functions in the mathematical and communication libraries. However, the Matlab models are not optimized, and allow minimal configuration options. A proper BER and FER simulation with large frame size and frame number¹ can take days to finish. A more flexible and efficient model can be built with a lower level programming language such as C/C++. Since a CPU² runs software instructions sequentially, using a C/C++ simulation model can still take a long time. In general, the more frames used in the simulation, the more accurate the result becomes but the simulation time will be extended. A solution to this, which is widely adopted in the industry, is to use a development platform with programmable logic. By building a hardware Turbo codes emulator, the time required to simulate enough data frames to perform useful BER experiments can be reduced to minutes. Also, because the code used for implementing the programmable logic can be easily transferred to an ASIC, the simulation platform can also be used to perform a functional hardware verification.

-
1. Multiple frames are usually needed for multiple runs of simulation to increase the accuracy of the simulation result
 2. Central processing unit

1.5 Thesis Outline

This thesis first assists readers with limited communication theory background to understand the Log-MAP algorithm, and then introduces them to the design and optimization of the Turbo decoder on the hardware emulator. The organization is as follows:

In Chapter 2, the MAP decoding theory which provide the basis of Turbo decoding is reviewed. The derivation and calculation of a-posteriori information for the use of message passing decoding is introduced.

In Chapter 3, the evolution of Log-MAP trellis decoder algorithms is discussed theoretically through the introduction of two other SISO decoders which are MAP and Max-LogMAP decoders. Detailed derivations of the theoretical equations are provided.

In Chapter 4, a demonstration of Turbo decoding with detailed procedures and numerical examples is given. This chapter provides an explanation on all steps in a Turbo decoder system, including the construction of the component encoder, trellis decoder and iterations.

In Chapter 5, the software simulated performance of the proposed Turbo decoder design is outlined. Factors affecting performance of the Turbo decoder are quantitatively discussed through simulation. These factors allow effective configuration of the Turbo decoder.

In Chapter 6, details on the hardware design on the proposed Turbo decoder are shown. Block diagrams, state diagrams and memory maps for each specific block are given. The hardware processes involve in encoding and decoding are outlined.

Finally, Chapter 7 presents the conclusions of this research work. The performance of the Turbo decoder is compared with other published results. The improvement in simulation time on hardware versus software simulation is summarized. Possible research extensions are also discussed.

Chapter 2: Maximum A-Posteriori Probability Decoding With Iterations

Turbo decoder was evolved from the classic Maximum A-Posteriori Probability (MAP) algorithm [BCJR1974]. The MAP algorithm estimates the most likely information data that has been transmitted through observing the sequence received given the a-priori data. It compares the likelihood of the possible values of the data to determine the Likelihood Ratio, which can be used to find the most probable input value, and to determine how reliable the estimate is.

2.1 Log Likelihood Ratio Calculation

Decoding using the likelihood ratio is illustrated in Figure 2.1. Assume that there is no encoding done and a BPSK¹ modulation scheme is used. The transmission of a logical zero is modulated to a signal -1 and a logical one is modulated to a signal +1. After noise corruption, the value of y_t represents what the decoder has received from the modulator at time t . The two curves give the probable values of u_t from y_t received over a noisy channel.

FIGURE 2.1. Decoding based on likelihood of data received

A signal is received and is equal to y_t . The original transmitted signal could have been ± 1 . In Figure 2.1, one would intuitively estimate the signal is a +1. Mathematically, the decision of whether the original data is a logical one or logical zero is determined by the conditional probab-

1. Binary Phase-Shift Keying, a type of modulation scheme in digital communication where the phase of the RF carrier is shifted 180 degrees in accordance with the two possible symbol {0,1} of a data bit.

ity $P(u_t = -1|y_t = 0.4) = \lambda_2$ and the probability of $P(u_t = 1|y_t = 0.4) = \lambda_1$. If $\lambda_2 > \lambda_1$, or y_t is in the left side of γ_t then the original data is likely to be a zero; If $\lambda_2 < \lambda_1$, or y_t is in the right side of γ_0 then the original data is likely to be a one. These probabilities are called the likelihood probabilities and the ratio between the two is the Likelihood Ratio as shown in Equation 2.1.

$$\text{Likelihood Ratio} = \frac{P(u_t = 1|y_t)}{P(u_t = -1|y_t)} \quad (2.1)$$

In the MAP algorithm, the whole received sequence Y is used to recover the original data. Therefore the probability is conditional on the whole received sequence Y . Hence Equation 2.1 can be rewritten as:

$$\text{Likelihood Ratio} = \frac{P(u_t = 1|Y)}{P(u_t = -1|Y)} \quad (2.2)$$

With an encoder which transmits a sequence of code words:

$$Y = (y_0, y_1, \dots, y_t, \dots, y_{(L-1)}) \quad (2.3)$$

Each code word for an RSC encoder contains systematic and parity bit:

$$\tilde{y}_t = (y_{st}, y_{pt}) \quad (2.4)$$

Here y_{st} and y_{pt} are the systematic and parity data generated by an RSC encoder at time t and L is the number of data inputs to the RSC encoder. If the fraction in Equation 2.2 is larger than 1, then the original data u_t is likely to be 1, else if the fraction is smaller than 1 the original data is likely to be 0. However the result from Equation 2.2 does not have a comparable scale for both 1/-1 decoder outputs. By using the natural log function with the fraction as shown in Equation 2.2, the result can be made symmetric. Hence the resulting ratio is called the log likelihood ratio (LLR).

$$LLR = L(u_t|Y) = \ln\left(\frac{P(u_t = 1|Y)}{P(u_t = -1|Y)}\right) \quad (2.5)$$

The sign of the LLR is an estimation of the sign of the modulated signal and the magnitude of LLR gives an estimate of confidence in the decision.

2.2 A-Posteriori Data Calculation for Iterations

With two sequences of code words, one is a permutation of the other, decoding is performed using two MAP decoders, one for each sequence. Knowledge gained from decoding one sequence can be used to bias the decoding of the other sequence to obtain new information about the original data. The new knowledge gained can then be used to bias the second decoding run. This process, which is called iteration¹, can be used to obtain a high decoding performance.

Consider Figure 2.2. After the first decoding process by decoder1, extra knowledge is gained about the original data. The extra knowledge gained is the extrinsic information $L_e(u_i)$ which is also called the a-posteriori data. A-posteriori data provides a correction to the its received systematic input y_{si} . The interleaved² a-posteriori data $L_e(u_{\pi(i)})$ from decoder1 then becomes the a-priori data $L(u_i)'$ of decoder2 and can be used as the basis for its decoding process.

A-priori data $L(u_i)'$ provide a bias for decoding with a second parity data³ y_{pi}' and an interleaved systematic data y_{si}' in decoder2. The output of decoder 2 is another a-posteriori data $L_e(u_i)'$ which gives even better knowledge about the original systematics data. This a-posteriori data is then de-interleaved⁴ $L_e(u_{\pi^{-1}(i)})'$ and become the a-priori data $L(u_i)$ of decoder1. Decoder1 repeats the decoding process, this time with the extra knowledge gained. After each decoding iteration, knowledge about the original data is generally improved.

-
1. It is also called the message passing technique for decoding.
 2. Interleaving is done by the random permutation function $\pi(i)$, which selectively rearranges the order of a sequence.
 3. The second parity data y_{pi}' is originated from the RSC output x_{pi}' of the second component encoder. In the encoder, this RSC output x_{pi}' is created by encoding the interleaved values of the systematic data x_{si} .
 4. De-interleaving is done by doing inverse permutation, i.e. $\pi^{-1}(i)$, which reorganizes the sequence to its original order as seen by the encoder.
-

FIGURE 2.2. Iterative Turbo decoder structure

The Turbo decoder uses iterations to achieve high decoding performance. To determine the a-posteriori data of each decoding run, it has been shown in [BGT1993] that the output LLR in Equation 2.5 can be re-written as in Equation 2.6 and in Chapter 3.

$$L(u_t|Y) = L(u_t) + L_c \cdot y_{st} + L_e(u_t) \quad (2.6)$$

where at time t , u_t is the original data, $L(u_t)$ is the a-priori data, y_{st} is the systematic input to the decoder, $L_e(u_t)$ is the a-posteriori data obtained from the current decoding run.

Channel reliability L_c is used to weight the reliability of the systematic input and it is calculated as in Equation 2.7.

$$L_c = \frac{2a}{\sigma^2} \quad (2.7)$$

a is the fading amplitude over the channel. In the scope of this research only the AWGN channel is considered, therefore $a = 1$. The noise power σ^2 is measured from the channel.

Since $L(u_t|Y)$ is the output from each decoder after the decoding run, to calculate the a-posteriori information and feed these as the a-priori information for the next decoding run, one can reorganize Equation 2.6 to generate $L_e(u_t)$ as in Equation 2.8.

$$L_e(u_t) = L(u_t|Y) - L(u_t) - (L_c \cdot y_{st}) \quad (2.8)$$

2.3 Trellis Decoding Using the MAP Algorithm

The method to determine the likelihood of data in the MAP algorithm is evolved from the trellis decoding method of the Viterbi decoder. In convolutional codes, the most common way to determine the likelihood of the original data is through trellis decoding. Since the encoder encodes the original data to generate parity bits, the received parity bits are compared with their expected value in the trellis. The result of these comparisons can be used to estimate the value of the original data.

A basic probability analysis of a trellis is shown in Figure 2.3. A typical state A_t can be reached from two previous state A_{t-1} and B_{t-1} .

Let : α_{A_t} be the probability of being in A_t ,
 $\alpha_{A_{t-1}}$ be the probability of being in A_{t-1} ,
 $\alpha_{B_{t-1}}$ be the probability of being in B_{t-1} ,
 γ_{AA_t} be the probability of a branch transition from A_{t-1} to A_t and
 γ_{BA_t} be the probability of a branch transition from B_{t-1} to A_t .

FIGURE 2.3. Probability decoding on trellis

Then the probability of being in A_t is:

$$\alpha_{At} = \alpha_{At-1} \cdot \gamma_{AA_{t-1}} + \alpha_{Bt-1} \cdot \gamma_{BA_{t-1}}. \quad (2.9)$$

In general, for any state S , assume the previous state is S' and $S, S' \in \{A, B, C, D, \dots, S_n\}$:

$$\alpha_{St} = \sum_{S' \rightarrow S} \alpha_{S't} \cdot \gamma_{S'S_t} \quad (2.10)$$

Note that if there is no transition arrow between two states, then $\gamma_{S'S_t} = 0$.

In trellis decoding using the MAP algorithm, the probability of the value of the original data u_t , is dependent on the types of state transition, which are the branches of the trellis. For example, in Figure 2.3, one would have had a 0 input for a transfer $S' \rightarrow S$ when any of:

$A_{t-1} \rightarrow A_t, B_{t-1} \rightarrow A_t, C_{t-1} \rightarrow B_t$ or $D_{t-1} \rightarrow B_t$ transition happened.

Thus the probability that the original data $u_t = 0$ based on the whole received sequence Y is:

$$P(u_t = 0|Y) = \sum_{\substack{S' \rightarrow S \\ u_t = 0}} P(S' \rightarrow S|Y) \quad (2.11)$$

And the probability that the original data $u_t = 1$ based on the whole received sequence Y is:

$$P(u_t = 1|Y) = \sum_{\substack{S' \rightarrow S \\ u_t = 1}} P(S' \rightarrow S|Y) \quad (2.12)$$

Using Figure 2.3, the probability of the original data being a 0 is then:

$$P(u_t = 0|Y) = \sum_{\substack{A_{t-1} \rightarrow A_t \\ B_{t-1} \rightarrow A_t \\ C_{t-1} \rightarrow B_t \\ D_{t-1} \rightarrow B_t}} P(S' \rightarrow S|Y) \quad (2.13)$$

and the probability of the original data being a 1 in Figure 2.3 is:

$$P(u_t = 1|Y) = \sum_{\substack{A_{t-1} \rightarrow C_t \\ B_{t-1} \rightarrow C_t \\ C_{t-1} \rightarrow D_t \\ D_{t-1} \rightarrow D_t}} P(S' \rightarrow S|Y) \quad (2.14)$$

α , called the path metric, is the probability of reaching a particular state S when traversing the trellis from the beginning at $t = 0$. This depends on the received code words $y_0, y_1, \dots, y_{t-1}$ on the trellis path.

γ , called the branch metrics, is the probability of having a state transition to a particular state S from a previous state S' given the current input y_t . This probability can be biased by the a-priori data $L(u_t)$ which is the extrinsic information $L_e(u_t)$ gained from the other decoder (refer to Figure 2.2).

β , called the reverse path metrics, is the probability of reaching a particular state S when traversing the trellis backward from the end at $t = L - 1$. Traversing the trellis from the end using future code words $y_{t+1}, \dots, y_{(L-1)}$ is possible because the received data will first be stored in memory before processing.

In Chapter 3, the methods of using and calculating a-priori $L(u_t)$, α , γ and β in a SISO decoder which include MAP decoders, Max-Log-MAP decoders, and Log-MAP decoders are given. In Chapter 4, a detailed example of using a Log-MAP decoder for SISO decoding in a Turbo decoder is illustrated.

Chapter 3: Soft Input Soft Output Decoders

There are several common SISO decoder algorithms used for implementing the low level decoder in a Turbo decoder. The SOVA variant of Viterbi algorithms, and the variants of MAP such as Max-Log-MAP and Log-MAP all have less complicated architectures than MAP, but only the Log-MAP algorithm can match the decoding performance of the MAP algorithm. The MAP, Max-Log-MAP and Log-MAP algorithms will be discussed in detail in this section.

3.1 MAP Algorithm

As described in Equation 2.5, the LLR is determined by comparing the likelihood of receiving a bit given that the original data was one and the likelihood of receiving a bit given that the original data was zero. To determine these likelihoods, Equation 2.5 is written in terms of the α , γ and β to give Equation 3.1.

$$LLR = L(u_t|Y) = \ln\left(\frac{P(u_t = 1|Y)}{P(u_t = -1|Y)}\right) \quad (\text{Equation 2.5 repeated})$$

$$L(u_t|Y) = \ln\left(\frac{\sum_{\substack{u_t = 1 \\ S' \rightarrow S}} \alpha_{S'_{t-1}} \cdot \gamma_{S'S_t} \cdot \beta_{S_t}}{\sum_{\substack{u_t = -1 \\ S' \rightarrow S}} \alpha_{S'_{t-1}} \cdot \gamma_{S'S_t} \cdot \beta_{S_t}}\right) \quad (3.1)$$

States S and S' are each one of the n trellis states. The previous state of S is indicated by S' . Here, n is the number of states in the trellis and is equal to 4, i.e. $S, S' \in \{A, B, C, D\}$. A state transition from left to right is indicated by $S' \rightarrow S$. Similarly $S' \leftarrow S$ indicates a back trace state transition from right to left. The calculations of α and β are performed iteratively using probability theory as is illustrated graphically in Figure 3.1.

$$\alpha_{S_t} = \sum_{S' \rightarrow S} \alpha_{S'_{t-1}} \cdot \gamma_{S'S_t} \quad (3.2)$$

$$\beta_{S_t} = \sum_{S' \leftarrow S} \gamma_{S'S_{t+1}} \cdot \beta_{S_{t+1}} \quad (3.3)$$

If one specifies the trellis initial state $S_0 = A$ and end state $S_L = A$, then the initial value of the probability α and β can be specified as:

$$\alpha_{s_0} = 0 \text{ and } \beta_{s_L} = 0 \text{ for all } S \text{ except when } S = A \quad (3.4)$$

FIGURE 3.1. Recursive calculation of α_{At} and β_{At}

Calculation of the transition probability $\gamma_{S'S_t}$ is crucial since it is used recursively to construct the path metrics α_{S_t} and β_{S_t} throughout the trellis. Evaluating the transition probability is done by considering the possibility of a branch transition at time t . Each branch transition is defined by its initial state S' , its received code word $\underline{y}_t$, and its destination state S . Hence the branch probability is the probability of arriving at state S with the code word $\underline{y}_t$ given the initial state S' , i.e.

$$\gamma_{S'S_t} = P(S, \underline{y}_t | S') \quad (3.5)$$

By using the properties of conditional probability:

$$\begin{aligned} \gamma_{S'S_t} &= P(S, \underline{y}_t | S') = \frac{P(S, \underline{y}_t, S')}{P(S')} = \frac{P(S, \underline{y}_t, S')}{P(S, S')} \cdot \frac{P(S, S')}{P(S')} \\ \gamma_{S'S_t} &= P(\underline{y}_t | S, S') \cdot P(S | S') \end{aligned} \quad (3.6)$$

From Equation 3.6, the branch transition probability $\gamma_{S'S_t}$ is equal to the products of two terms. The first term is the probability of receiving the code word $\underline{y}_t$ given the transition from $S' \rightarrow S$. The second term is the probability of arriving at state S given the previous state S' . In the beginning of the decoding process without any prior knowledge of the inputs, the two possible transitions out of state S' are equally probable for a well designed system [Ham1986]. Therefore $P(S|S')$ is equal to 1/2. After the first decoding however, the output of the other decoder is expected to provide the a-priori data $P(u_t)$ to bias one transition over the other. Hence for all decoding processes except the first one,

$$P(S|S') = P(u_t) \quad (3.7)$$

Now consider the first term $P(\underline{y}_t|S, S')$ of Equation 3.6. Since each branch $S'S$ has its expected code word, it can be rewritten as:

$$P(\underline{y}_t|S, S') = P(\underline{y}_t|\underline{x}_t) \quad (3.8)$$

Therefore, Equation 3.6 becomes,

$$\gamma_{S'S_t} = P(\underline{y}_t|\underline{x}_t) \cdot P(u_t) \quad (3.9)$$

Note that, $\underline{y}_t = (y_{st}, y_{pt})$, therefore:

$$P(\underline{y}_t|\underline{x}_t) = P(y_s|x_s) \cdot P(y_p|x_p) \quad (3.10)$$

And generally, for k symbols in a code word,

$$P(\underline{y}_k|\underline{x}_k) = \prod_{l=1}^k P(y_{lt}|x_{lt}) \quad (3.11)$$

Due the noise in the channel, the value of the symbols in the received code word have some type of randomness. Assuming the channel has AWGN with BPSK¹ modulation. The possible values of y_l given x_l are shown in Figure 3.2.

1. Binary Phase-Shift Keying, see Section 2.1 for an explanation

The probability of receiving a value y_l given the original bit x_l was -1 .

The probability of receiving a value y_l given the original bit x_l was +1 .

FIGURE 3.2. Distribution of probable value of the received data y_l

Using the normal distribution function to calculate probability, Equation 3.11 then becomes:

$$P(\underline{y}_t | \underline{x}_t) = \prod_{l=1}^k P(y_{lt} | x_{lt}) = \prod_{l=1}^k \frac{1}{\sqrt{2\pi}\sigma} e^{-\left(\frac{E_b R}{2\sigma^2} (y_{lt} - ax_{lt})^2\right)} \quad (3.12)$$

where k is the number of data bits in the received code word. The RSC¹ decoder shown in Figure 1.6 has $k = 2$, thus y_l represents the systemic bits y_s , and the parity bits y_p . The noise power is indicated by σ^2 . The energy per message bit E_b is multiplied by the code rate which is the same as the energy per message symbol E_s . Here E_s will be made one. The fading amplitude a is equal to one in an AWGN channel as mentioned before. Finally Equation 3.12 is simplified and substituted into Equation 3.9:

$$\gamma_{SSl} = \left(\prod_{l=1}^n \frac{1}{\sqrt{2\pi}\sigma} e^{-\left(\frac{1}{2\sigma^2} (y_{ll} - x_{ll})^2\right)} \right) \cdot P(u_t) \quad (3.13)$$

Figure 3.3 gives an example of how γ_{SSl} is calculated numerically.

1. Recursive Systematic Codes, refer to earlier chapter for explanation.

A-priori $P(u_t = -1) = 0.4$

Received Codeword $(y_{pt}, y_{st}) = (-0.9, 0.1)$

FIGURE 3.3. Numerical calculation of $\gamma_{S't}$ using the MAP algorithm

MAP decoding is a computational intensive process involving multiplication and using the exponential function. Hence it is very difficult to implement efficiently in hardware.

3.2 Max-Log-MAP Algorithm

In order to reduce the complexity of the MAP algorithm, one can use logarithms to convert multiplication to addition. One such simplified method is the Max-Log-Map algorithm. By defining new symbols:

$$\alpha_{S't} = e^{A_{S't}} \quad \text{i.e.} \quad A_{S't} = \ln(\alpha_{S't}) \quad (3.14)$$

$$\beta_{S't} = e^{B_{S't}} \quad \text{i.e.} \quad B_{S't} = \ln(\beta_{S't}) \quad (3.15)$$

$$\gamma_{S'S't} = e^{\Gamma_{S'S't}} \quad \text{i.e.} \quad \Gamma_{S'S't} = \ln(\gamma_{S'S't}) \quad (3.16)$$

One can apply the approximation of Equation 3.17 to calculate the path metrics. As an example, when $a > b$, then $e^a \gg e^b$. An exponential value increases rapidly when its argument increases.

Therefore when adding exponential terms together, the term with the highest index can approximate the sum.

$$\ln\left(\sum_i e^{x_i}\right) \approx \ln(e^{\max(x_i)}) = \max(x_i) \quad (3.17)$$

Convert Equation 3.2, Equation 3.3 and Equation 3.13 for α_{S_t} and β_{S_t} to:

$$\begin{aligned} A_{S_t} &= \ln(\alpha_{S_t}) \\ &= \ln\left(\sum_{S' \rightarrow S} \alpha_{S' \rightarrow S} \cdot \gamma_{S'S_t}\right) \\ &= \ln\left(\sum_{S' \rightarrow S} e^{A_{S' \rightarrow S} + \Gamma_{S'S_t}}\right) \\ A_{S_t} &\approx \max_{S' \rightarrow S} (A_{S' \rightarrow S} + \Gamma_{S'S_t}) \end{aligned} \quad (3.18)$$

Similarly,

$$\begin{aligned} B_{S_t} &= \ln(\beta_{S_t}) \\ &= \ln\left(\sum_{S' \leftarrow S} \gamma_{S'S_t+1} \cdot \beta_{S_t+1}\right) \\ &= \ln\left(\sum_{S' \leftarrow S} e^{\Gamma_{S'S_t+1} \cdot B_{S_t+1}}\right) \\ B_{S_t} &\approx \max_{S' \leftarrow S} (B_{S_t+1} + \Gamma_{S'S_t+1}) \end{aligned} \quad (3.19)$$

Equation 3.18 is essentially what is being used in the path metrics calculation of a Viterbi decoder. The difference between the Viterbi algorithm and the Max-Log-MAP algorithm is how the branch metrics are calculated. Equation 3.20 shows the result of derivation of calculating the branch metrics based on Equation 3.13 and Equation 3.17.

$$\begin{aligned}
\Gamma_{S'S_t} &= \ln(\gamma_{S'S_t}) \\
&= \ln \left(\left(\prod_{l=1}^n \frac{1}{\sqrt{2\pi\sigma}} e^{-\left(\frac{1}{2\sigma^2}(y_{lt}-x_{lt})^2\right)} \right) \cdot P(u_t) \right) \\
&= \ln \left(\frac{1}{\sqrt{2\pi\sigma}} \right) + \ln \left(e^{-\left(\frac{1}{2\sigma^2}(y_{st}-x_{st})^2\right)} \right) + \ln \left(e^{-\left(\frac{1}{2\sigma^2}(y_{pt}-x_{pt})^2\right)} \right) + \ln(P(u_t)) \\
&= \ln \left(\frac{1}{\sqrt{2\pi\sigma}} \right) - \frac{1}{2\sigma^2}((y_{st}-x_{st})^2 + (y_{pt}-x_{pt})^2) + \ln(P(u_t)) \\
&= \ln \left(\frac{1}{\sqrt{2\pi\sigma}} \right) - \frac{1}{2\sigma^2}(y_{st}^2 - 2y_{st}x_{st} + x_{st}^2 + y_{pt}^2 - 2y_{pt}x_{pt} + x_{pt}^2) + \ln(P(u_t)) \\
&= \underbrace{\ln \left(\frac{1}{\sqrt{2\pi\sigma}} \right) - \frac{1}{2\sigma^2}(x_{st}^2 + x_{pt}^2 + y_{st}^2 + y_{pt}^2)}_{\Gamma_{S'S_t} \approx C(t)} + \underbrace{\frac{1}{\sigma^2}(y_{st}x_{st} + y_{pt}x_{pt})}_{\frac{1}{2}u_t L(u_t)} + \underbrace{\ln(P(u_t))}_{\frac{L_c}{2} \sum_{l=1}^t y_{lt}x_{lt}}
\end{aligned}$$

(This transition is thoroughly explained in [Wil1998], equation 17 to 19)

$$\Gamma_{S'S_t} \approx C(t) + \frac{1}{2}u_t L(u_t) + \frac{L_c}{2} \sum_{l=1}^t y_{lt}x_{lt} \quad (3.20)$$

Because x_{lt}^2 is always 1 and y_{lt}^2 is the same across all branches at the same time t , $C(t)$ is a constant added to all branches. It is the same for $u_t = 1$ and $u_t = -1$ and these will cancel when they are subtracted in Equation 3.21. Equation 3.20 shows an extension to the Viterbi algorithm where the a-priori information $L(u_t)$ is considered and the sum of comparing the input code word to the expected code word $\sum_{l=1}^n y_{lt}x_{lt}$ is weighted by the channel reliability L_c .

Again using the approximation Equation 3.17 and the LLR calculation of the MAP algorithm Equation 3.1, the LLR calculation of Max-Log-MAP is determined as in Equation 3.21.

$$\begin{aligned}
L(u_t|Y) &= \ln \frac{\sum_{\substack{u_t=1 \\ s' \rightarrow s}} \alpha_{s_{t-1}} \cdot \gamma_{s's_t} \cdot \beta_{s_t}}{\sum_{\substack{u_t=-1 \\ s' \rightarrow s}} \alpha_{s_{t-1}} \cdot \gamma_{s's_t} \cdot \beta_{s_t}} \\
&= \ln \left(\sum_{\substack{u_t=1 \\ s' \rightarrow s}} \alpha_{s_{t-1}} \cdot \gamma_{s's_t} \cdot \beta_{s_t} \right) - \ln \left(\sum_{\substack{u_t=-1 \\ s' \rightarrow s}} \alpha_{s_{t-1}} \cdot \gamma_{s's_t} \cdot \beta_{s_t} \right) \\
&= \ln \left(\sum_{\substack{u_t=1 \\ s' \rightarrow s}} e^{A_{s_{t-1}} + \Gamma_{s's_t} + B_{s_t}} \right) - \ln \left(\sum_{\substack{u_t=-1 \\ s' \rightarrow s}} e^{A_{s_{t-1}} + \Gamma_{s's_t} + B_{s_t}} \right) \\
L(u_t|Y) &\approx \max_{\substack{u_t=1 \\ s' \rightarrow s}} (A_{s_{t-1}} + \Gamma_{s's_t} + B_{s_t}) - \max_{\substack{u_t=-1 \\ s' \rightarrow s}} (A_{s_{t-1}} + \Gamma_{s's_t} + B_{s_t}) \quad (3.21)
\end{aligned}$$

The Max-Log-MAP algorithm calculated the LLR by subtracting the path metrics of the largest -1 branch from the path metrics of the largest 1 branch. The final LLR output not only can be used for determining the hard output, but it also tells the confidence level of the result. A larger value of LLR means a higher confidence of recovering the original data.

The approximation used in Equation 3.17 is a simplification of Jacobian logarithm which can actually give an exact conversion of the logarithmic equation. This method is what the Log-MAP algorithm is based on to achieve high decoding performance.

3.3 Log-MAP Algorithm

The computational complexity for the MAP algorithm can be reduced by operating in the Log domain as demonstrated in Section 3.2. In the Max-Log-MAP algorithm, the approximation

Equation 3.17 is used to achieve suboptimal error correcting performance. The loss in performance can be recovered by using the Jacobian logarithm formula which performs the exact conversion of $\ln(e^{x_1} + e^{x_2})$ as shown in Equation 3.22. The Log-MAP algorithm uses this exact conversion, and is able to match the performance of the MAP algorithm with much reduced computational cost [RVH1995].

$$\begin{aligned}\ln(e^{x_1} + e^{x_2}) &= \max(x_1, x_2) + \log(1 + e^{-|x_1 - x_2|}) \\ &= \max(x_1, x_2) + f_c(|x_1 - x_2|)\end{aligned}\quad (3.22)$$

The correction factor $f_c(|x_1 - x_2|)$ can be easily obtained either by a small look-up table or combinational logic. A small look-up table is sufficient, because as the difference between the two parameters increases, the value of the correction factor decreases and eventually becomes negligible. This is shown in Table 3.1. When delta $\Delta = |x_1 - x_2|$ is larger than 4.5 at an accuracy requirement of 0.01, the resulting correction factor can be omitted. Large deltas happens on high channel SNR or when several iterations are performed between multiple decoder components.

Δ	0	0.5	1	1.5	2	2.5	3
$\ln(1 + e^{- \Delta })$	0.693147	0.474077	0.313262	0.201413	0.126928	0.07889	0.048587
Δ	3.5	4	4.5	5	5.5	6	6.5
$\ln(1 + e^{- \Delta })$	0.02975	0.01815	0.011048	0.006715	0.004078	0.002476	0.001502

TABLE 3.1. Correction factors for Log-MAP decoder

Based on Equation 3.22, the calculation of A_{S_t} and B_{S_t} can be derived as shown in Equation 3.23 and Equation 3.24.

$$A_{S_t} = \max_{S' \rightarrow S} (A_{S'_t-1} + \Gamma_{S'S_t}) + f_c \quad (3.23)$$

$$B_{S_t} = \max_{S \rightarrow S'} (B_{S'_t+1} + \Gamma_{S'S_t+1}) + f_c \quad (3.24)$$

The correction factor f_c is calculated by using the difference of the two different path metrics that reach a particular state.

The branch metrics calculation, i.e. $\Gamma_{s's_t}$, of the Log-MAP decoder can be calculated using Equation 3.25 which is simplified from Equation 3.20:

$$\Gamma_{s's_t} \approx C(t) + \frac{1}{2}u_t L(u_t) + \frac{L_c}{2} \sum_{l=1}^n y_{lt} x_{lt} \quad (\text{Equation 3.20 repeated})$$

$$\Gamma_{s's_t} \approx \frac{1}{2}u_t L(u_t) + \frac{L_c}{2} \sum_{l=1}^n y_{lt} x_{lt} \quad (3.25)$$

The method to calculate the final LLR output of the Log-MAP decoder is also the same as in a Max-Log-MAP decoder and as shown in Equation 3.21:

$$L(u_t|Y) \approx \max_{\substack{u_t = 1 \\ s' \rightarrow s}} (A_{s't-1} + \Gamma_{s's_t} + B_{s_t}) - \max_{\substack{u_t = -1 \\ s' \rightarrow s}} (A_{s't-1} + \Gamma_{s's_t} + B_{s_t}) \quad (\text{Equation 3.21 repeated})$$

Finally, one needs to obtain a-posteriori data with Equation 2.8 for the use of other Log-MAP decoder for iteration as mentioned in Chapter 2.

$$L_e(u_t) = L(u_t|Y) - L(u_t) - (L_c \cdot y_{st}) \quad (\text{Equation 2.8 repeated})$$

With a simpler mathematical operation, a Log-MAP decoder has an unnoticeable degradation in performance when compared to a MAP decoder [RVH1995]. Hence it is common to refer to a Log-MAP decoder as a MAP decoder because of their mathematical and performance equivalency.

Chapter 4: Turbo Decoder With Log-Map Algorithm Demonstration

In this chapter, a step by step example of a Turbo decoder using the Log-MAP algorithm is demonstrated. Table 4.1 lists the parameters used in this example.

Channel	AWGN@ Eb/No = 1.5dB
Modulation	BPSK
Quantization	No quantization
Component encoders	2 identical RSCs
Component encoder code rate R	1/3 non-puncture ^a
RSC encoder polynomial	$G_0 = 7, G_1 = 4$, ^b
Decoder algorithm	Log-MAP
Interleaver type	Random permutation, L = frame size
Frame size L	32 bits
Number of iterations I	4
Method of termination	Single trellis terminated ^c

TABLE 4.1. Parameters of Turbo decoder example using Log-MAP algorithm

- a. Puncturing is explained in Section 5.4.
- b. Generator polynomial is explained in Section 1.3.3.
- c. Termination can only be done on the non-interleaved data because termination sequence used for decoder 1 will not be able to terminate decoder 2 after interleaving

The first step of constructing the trellis for Log-MAP decoding is to derive the state diagram from the encoder structure as shown in Figure 4.1. Notice the transition of parameters name and value as they appeared in the encoder and the decoder are outlined in Table 4.2.

Encoder parameters	Decoder parameters
$d_i \in \{0, 1\}$	$u_i \in \{1, -1\}$
$z_{si} \in \{0, 1\}$	$x_{si} \in \{1, -1\}$
$z_{pi} \in \{0, 1\}$	$x_{si} \in \{1, -1\}$

TABLE 4.2. Parameter transitions outline

FIGURE 4.1. Trellis construction for Log-MAP decoder

As shown in the Turbo decoder in Figure 1.8, the received data is stored in memory. In this example, since the code rate is 1/3, one systematic bit and two parity bits are received for each information bit. Decoder1 processes the systematic bit and the first set of parity bits using the trellis defined. A_{S_t} , B_{S_t} , $\Gamma_{S_{S_t}}$ and LLR are calculated using the methods described in Section 3.3. A-posteriori output is calculated using Equation 2.8.

4.1 Sample $\Gamma_{S_{S_t}}$ Calculations

Decoding using Log-MAP algorithm involves many branch metric $\Gamma_{S_{S_t}}$ calculations using Equation 3.25. As mentioned in Section 3.2, the term $C(t)$ from Equation 3.20 can be omitted since it remains the same for all branches in time = t .

$$\Gamma_{S_{S_t}} \approx \frac{1}{2} u_i L(u_i) + \frac{L_c}{2} \sum_{l=1}^n y_{lt} x_{lt} \quad (\text{Equation 3.25 repeated})$$

Branch metric calculation using the Log-MAP algorithm involve sign comparisons, binary divisions and also constant scalings of variables. Efficient hardware can be implemented for these operations as described later in Section 6.7.3. A sample calculation of Γ_{SSt} are shown in Figure 4.2.

FIGURE 4.2. Numerical calculation of Γ_{SSt} using Log-MAP algorithm

4.2 Sample A_{St} Calculations

Calculation of the forward path metrics A_{St} is performed iteratively along the trellis using Equation 3.23.

$$A_{St} = \max_{S' \rightarrow S} (A_{S't-1} + \Gamma_{SSt}) + f_c \quad (\text{Equation 3.23 repeated})$$

There are add-compare-select (ACS) operations involved which usually form the critical path of the Log-MAP decoder hardware design. To obtain the correction factor f_c , a look-up-table can be easily implemented. As indicated in Figure 4.3, when the difference between two path metrics is large, the resulting new path metrics is just the sum of the larger path metric and the branch metric, the value of the correction factor f_c is negligible.

FIGURE 4.3. Numerical calculation of A_{S_t} using Log-MAP algorithm

FIGURE 4.4. Iterative calculations of A_{S_t} along trellis starting at state A at time t = 0

Figure 4.4 shows how all path metrics are generated iteratively. The initial state S_t at time $t = 0$ is state A by convention so the path metric originating from state A is assigned a very large value while all the other path metrics are initialized to zero. The correct path representing the

encoder input u_t is highlighted, and as indicated, the path metrics related to the most probable path have the largest path metrics at each time step.

4.3 Sample B_{S_t} Calculations

Calculation of the backward path metrics B_{S_t} is performed iteratively from right to left along the trellis using Equation 3.24 .

$$B_{S_t} = \max_{S \rightarrow S'} (B_{S_{t+1}} + \Gamma_{SS_{t+1}}) + f_c \quad (\text{Equation 3.24 repeated})$$

The operation and hardware required are the same as for calculating the forward path metrics except that it traverses the trellis from the end of the data stream to the beginning, and therefore the previous state connections for each particular state is changed. For example, as shown in Figure 4.5, when calculating A_{S_t} , the previous state of state B is state C for a minus one transition and state D for a positive one transition. When travelling backward for B_{S_t} calculations, the previous state of B becomes state A for a minus one transition and state C for a positive one transition. This is an important reminder when the decoder is built in either hardware or software. In Figure 4.5 the correction factor f_c become negligible when the difference between two merging paths is large.

FIGURE 4.5. Numerical Calculation of B_{S_t} using Log-MAP algorithm

FIGURE 4.6. Iterative calculations of B_{S_i} along trellis from the end of the data sequence to the start, using state A as the initial state at the at time $t = 32$

Figure 4.6 shows how all reverse path metrics are generated iteratively. To enhance the decoding performance, a technique called trellis termination can be used to put a proper bias for B_{A_i} . More details will be given in Section 4.4. The correct path, in which all encoder input are +1, is highlighted, and as indicated in Figure 4.6, the path metrics related to the most probable path has the largest path metrics in each time step.

4.4 Trellis Termination

A trellis termination sequence is used for resetting the last trellis state back to a known state (state A) by transmitting appropriate tail bits after all systematic and parity bits are transmitted. After the actual data transmission, $(N-1)$ zeros are shifted into the first flip-flop of the decoder where N represents the number of bits in the encoder shift register. Since the final state when finishing the transmission is known to the receiver side, the decoder can initialize a high path metric bias to the final state for back tracing. The reverse path metric originated from state A after time $t = 31$ is assigned a very large value ($B_{A_i} = 62$) as indicated in Figure 4.6 while all the other path metrics are initialized to be zero.

4.5 Sample LLR and A-Posteriori Data Calculations

In Figure 4.7, typical calculations of LLR and a-posteriori output, using Equation 3.21 and Equation 2.8, are shown. The process requires multiple ACS circuits for comparisons of all path metrics which can incur a long delay. Section 6.7.5 describes a pipelining architecture that can reduce the clock period, while increasing the data throughput in a hardware implementation.

$$L(u_t|Y) \approx \max_{\substack{u_t = 1 \\ S' \rightarrow S}} (A_{S't-1} + \Gamma_{S'S_t} + B_{S_t}) - \max_{\substack{u_t = -1 \\ S' \rightarrow S}} (A_{S't-1} + \Gamma_{S'S_t} + B_{S_t}) \quad (\text{Equation 3.21 repeated})$$

$$L_e(u_t) = L(u_t|Y) - L(u_t) - (L_c \cdot y_{st}) \quad (\text{Equation 2.8 repeated})$$

FIGURE 4.7. Numerical calculation of LLR $L(u_t|y)$ and a-posteriori output $L_e(u_t)$ using Log-MAP algorithm

FIGURE 4.8. Calculations of LLR and a-posteriori data along trellis

Computations of LLR and a-posteriori data along the trellis are shown in Figure 4.8. Since the calculations of the reverse path metrics B_{S_t} are not required for $t = -1$ in order to calculate the LLR and a-posteriori data, these steps can be omitted. Similarly the path metrics A_{S_t} are not required for $t = 32$. The highlighted trellis path indicated the correct data trace. When the resulting LLR is negative or positive, the decoder output represents a -1(0) or 1 respectively. The original data is recovered correctly in this example given that the E_b / N_0 is 1.5dB.

4.6 Interleaver

A stand alone trellis decoder does not perform well under noise bursts which result in consecutive symbol distortion from the channel. With the use of an interleaver, burst errors can be leveled out and performance can be greatly increased. As indicated in Figure 1.7, the interleaver permutes the position of the second parity bits. If a burst error corrupted consecutive symbols, the permutation of the second parity bits will avoid the burst error affecting its related systematic bit. Because of the encoder interleaver, there is an interleaver and a deinterleaver in the decoder as shown in Figure 1.8. The decoder interleaver permutes the systematic bits from the channel memory and the a-posteriori data from the first decoder for the use of the second decoder. The decoder deinter-

leaver reverses the permutation such that the a-posteriori data can be converted to a-priori which follows the order of the original data.

The interleaving function used in the example of this chapter is shown in Table 4.3. The original position of the data in the frame is indicated by i . The new position of the data after interleaving is indicated by $\pi(i)$.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi(i)$	10	23	8	19	7	21	29	20	12	4	16	24	3	28	14	5
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi(i)$	18	30	6	27	11	1	26	15	31	22	9	17	25	13	0	2

TABLE 4.3. Random interleaving table for 32 bit frame

The deinterleaving function is shown in Table 4.4. i indicates the position of the data in the interleaved frame. $\pi^{-1}(i)$ indicates the position of the data after deinterleaving which is identical to the original position of data before interleaving. For example, the data in the 1st position will be moved to the 23th position after interleaving and the same data will be moved back to the 1st position after deinterleaving.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi^{-1}(i)$	30	21	31	12	9	15	18	4	2	26	0	20	8	29	14	23
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$\pi^{-1}(i)$	10	27	16	3	7	5	25	1	11	28	22	19	13	6	17	24

TABLE 4.4. Deinterleaving table for random interleaving of a 32 bit frame

4.7 Iterations

By interleaving the a-posteriori information generated by the one decoder, a-priori information is generated and feed into the other decoder which in turns generates an improved version of the a-posteriori information. This message passing decoding method used in Turbo decoder is known as iteration. After a sufficient number of iterations, the LLR output of the Turbo decoder can recover the original data with a low error rate. Table 4.5 demonstrated the changes in LLR when the number of iterations increase. An increase in the absolute value of the LLR along with more iterations indicated the confidence level of the code recovery increase. A small LLR indicates low

certainty in the decoded result. As one can see, if there were no iteration, the decoder output will have error at $t = 4$ and $t = 7$.

Time (t)	0	1	2	3	4	5	6	7	8	9	10
Least Square Error	2.3	1.2	0.8	0.8	0.8	1.5	1.5	1.1	1.3	1.9	1.7
Input Data	1.0	-1.0	1.0	-1.0	-1.0	1.0	-1.0	1.0	-1.0	-1.0	1.0
0.5 Iteration	10.8	-8.7	8.9	-6.1	1.6	6.7	-4.9	-1.6	-4.9	-4.9	8.3
1.5 Iterations	19.4	-26.5	20.5	-19.4	-25.5	15.1	-29.2	30.5	-15.1	-27.8	17.1
2.5 Iterations	39.1	-38.3	47.2	-39.1	-38.5	40.0	-47.3	42.1	-38.6	-51.1	36.2
3.5 Iteration	53.6	-50.6	57.4	-53.6	-51.1	50.5	-50.5	58.3	-50.5	-52.1	51.5
4 Iterations											

Time (t)	11	12	13	14	15	16	17	18	19	20	21
Least Square Error	1.4	0.6	0.9	1.6	1.7	1.8	1.6	0.9	1.4	1.7	1.7
Input Data	1.0	-1.0	1.0	-1.0	1.0	1.0	-1.0	1.0	-1.0	1.0	1.0
0.5 Iteration	11.9	-4.2	1.1	-1.1	1.1	4.7	-6.2	6.1	-5.8	5.7	5.9
1.5 Iterations	19.8	-25.1	17.1	-28.8	26.3	25.2	-19.5	27.9	-24.7	20.0	25.0
2.5 Iterations	38.6	-37.5	38.7	-46.1	39.9	36.4	-39.1	43.4	-39.3	39.1	35.4
3.5 Iteration	53.8	-37.6	59.6	-60.3	42.5	42.9	-58.9	49.4	-42.9	60.0	35.8
4 Iterations											

Time (t)	22	23	24	25	26	27	28	29	30	31
Least Square Error	0.6	1.3	1.5	0.7	1.4	1.8	1.6	1.5	1.2	1.1
Input Data	-1.0	1.0	-1.0	1.0	-1.0	-1.0	1.0	1.0	1.0	1.0
0.5 Iteration	-8.7	12.6	-7.2	8.2	-6.1	-7.2	7.2	5.8	9.6	9.4
1.5 Iterations	-25.9	20.1	-24.2	25.9	-16.5	-21.7	27.6	16.5	21.7	31.9
2.5 Iterations	-52.3	43.7	-33.1	49.8	-41.1	-40.7	41.2	41.1	33.1	41.2
3.5 Iteration	-56.6	57.2	-35.7	51.2	-54.1	-41.8	49.6	54.1	44.4	49.4
4 Iterations										

TABLE 4.5. LLR table for the example Turbo decoder

Chapter 5: Simulated Decoding Performance of the Turbo Decoder

There are many factors affecting the decoding performance of the Turbo decoder such as the number of iterations, the choice of puncturing, code rate, data frame length, encoder polynomial and interleaver details [WH2000]. In order to verify the performance of the resulting hardware, a software model was constructed in Matlab and is included in Appendix B. The software model also served the purpose of pre-design optimization due to its flexibility. Design parameters such as the level of quantization and the range of data variables can be determined for the hardware. In this chapter, the simulated performance of the Turbo decoder in different configurations is discussed. The default configurations of all the simulations are outlined in Table 5.1.

Channel	AWGN
Modulation	BPSK
Channel reliability assumption	$L_c = 1$ for high noise (0 dB)
Component encoders	2 identical RSC
Component encoder code Rate R	1/3 non-puncture
RSC encoder polynomial	G0 = 13, G1 = 15
Decoder algorithm	Log-MAP
Interleaver type	Random interleaver, L = frame size
Frame size L	4096 bits
Number of iterations I	8
Number of Frame	500
Method of termination	Both trellis terminated

TABLE 5.1. Default configuration for Turbo decoder simulator

5.1 Number of Iterations

Turbo decoder uses a message passing technique to perform decoding iteration. A-posteriori data generated from one decoder is passed to the other decoder as a-priori data. Generally the larger the number of iterations, the more decoding performance can be achieved. However, more iterations also create more latency for the decoding output. From Figure 5.1 it is shown that performance beyond 8 iterations does not improve significantly. The ratio of energy per message bit

to noise power E_b/N_o between 8 iterations and 12 iterations is less than 0.1 dB at a desire BER of 0.01%. Hence using more than 8 iterations does not increase the performance enough to justify the increased latency.

FIGURE 5.1. Performance of Turbo decoder for different number of iterations at $L=1024$ bits

In Figure 5.1, the long number of iterations (8 to 12) required excessive simulation time. Hence the number of frames was reduced from 500 to 300. This is thought to be the reason for the knee in the graph.

5.2 Channel Reliability Factor

As mentioned near Equation 2.7, the channel reliability L_c is determined by the type and level of noise in the channel. Since the Log-MAP algorithm is mainly used in noisy environments, and an extremely low bit error rate is not required for a low noise environment if an outer decoder is used as shown in Figure 1.1, the value of L_c can be set equal to 1 which means it can be eliminated in Equation 3.20 for easy hardware implementation. In high noise areas ($E_b/N_o < 0.5$ dB), approximating L_c equal to 1 has mildly improved the performance of the decoder since it reduces

the importance of received symbols and increases the importance of a-priori data. The effect of setting L_c equal to 1 is shown in Figure 5.2.

One could easily change L_c in 3dB steps by making $L_c = 1, 2, 4$, etc. This would still avoid implementing a hardware multiplier.

FIGURE 5.2. Performance of Turbo decoder for different L_c setting

5.3 Generator Polynomial

The generator polynomial, i.e. the location of taps and the size of the encoder determines how efficiently the parity of the data can be generated. A good generator polynomial, which is called a primitive polynomial, maps input data to a higher dimension space and creates “dis likeness” between coded data. Inappropriate generator polynomials which are not primitive can also lead to a so-called catastrophic code¹ [Vit1971] in decoding. Figure 5.3 and Figure 5.4 show the performance variations between different generator polynomials in different frame sizes. The polynomial G0 13, G1 15 is a primitive polynomial versus G0 15, G1 17 which is not primitive

1. Unlimited errors occur because a small error happened in one stage and lead to a completely different path with different decoder output in the trellis.

[Hou2001]. It can be seen that for the polynomial G0 13, G1 15 performs slightly better than G0 15, G1 17 in environments with lower noise. The effect of having a primitive polynomial is more noticeable in code with a shorter constraint length as seen in Figure 5.4 [WB2000]. This is because when there is a smaller set of code words available for small constraint length, the uniqueness of code words becomes very crucial. In Figure 5.4, G0 7, G1 5 is a primitive polynomial while the rest of the polynomials are not.

FIGURE 5.3. Performance of Turbo decoder for different generator polynomials

FIGURE 5.4. Performance of Turbo decoder for different generator polynomial with different constraint length [WB2000]

5.4 Code Puncturing

Puncturing the transmitted data, i.e. not transmitting all of the parity bit from each decoder will increase the code rate of transmitter. Increasing the code rate will effective reduce the redundancy of the transmit data and degrade the decode performance as shown in Figure 5.5. Puncturing the transmit data to change the code rate is easier to implement than changing the generator polynomial of the encoder since the same decoder structure can be used in the former case. Puncturing the transmitted data is usually done to preserve bandwidth when the noise level is low.

FIGURE 5.5. Performance of a Turbo decoder under code puncturing

5.5 Interleaver Function

Interleaving the data can reduce the data distortion from burst noise since the parity bits from the same data correlate with different bits near by and will be transmitted in a different time slot.

Interleaving the data can also help reduce the BER at high $\frac{E_b}{N_o}$ environment [BMD2003]. Having

an interleaver is crucial for message passing decoding as shown in Figure 5.6. This is because two different perspectives of information from the same source will be combined by the two log-MAP decoders to generate a-priori data for each other. In general, a larger interleaver allows nearby data to move to a further distance away from each other and hence they are less affected by error bursts. There are a lot of proposed interleavers that will benefit the BER performance in a low noise environment. In [WH2000] performance of these interleavers are reviewed. For this research, a random decoder was built.

FIGURE 5.6. Effect of interleaver in performance of Turbo decoder

5.6 Frame Size

A larger frame size allows more accumulation of path differences, and hence the decode performance will increase as shown in Figure 5.7. In the original Turbo decoding publication by Berrou *et al.* [BGT1993] and several other publications, outstanding decoding performance that approaches Shannon's limit was achieved by using a very large frame size. However, larger frame sizes will introduce more latency in obtaining the decoder's output. Therefore depending on the type of applications, different frame sizes will be used. Also in [DDP1998], it is shown that the performance of Turbo decoder can still approach Shannon's limit even for frames of a few hundred bits. This lead to the possibility of performing windowing in a hardware implementation as described later in Section 6.7.1.

FIGURE 5.7. Performance of Turbo decoder for different frame sizes

5.7 Decoding Algorithm

The decoding algorithm is a major factor that determines the decoding performance of the Turbo decoder. Viterbi and Max Log-MAP algorithms do not perform as well as MAP and Log-MAP algorithms because they discard the path and path metrics of the losing path as mentioned in the earlier chapters. In the Max Log-MAP decoder, this discard of information causes a loss in decoding performance which can be compensated by the correction factor in the Log-MAP algorithm. This causes the Log-MAP decoder to perform approximately 0.3dB better for a BER of 0.01% as shown in Figure 5.8.

FIGURE 5.8. Performance of Turbo decoder for different decoder algorithms

5.8 Trellis Termination

Terminating the trellis, as discussed in Section 4.4, provides a good starting advantage for the B_{St} calculation. This gives a small improvement to the decoding performance of about 0.2 dB at a BER of 0.01% as shown in Figure 5.9. Terminating the trellis involves sending a termination sequence which will reduce the user data transmission rate. For a very large frame size, this side effect is negligible.

FIGURE 5.9. Performance of Turbo decoder for different choices of termination

Chapter 6: Hardware Design and Implementation

The Turbo Decoder Emulator (TDE) is implemented on a Xilinx MicroBlaze and Multimedia demonstration board. Verilog HDL¹ was used for programming the Xilinx FPGA². The Verilog HDL coded TDE is included in Appendix A.

6.1 Design Platform Description

A picture of the Xilinx MicroBlaze and Multimedia demonstration board is shown in Figure 6.1. Although its functions are not fully utilized for the purpose of Turbo decoder simulation, this platform was chosen because of the number of ZBT³ memories (5) available to be used by the large capacity Xilinx Virtex II XC2V2000 FPGA. The FPGA contains on-chip memory which is beneficial to the hardware simulation of the Turbo decoder. An FPGA program file can either be downloaded by JTAG⁴, Ethernet or be stored and retrieved from a CompactFlash card through the CompactFlash adapter. In order to run the simulation environment, a JTAG cable must be connected to the board for programming, and for the Xilinx Chipscope Pro⁵ program to access the FPGA from a PC.

FIGURE 6.1. Xilinx MicroBlaze and Multimedia demonstration board

1. Verilog Hardware Descriptive Language - a very common hardware programming language, particularly in North America. The other very common one is called VHDL. VHDL is more popular in Europe.
2. Field Programmable Gate Array - a reprogrammable, volatile, device which can be programmed using Verilog HDL to execute different functions.
3. Zero-bus turnaround SRAM with no bus latency memory is a synchronous-burst SRAM with a simplified interface that fully uses available bandwidth. ZBT SRAM devices use the full bandwidth because they do not require turnaround cycles which is the idle cycles between read and write operations.
4. Stands for Joint Test Action Group. It is an standard named "IEEE Standard 1149.1" that specify how to control and monitor the pins of compliant devices on a printed circuit board.
5. Chipscope Pro is an FPGA on chip verification tools from Xilinx.

6.2 Variable Quantization

As the first step before implementing the hardware, the software model described in Chapter 5 was used to determine the precision and width of data variables. A hardware optimization technique called fixed windowing (see Section 6.7.1) was used such that path metrics A_{S_t} and B_{S_t} are set to smaller pre-initialized values periodically, or when the metrics overflow. The final bit-width of the variables are listed in Table 6.1.

Variable Name	Width of Data Bit	Low E_b/N_o			High E_b/N_o		
		Step Size (Precision)	Lower Limit	Higher Limit	Step Size (Precision)	Lower Limit	Higher Limit
Systematic	4	0.25	-2	1.75	0.25	-2	1.75
Parity	4	0.25	-2	1.75	0.25	-2	1.75
$\Gamma_{S'S_t}$	8	0.25	-32	31.75	0.5	-64	63.5
A_{S_t}	16	0.25	-8192	8191.75	0.5	-16384	16383.5
B_{S_t}	16	0.25	-8192	8191.75	0.5	-16384	16383.5
A-Priori data	8	0.25	-32	31.75	0.5	-64	63.5
A-posteriori data	8	0.25	-32	31.75	0.5	-64	63.5
LLR	16	0.25	-8192	8191.75	0.5	-16384	16383.5

TABLE 6.1. Data variable quantization values in order to reduce the width of the data representation

6.3 Turbo Decoder Emulator (TDE) Design

The top level design of the TDE (Figure 6.2) is customized to fit the architecture of the Xilinx MicroBlaze and Multimedia demonstration board. The default parameters of the TDE are listed in Table 6.2.

Channel	Random noise
Modulation	BPSK
Quantization	Lc = 1, Refer to Table 6.1 for other values
Component encoders	2 identical RSC
Component encoder code rate	1/3 non-puncture
RSC encoder polynomial	G0 = 7, G1 = 4, K = 3
Decoder algorithm	Log-MAP
Interleaver type	Random permutation, interleaver size $L_{\pi} = L$
Frame size L	1 Mbits
Number of iterations	8
Method of termination	Both trellis terminated
Number of frames	1000
Windowing method	Fixed windowing
Window size	1023

TABLE 6.2. Default configuration for TDE

The TDE integrated the ZBT memory interface and clock generator supplied by Xilinx into its design. The ZBT Memory Interface contains five individual interfaces for the five ZBT memory chips. Each separate interface accommodates special timing requirements of the specific ZBT for data access. The clock generator instantiates DLL¹ elements to phase lock the 27MHz system clock, create internal clocks and deskew the ZBT memories input clocks.

1. Delay lock loop

FIGURE 6.2. TDE block diagram

The Chipscope Pro interface is accessed by an external PC to debug and control the setting of the TDE. The interface consists of three basic components which are the ICON¹, ILA² and VIO³ [Xil2004] for interfacing between the software and hardware.

The purpose of the data generator is to create soft data input that is already interleaved that can be accessed in order by the decoder. This data is stored in memory for the decoder to retrieve. More details are given in Section 6.5.

1. Integrated controller - Communicates with software via JTAG and controls other block within the Chipscope interface block
2. Integrated logic analyzer - Logic Analyzer built by internal FPGA logic and memories. Data samples are stored and shifted out from the internal memories.
3. Virtual input / output - Interface between software and hardware to support virtual functions such as pushing buttons and lighting LEDs in the software interface.

The Turbo decoder is the main component of the TDE. It uses the Log-MAP algorithm and is described in Section 6.6

The comparator compares the output from the Turbo decoder with the input of the component encoder (shown in Figure 6.10). It notifies the data collector when an error occurs. Only statistical data is stored in the data collector such as the number of bit errors and the number of frame errors.

The top hierarchy also contains of two address generators used by several low level blocks. Two generators are used so that two addresses can be generated at the same time, for example, one for reading and one for writing data simultaneously. More details on the internal structure of the address generator are given in Section 6.4.

6.3.1 TDE Controller Design

The TDE controller controls access to the memory interface, creates enable signals, and keeps track of the total number of frames left for decoding. To control access to the memory interfaces, there is an internal utility counter within the controller creating delays during single and multiple memory accesses. This is necessary because the memory access is pipelined and it takes four cycles for read data to become valid. An address has to be fetched before the data become valid, and the last set of data read from the memory need to be flushed out from the pipeline. The utility counter gives a reference points to time all these events and appropriate control signals can be generated according to each state with memory access. The user can also control the number of frames required for simulation to achieve the desirable precision of the statistical result. The number of frames generated is controlled by a configurable frame counter. The state diagram of the TDE controller is shown in Figure 6.3 to describe the processes of the Turbo decoder simulation.

FIGURE 6.3. State diagram of TDE

6.3.2 User Interface

As seen in Figure 6.4, Chipscope Pro gives the user the ability to view the waveforms of a pre-assigned set of internal signals. It provides basic functions of a logic analyzer to assist in debugging the user design. It also provide Virtual Input/Output (VIO) when the hardware does not have enough physical external inputs or outputs. From the VIO interface, the users can perform run time selections of the total frame number, termination requirement, number of iterations and a choice of two noise levels. The selection of different E_b/N_o values can be implemented as a future extension.

FIGURE 6.4. User interface of the TDE through the Chipscope Pro program

6.4 Address Generator (Interleaver) Design

The address generator consists of two major sections. One is the random counter to generate a random addresses for the use of interleaving and deinterleaving. Another one is the up/down counter that creates ascending or descending address sequences for traversing the memory to access data. A block diagram of the address generator is shown in Figure 6.5.

To perform interleaving, the destination address is generated by the LFSR¹. Data is then written to this location. Due to the width (32bits) and the number (5) of the available memory module, the extra memory available can be used to simplify the hardware in the interleaver. The original address of the data is also written to the same permuted address location for use in deinterleaving.

1. Linear Feedback Shift Register

To perform deinterleaving, when the interleaved data is read, the original location of this data can also be retrieved in the same time and become the deinterleaving destination address.

FIGURE 6.5. Block diagram of the address generator

The hardware supporting the interleaving function of the simulation system is the up/down counter and the random counter.

The up/down counter is used for reading and writing data from both ascending and descending sequential address locations.

Each random counter contains a LFSR which is used for generating the random address for interleaving. The LFSR also supplies data to be written into the memory for deinterleaving purposes.

A normal LFSR has two limitations preventing it to be used in an interleaver. The first one is that a normal LFSR always has one state less than a binary counter due to the absence of an all zero state. This prevents random permutation on all the address locations. The second is that a normal LFSR does not support the use of termination sequence. Termination sequences are required to be kept in the ending portion of the data stream.

To modify a normal LFSR to include state zero for interleaving, an override circuit is inserted as shown in Figure 6.6 to deal with the single exception case when all bits in the LFSR are zeros. The all zeros state is inserted into the chain of state transitions. The output address hence will be unique and this set up can be used as an interleaver.

FIGURE 6.6. Zero state insertion circuit in LFSR

Another override circuit is also implemented into the interleaver design to prevent interleaving of the trellis termination sequence as shown in Figure 6.7.

FIGURE 6.7. Block diagram of the random address generator

These special addresses are shown in Table 6.3 for several different random interleaver sizes. These address are specific to the structure of the LFSR. When these addresses are detected, an override address will be used for the permutation instead. The override addresses must be one of the other special address in order to maintain the uniqueness and randomness of the interleaver address output.

Interleaver Size 2^n	From Address	Random Address	Substitute Random Address
n = 10	0	3ff	2bd
	2bd	3fe	0
n = 11	0	7ff	229
	229	7fe	0
n = 12	0	fff	946
	946	ffe	0
n = 13	0	1fff	b3a
	b3a	1ffe	0
n = 14	0	3fff	69d
	69d	3ffe	0
n = 15	0	7fff	1
	1	7ffe	0
n = 16	0	ffff	5248
	5248	fffe	0
n = 19	0	7fff	19a2a
	19a2a	7ffe	0

TABLE 6.3. Address substitutions for trellis termination

6.5 Data Generator Design

The function of the data generator is to create the input for the Turbo decoder. These decoder inputs are random, valid and have had noise added. The block diagram of the data generator is shown in Figure 6.8. The LFSR random input creator creates random sequence of 1s and 0s. This sequence is then written into the memory in both normal and permuted order. The component encoder then reads these inputs to create valid parity and termination bits for both original and interleaved sequences. Finally these RSC outputs are changed to that the value of (1,0) becomes (-1,+1) respectively and added with noise in the channel data mapper. The resulting soft values are written into the memory and stay the same throughout the decoding process until the creation of a new data frame.

FIGURE 6.8. Block diagram of the data generator

6.5.1 Data Generator Controller Design

The data generator controller interfaces all local control signals as well as the control signals to the memory interface. The decoding process followed by the controller is described in its state diagram as shown in Figure 6.9. The data generator also has an internal utility counter for the same reason explained in Section 6.3.1.

FIGURE 6.9. State diagram of the data generator controller

6.5.2 LFSR Random Input Generator

The random input generator outputs random 1s and 0s to be used by the component encoder as inputs. Since there is no requirement on the randomness of the sequence to improve the simulation, a small LFSR with function $x^2 + 1$ is used.

6.5.3 Component Encoder

The component encoder contains two RSC blocks. Two input data sequence, one normal and one interleaved are read from the memory and shifted into the RSC. The generated parity bits, together with the normal encoder input, which are the systematic bits, are written back to memory. At the end of the transmission sequence, the two switches indicated in Figure 6.10 are closed to make the output of the XOR gates zeros for the generation of the termination bits. The termination bits are padded as tail bits to the output sequence for trellis termination as described in Section 4.4.

FIGURE 6.10. Block diagram of the component encoder

6.5.4 Channel Data Mapper

The channel data mapper inputs uses the systematic and two parity sequences from the memory to create soft input for the Turbo decoder. The process of the mapping included modulating and adding noise to data as shown in Figure 6.11. The value 1000 is not used because the 2's complement of it would overflow. The value 0000 is neutral and can be used to substitute data that is missing as the result of puncturing. In the current implementation of the channel data mapper, only uniform noise is added to the data. A realistic hardware implementation of the channel model should be given by the user of TDE to a perform emulation with measurable decoding performance.

FIGURE 6.11. Data mapping diagram

6.5.5 Address Generator Utilization

The address generators are located in the top level of the design hierarchy and are shared between multiple blocks. The two address generators A and B allows concurrent memory access to more than one address location on different memory modules so that data can be processed without interruption. The usage of address generators for different purpose in the data generator are outlined in Table 6.4.

Process	Read	Write
Create Input and Interleaved Input		Bank 0,1 : Input/Interleaved Input
Encode Input Data	Bank 0,1 : Input/Interleaved Input	Bank 2 : Encoder Output
Map Encoder Hard Outputs to Decoder Soft Inputs	Bank 2 : Encoder Hard Output	Bank 0,1 : Decoder Soft Inputs
Created Interleaved Soft Input	Bank 0,1 : Decoder Soft Inputs	Bank 2 : Interleave Decoder Soft Input

A Address

B Address

TABLE 6.4. Address generator utilization in data generator

6.5.6 Data Generator Memory Map

The data outputs from the generator are created and stored in memory in a format that helps increase data throughput and reduce the decoding latency of the Turbo decoder as shown in Table 6.5. Five memory modules are fully utilized to allow immediate access of the data variable. Since the same systematic bits are used for both Log-MAP decoders, the soft systematic outputs from the channel data mapper are interleaved and written to the memory in the Interleave state. This causes the need for two locations to store the soft values of parity 2, because it needs to be concatenated with the systematic bits and be written back to another memory location.

The soft systematic, soft parities and encoder inputs will be preserved in memory for the use of the Turbo decoder and result comparator in later processes.

Function	Encoder Hard Input and temporary storage for Interleaved Soft Parity 2																															
Address	(Chip) Bank 0: 0FFFF-7FFF																															
Byte	D								C				B								A											
Bit	7	6	5	4	3	2	1	0	7:1				0				7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Reserved				Soft Par2				Reserved				Encoder Input				Reserved															

Function	Encoder Interleaved Hard Input												Decoder 1 Soft Systematic and Parity Encoder Interleaved Input																			
Address	(Chip) Bank 1: 0FFFF-7FFF																															
Byte	D								C								B				A											
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7:0				7	6	5	4	3	2	1	0 (Share)				
Meaning	Reserved								Interleaved Input								Reserved				Soft Systematic				Soft Parity1				Encoder Input			

Function	Encoder Hard Output																Decoder 2 Interleaved Soft Systematic and Parity															
Address	(Chip) Bank 2: 0FFFF-7FFF																															
Byte	D				C								B								A											
Bit	7:4		3	2	1	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0			
Meaning	Rsv	Sys	Par1	(Int) Par2	Reserved																Soft Systematic				Soft Parity2							

Function	Not Use																															
Address	(Chip) Bank 3: 0FFFF-7FFF																															
Word	D								C								B								A							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Reserved																															

Function	Not Use																															
Address	(Chip) Bank 4: 0FFFF-7FFF																															
Word	D								C								B								A							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Reserved																															

TABLE 6.5. Data generator memory map

6.6 Turbo Decoder Design

The top level Turbo decoder has two Log-MAP decoders and a controller as shown in Figure 6.12. This Turbo decoder implements the MAP decoding theory as described in Chapter 2. It has several hardware optimizations as described in later sections to reduce hardware complexity while not sacrificing too much of the decode performance. The controller initializes and terminates the decoding cycle of the Log-MAP decoder while also triggering signals for memory access. The Log-MAP decoder follows the decode procedure as demonstrated in Chapter 4 to calculate a-posteriori data for the use of the other Log-MAP decoder. Memory is accessed to obtain the a-priori input, systematic and parity inputs, while B_{sk} is written to memory as an intermediate value before calculating the a-posteriori data. When calculations are finished, the a-posteriori data are interleaved and written into memory, and become a-priori input for the other Log-MAP decoder. The a-priori values are set to zero for decoder 1 during the first iteration.

FIGURE 6.12. Block diagram of Turbo decoder

6.6.1 Turbo Decoder Controller Design

The Turbo decoder controller uses the internal utility counter to control memory access of the Turbo decoder and to activate one of the Log-MAP decoders at the appropriate time by monitoring the address counters. It also contains a loop counter for tracking the number of iterations dur-

ing the decoding process. During the last iteration, it will switch the mode of the second Log-MAP decoder to output LLR instead of a-posteriori data. The LLR are written into memory and are retrieved later for comparing with the encoder input. The complete decode process is described in state diagram shown in Figure 6.13.

FIGURE 6.13. State diagram of the Turbo decoder

6.6.2 Address Generator Utilization

The Turbo decoder shares the same two address generators in the top level with the other blocks. It uses the two address counter for read / write access alternatively in each process. The usage of the address generator in the decoding process is described in Table 6.6. Note that when decoder 1 writes the a-posteriori signal to the memory, one address generator is used for reading the input, while the other reads the deinterleaving address after a delay. The deinterleaving address is read in only when the output a-posteriori data are ready to be rewritten back to the deinterleaving address

location in the memory, and be used as the a-priori signals from the second decoder to the first decoder,

Process	Read	Write
Decoder one reverse calculation for Beta	Bank 1: A-priori, Soft Sys and Parity	Bank 3,4 : Beta
Decoder one calculation for Alpha and A posteriori	Bank 3,4: Beta	Bank 2: A-posteriori
	Bank 1: A-priori, Soft Sys and Parity	Bank 0,2: De-Interleave Address
Decoder two reverse calculation for Beta	Bank 2: A-priori, Soft Sys and Parity	Bank 3,4: Beta
Decoder two calculation for Alpha and A posteriori	Bank 3,4 : Beta	Bank 1: A-Posteriori
	Bank 2: De-Interleave Address[18:16], A-priori, Soft Sys and Parity	
	Bank 0: De-Interleave Address[15:0]	
Decoder two calculation for Alpha and A posteriori	Bank 3,4 : Beta	Bank 1: LLR
	Bank 2: De-Interleave Address[18:16], A-priori, Soft Sys and Parity	
	Bank 0: De-Interleave Address[15:0]	

	A Address
	B Address
	Interleaved Address

TABLE 6.6. Address generator utilization in Turbo decoder

6.6.3 Data Generator Memory Map

The memory map of the Turbo decoder is arranged in a format that help increase the data throughput and reduce the decoding latency as shown in Table 6.7. Five memory modules are fully utilized to allow immediate access to the data variables. Note that memory banks 4 and 5 are shared by both decoders to store immediate values of B_{sk} before a-posteriori data or LLR calculation. These values are not interleaved since they are used locally by the decoder. Also when the data width of the deinterleaving address exceeds 16 bits, two different memory banks (0,2) are used to store the extended deinterleave address. Because memory bank 2 which contains the top 3 bits of the deinterleave address is read a few cycles before the rest of the address in memory bank 0, a small pipeline is used to align the address while deinterleaving the a-posteriori data and LLR data.

Function	Encoder Hard Input						Original Address[15:0] (Used by Interleaver)															
Address	(Chip) Bank 0: 0FFFF-7FFF																					
Byte	D		C				B								A							
Bit	7:0		7:1		0		7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Reserved				Input (Sys)		From_Addr [15:0]															

Function	Turbo Decoder Output												Decoder 1 A-Priori, Soft Systematic and Parity																			
Address	(Chip) Bank 1: 0FFFF-7FFF																															
Byte	D								C								B								A							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	LLR[15:0]												A-Priori								Soft Systematic				Soft Parity							

Function	Extended Original Address[18:16] (Used by Interleaver)						Decoder 2 A-Priori, Soft Systematic and Parity																
Address	(Chip) Bank 2: 0FFFF-7FFF																						
Byte	D		C				B								A								
Bit	7:0		7:3		2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Reserved				From_Addr [18:16]			A-Priori								Soft Systematic			Soft Parity				

Function	Beta Value Positive Input																															
Address	(Chip) Bank 3: 0FFFF-7FFF																															
Word	D								C								B								A							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Beta_a																Beta_c															

Function	Beta Value Negative Input																															
Address	(Chip) Bank 4: 0FFFF-7FFF																															
Word	D								C								B								A							
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Meaning	Beta_b																Beta_d															

TABLE 6.7. Turbo decoder memory map

6.7 Log-MAP Decoder Design

Log-MAP decoder consists of blocks that calculate branch metrics (Γ_{S_t}), path metrics (A_{S_t} , B_{S_t}), a-posteriori data and LLR data as shown in Figure 6.14. A controller was implemented to receive control inputs, initialize all local control signals and to notify the blocks in the upper hierarchy when outputs are valid. There are four identical path metrics groups in the Log-MAP decoder. In normal operation, the path metric groups feed their output values back to accumulate path metrics. To implement the fixed windowing technique as describe in Section 6.7.1, the controller contains an LFSR to trigger the refreshing of the path metrics. Path metrics groups, beta calculation and alpha calculation, are refreshed by the outputs of other two path metric groups, beta init and alpha init respectively. All path metric groups share the same branch metric calculation unit. The precision of the branch metric calculation is controlled by the SNR_select.

FIGURE 6.14. Block diagram of the Log-MAP decoder

Since the a-priori input, systematic and parity are read from the memory for branch metric calculations several cycles before a-posteriori/LLR calculations, pipelines were inserted to store the delay inputs in order to synchronize them with the B_{S_t} inputs.

6.7.1 Fixed Windowing Technique

One way to reduce memory requirements on the hardware is to use a technique called windowing. Windowing is common in the hardware implementation of Viterbi decoders. In a Viterbi decoder, it is very likely that multiple paths on a trellis diagram tend to originate from a common state when the path length is at least 5 times the constraint length of the convolutional encoder [Vit1998]. The trace back length is called the window and its size is symbolized as W . Since it is not necessary to back trace the complete message to obtain the output, only variables within the range W need to be stored in the memory. In Equation 2.5,

$$LLR = L(u_t|Y) = \ln\left(\frac{P(u_t = 1|Y)}{P(u_t = -1|Y)}\right) \quad (\text{Equation 2.5 repeated})$$

the output of a low level decoder is dependent on the the whole message Y . The output LLR does not have to depend on all symbols from the received message, and the equation can be rewritten as:

$$LLR = L(u_t|Y_w) = \ln\left(\frac{P(u_t = 1|Y_w)}{P(u_t = -1|Y_w)}\right) \quad (\text{EQ 6.1})$$

where,

$$Y_w \in Y \quad (\text{EQ 6.2})$$

Only a subset Y_w of symbols from Y are used in calculating the output LLR. There are two common windowing methods in Viterbi decoder design. One requires W backtracing steps to be performed in each time step to obtain an output. This windowing method is called sliding windows. It can causes an increase in the number of decoding calculation steps. To reduce the number of decoding calculation steps, another windowing method of Viterbi decoders called fixed windowing can be used. In the fixed windowing method, the path metrics are first initialized with W time steps of received symbols, then each subsequence accumulative calculation can generate valid path metrics. Figure 6.15 demonstrates the difference between sliding window and fixed windowing method.

FIGURE 6.15. Comparison between sliding window and fixed windowing techniques

Sliding windows was proposed in [BD1995] for the SISO decoder in a Turbo decoder system. Unlike normal convolutional codes without feedback, RSC correlates the input data a lot longer than convolutional codes, and the window size W within a turbo decoder system has to be many times larger than the constraint length, but it still can be significantly less than the full frame size. In this hardware implementation, W is equal to 1023, and satisfactory decoding performance was observed in simulation. Research [BDMP1996], [SCE2001], [WP2002], [TGV+2002], [MS2003], [LSS2003] and [TZC2004] was performed to improve the sliding windows method for MAP decoder through either pipelining or parallelism. In the log-MAP decoder implementation of this research, another common Viterbi decoder windowing technique called fixed windowing was used to reduce memory requirement.

In an internal research report [All2002], the fixed windowing technique was applied on reverse path metrics B_{S_t} calculations of a Log-MAP decoder. The resulting hardware consumed low power and had high data throughput. However, the Log-MAP implementation in [All2002] does not consider a-priori data and a-posteriori data, and hence was not usable by a Turbo decoder. The log-MAP decoder hardware was redesigned for the Turbo decoder in this research. Also the Log-MAP decoder in this hardware implementation uses fixed windowing on both forward A_{S_t} and reverse path metrics B_{S_t} calculations to allow path metrics overflow protection.

6.7.2 Log-MAP Decoder Controller Design

The controller operates based on the Mode inputs. When Mode = 01, beta calculation units are enabled. When Mode = 1d, where d represents “don’t care”, all the alpha calculation units are enabled. When the outputs are valid, a data valid signal will be sent to the blocks in the upper hierarchy. There is an LFSR counter that enables refreshing of the path metrics at a fixed period during path metric calculations. The LFSR counter is reset at state changes, and is only enabled when not in the RESET state. The process of decoding in Log-MAP decoder is described in Figure 6.16.

FIGURE 6.16. State diagram of the Log-MAP decoder

6.7.3 Branch Metric Calculation Unit Design

The branch metric calculation unit implements Equation 3.20 in hardware as shown in Figure 6.17. It does that by comparing expected a-priori, systematic, and parity bits to their soft values. When the expected value does not agree with their soft value, the soft value are inverted by 2's complement. Depending on the SNR setting, the soft systematic and parity data may be shifted. Finally the branch metrics are the sum of all these processed soft values divided by two according to Equation 3.25.

$$\Gamma_{SSl} \approx \frac{1}{2}u_l L(u_l) + \frac{L_c}{2} \sum_{l=1}^n y_{lt} x_{lt} \quad (\text{Equation 3.25 repeated})$$

Normally a 2's complement is done by adding 1 to the inverted bits. To add 2's complement values in hardware, the expected values are sent through a controlled inverter (XOR) with each expected value as inversion controls. The number of inversions *num_inv* is then added to the sum of inverted/non-inverted inputs to complete the 2's complement process.

FIGURE 6.17. Block diagram of the branch metrics calculation unit

Figure 6.17 shows branch metrics calculation on a single branch. In this Turbo decoder implementation, there are a total of 8 branches. In order to simplify the higher level implementation, 8 branch metrics calculation units are grouped into a bundle as shown in Figure 6.18. Note that because four branches have the same expected encoder input, systematic and parity, these redundant units can be removed. The synthesis tools perform the truncation automatically. The RTL codes are kept the same for clarification.

FIGURE 6.18. Block diagram of the branch metrics bundle

6.7.4 Path Metric Calculation Unit Design

The path metric calculation unit shown in Figure 6.19 is the hardware implementation of Equation 3.23 or Equation 3.24 for A_{S_t} or B_{S_t} calculation.

$$A_{S_t} = \max_{S'} (A_{S_{t-1}} + \Gamma_{S'S_t}) + f_c \quad (\text{Equation 3.23 repeated})$$

$$B_{S_t} = \max_{S'} (B_{S_{t+1}} + \Gamma_{S'S_{t+1}}) + f_c \quad (\text{Equation 3.24 repeated})$$

When two branches merge into a state, the sum of the upper branches metrics with its previous path metrics becomes the upper path sum. Similarly the lower path sum is calculated for comparison. Generally the highest path sum will become the new path metrics after summing with the correction factor. However the correction factor can be ignored in some cases as shown in Table 3.1 due to precision requirements. In the TDE, the correction factor can be ignored when the absolute difference between path sums are larger than 2 for low SNR setting and 1 for high

SNR setting. This causes three possible data paths to be chosen by the output of the comparator. When the difference of the upper and lower path sum are large, one of the two path sums will become the new path metrics. When the difference of the two path sums is small, a correction factor will be added to the path sum to form the new path metrics.

FIGURE 6.19. Block diagram of the path metrics calculation unit

Parallel processing was used to reduce the total delay caused by the adder and comparator. When the 2x16 bits comparator is comparing the two path sums, the operations for determining the correction factor and adding the correction factor to the path sum are also performed in parallel. This makes sure that if the path sums [15:4] are equal for both the upper and lower paths, and the correction factor is needed, their sum will be ready immediately to be chosen when the comparator output is ready

To determine the correction factor, the least significant four bits of both path sums are required. A look-up-table (LUT) was used to determine the correction factor as shown in Figure 6.19. Since

the correction factor is a small value, the full 2x16 bit adder is not used to add it to the path sum. The least significant four bits of the path sum and correction factor are added together. The carry out of this adder is then use for the incrementing the path sum[15:4]. Finally the concatenation of the adder and decoder output is used to provide possible values of new path metrics.

Path differences	Correction factor		
	Low SNR	High SNR	
1000	01	00	
0111			
0110			
0101			
0100			
0011	10	01	
0010			
0001			
0000	11		
1111	10		00
1110			
1101			
1100	01	00	
1011			
1010			
1001			

TABLE 6.8. Look-up-table (LUT) to determine correction factor

Figure 6.19 shows the path metrics calculation on a single state. In the Turbo decoder implementation, there are a total of 4 states. In order to simplify the higher level implementation, 4 path metrics unit are grouped into a bundle as shown in Figure 6.20.

FIGURE 6.20. Block diagram of the path metrics bundle

6.7.5 A-Posteriori Data and LLR Calculation Unit Design

To calculate the output of the Log-MAP decoder, the a-posteriori data and LLR calculation unit implements Equation 2.8 and Equation 3.21 in hardware. The inputs of this unit include the preliminary path sum of $A_{S_{t-1}} + \Gamma_{S_{t-1}}$ as indicated by two of the outputs in Figure 6.20. The B_{S_t} are also read from memory for each state of the trellis. For a-posteriori data calculation, the systematic and a-priori inputs are obtained from the output of a delay pipeline as mentioned in Section 6.7. Since the process of obtaining the result in this unit requires multiple steps of adding, compare-select and subtraction, a pipeline structure as shown in Figure 6.21 is used to reduce clock cycle requirements to increase the throughput of the decoder.

FIGURE 6.21. Block diagram of the a-posteriori data and LLR calculation unit

Chapter 7: Emulator Performance and Conclusions

A hardware development platform for system verification is needed when software simulation time becomes excessive. This research explores the usage of such a platform to perform verification on the design of one such computation intensive system.

There are two major achievements of this research work. The first one is to improve the simulation environment of Turbo decoder through the implementation of a hardware emulator. The second achievement is certain hardware improvements on the Turbo decoder itself. The following sections present the results of this research.

7.1 Turbo Decoder Simulation Improvements

7.1.1 Reduction of Simulation Time

As an emulator, the TDE greatly reduces the time required for simulation of large frame size compared to software based simulators. The total time¹ required to perform simulation can be approximated using the following equation:

$$\begin{aligned} \text{Delay caused by:} & \quad \underbrace{\text{Create Data}}_{\text{Decode Data}} \quad \underbrace{\text{Verify Output}}_{\text{Decode Data}} \\ M &= \frac{1}{f} F(L + L + L + L + 2LI + L) \\ &= \frac{FL(5 + 2I)}{f} \end{aligned} \tag{7.1}$$

where f is the system clock frequency, F is the number of frames, L is the number of bits in a data frame and I is the number of iterations. The delay in creating data is caused by the processes required to put different kinds of data into the same memory address as shown in the memory map. By doing this, the decoder can reduce the number of memory accesses and reduce the overall simulation time.

1. This total time does not include the small overhead required for state transitions and memory delay for operations such as fetching addresses and flushing data.

The following example gives a measure of possible simulation time. For a high precision bit error rate simulation at a clock speed of 54Mhz, a frame size of 1 Mbit, 1000 frames and 8 iterations, TDE will need 6.5 minutes to finish the simulation. This represents a significant improvement over spending more than 24 hours to run the same simulation in a Matlab model (included in Appendix B) on an AMD Athlon 2600+ computer with 512 MB RAM. Although the reconfigurable logic does not provide the same level of flexibility as software, users of the reconfigurable logic still have the option of modifying the hardware model in Verilog HDL and recompiling the firmware. Typically it takes 10 to 15 mins to regenerate FPGA firmware. Even with the extra time needed to recompile the firmware, using the TDE to perform simulation still has a significant advantage.

7.1.2 Runtime Verification

TDE enables runtime design verification through the usage of the Xilinx Chipscope Pro tool. Users of the TDE can configure parameters such as number of frames, number of iterations, choice of termination and choice of puncturing in the user interface. The status of internal variables can be obtained as numerical values as well as be displayed as waveforms. This allows high flexibility during the verification process of the Turbo decoder design.

7.1.3 Portability to Hardware Implementation

Since the parallel concatenate encoders and Turbo decoder are implemented in synthesizable Verilog HDL, they can be used to synthesize to other families of FPGAs or to a standard cell design. Only parts like the clocking, memory access and I/O circuits would have to be changed.

7.2 Turbo Decoder Hardware Specification and Benchmarking

7.2.1 Special Hardware Features

Fixed Windowing

This Turbo decoder design applies a fixed windowing technique in the Log-MAP decoder for path metric calculation which was suggested by G. Allan at Carleton University [All2002]. As mentioned in Section 6.7.1 this technique allow a reduction in memory requirements, while using fewer calculations than the more common sliding window method. In the fixed windowing

method, the path metrics are pre-initialized, then each subsequent accumulative calculation can generate valid path metrics directly. Fixed windowing also allows the introduction of the overflow protection logic, which make use of the path metrics pre-initialization unit as a back up source of path metrics. When an overflow in a path metrics occur, the pre-initialization unit will refresh the branch metric calculation unit immediately to prevent further corruption of data variables in the Turbo decoder.

ACS Circuit with Correction Factor Parallel Processing

A parallel hardware implementation was applied to the path metric calculation unit. The ACS circuit uses parallelism to reduce logic delays in the generation of path metrics. The two path metric inputs are always assumed to have very close values, and the correction factor is always generated using the lower 4 bits of the two path metrics. Depending on the result of comparing the two path metric inputs, the correction factor can be added to the largest path metric if necessary. This process is executed in parallel to the comparison of the upper 12 bits of the path metrics to reduce the elapsed time of the whole process.

Improved 2's Complement Adder

An improvement in the hardware design was done in the branch metric calculation unit, as described in Section 6.7.3. A special 2's complement adder circuit was implemented to simplify the hardware. This circuit is optimized to perform additions on 2's complement inputs which are selectively inverted. Control inverters first invert soft channel inputs that do not agree with the expected values. Then, the number of inversions needed will be added as an operand in the final addition to complete the 2's complement add operation and generate the final branch metric.

Random Interleaver using LFSR

Another design feature is in the interleaving logics. The random interleaver makes use of the available memory on the development board for hardware simplification. It stored the source addresses as data into the interleaved address location. During deinterleaving, this data can be retrieved to allow reverse permutation of data. This scheme simplified the hardware requirements. Only an up-down counter, a LFSR and combinational logic are needed for each interleaver. The

pseudo random interleaver also gives the full address range and avoids permutation of the termination sequence which allows trellis termination on both Log-Map decoders.

7.2.2 Other Hardware Implementation Features

Other beneficial design practices were implemented in the hardware. First, the channel reliability estimate L_c was made 1 to avoid a multiplication in the calculations. L_c can also be changed in 3 dB steps by changing one parameter and still avoid the multiplication. The data width of data variables such as beta, alpha, gamma, a-posteriori data, a-priori and LLR were selected based on the Matlab system model to give the decoding calculations enough precision.

Finally, in the a-posteriori data and LLR calculation unit, a pipelining architecture was used to enable a higher clock rate and increase data throughput.

7.2.3 TDE Performance Data

After synthesizing the TDE into a Xilinx Virtex II XC2V2000, the hardware performance data was obtained and is listed in Table 7.1.

The critical path is indicated in Figure 7.1. As expected, the worst delay appears in the Path metrics calculation unit which is the sum of the delay from a 2x16 input adder, a 2x12 comparator and a 3x16 input multiplexer.

Categories	Turbo Decoder	Emulator Total
Equivalent gate count	62 K	4 Millions
Number of flip flops	1.7 K	2.8 K
Number of 4-input LUT	5.6 K	6 K
Average flip flop to flip flop Delay	1.8 ns	8.5 ns
Maximum flip flop to flip flop Delay	11.1 ns	15.6 ns
Maximum operating frequency	64 MHz	
Maximum throughput of Log-Map decoder	32 Mbps	
Maximum throughput of Turbo decoder at 4 iterations	4 Mbps	
Maximum throughput of Turbo decoder at 8 iterations	2 Mbps	
Number of 1Mbit frame TDE simulation at 4 iterations	-	295 fpm
Number of 1Mbit frame TDE simulation at 8 iterations	-	182 fpm

TABLE 7.1. Performance data of the TDE FPGA

FIGURE 7.1. Indication of the critical path in the TDE.

This design presents several non-traditional ways of hardware implementation. As shown in Table 7.2, its performance exceeds certain FPGA and DSP Turbo decoders. Its throughput with 8 iterations met the 2Mbps requirement of the 3G wireless standard [3GPP2001]. However, since it is implemented on a FPGA, the performance is not as high as a few others recent Turbo decoder

ASIC designs. This is because logic implemented in ASIC have the flexibility of applying transistor level customization.

Design	Operating Frequency	Throughput	Note
Turbo decoder in TDE	64MHz	4Mbps @ 4 Iterations 2Mbps @ 8 Iterations	FPGA design on Xilinx Virtex II FPGA
[BGP+02]	128.8MHz	2Mbps	ASIC design Tested on Xilinx FPGA
[Fei2000]	57.7	Unspecified	ASIC design on 0.35 μ m CMOS
[GG2003]	Unspecified	13.3Mbps	ASIC design on 0.35 μ m CMOS
[GTO+2005]	200MHz	1.5Mbps	Programmable LSI design on 0.13 μ m CMOS
[KC2004]	Unspecified	6Mbps	Low power ASIC design on 0.13 μ m CMOS
[LAJ2003]	200MHz	1.919Mbps	Analog Device TigerSHARC DSP based design

TABLE 7.2. Comparison of the recent Turbo decoder design specifications

7.3 Construction of a Matlab Model

A Matlab implementation of Turbo decoder was used to prove the level of understanding of the theory as well as finding possible areas where optimization can be done.

The error correcting performance of the Matlab model matches the performance of published literature [BDMP1996], [WB2000] and [BMD2003] in simulation. This indicates the algorithm implemented in the Matlab model is correct and the level of understanding is verified.

Throughout the Matlab simulations, factors that affect the performance of the Turbo decoder were verified as shown in Chapter 5. From the simulation it is noticed that keeping the channel reliability ratio $L_c = 1$ does not affect the error correction performance of the Turbo decoder in a high noise environment. Also, the simulation allows the extraction of all variable ranges and precision required. These informations are useful for the hardware implementation portion of this research.

7.4 Research Extension

There are several possible extensions to this research:

First of all, rearrangement of the memory allocation can possibly make the TDE accommodate trellises with higher number of states.

Secondly, a better noise generator should be built to allow calculation of BER in different situations.

Thirdly, different interleavers can be used to improve the data throughput. With the current random interleaver with a size equal to the frame, the decoder implementation is not taking full advantage of windowing.

And finally, the TDE can give users the ability to adjust the frame length in real time by arranging to restart the LFSR and counter after the counter reaches the desired limit. Inserting a table of the feedback taps for all the desired LFSRs and user configurable end points for the counters in the design would allow the accommodation of different frame length settings.

List of References

- [3GPP2001] 3G Technical Specification, "Multiplexing and Channel Coding, 25.212 v4.2.0, 2001," *Third Generation Partnership Project* <http://www.3gpp.org>
- [All2002] G. Allan, "2.5 mW - 10 Mbps, Low Area MAP Decoder." in *Carleton University Internal Paper*, May 2002. (Fixed Windowing)
- [BCJR1974] 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. IT-20, pp. 284–287, Mar. 1974. (MAP)
- [BDMP1996] S. Benedetto, D. Divsalar, G. Montorsi, and F. Pollara, "Soft-output decoding algorithms in iterative decoding of turbo-codes," in *JPL TDA Progress Rep.*, pp. 42–124, Feb. 1996.
- [BGP+2002] M.A. Bickerstaff, D. Garrett, T. Prokop, C. Thomas, B. Widdup, G. Zhou, L.M. Davis, G. Woodward, C. Nicol and R. Yan, "A Unified Turbo/Viterbi Channel Decoder for 3GPP Mobile Wireless in 0.18- μ m CMOS," in *IEEE J. Solid-State Circuits*, vol. 37, no. 11, pp. 1555-1564, Nov. 2002.
- [BGT1993] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding. Turbo codes," in *Proc. Int. Conf. Communications*, May 1993, vol. 2, pp. 1064–1070. (Turbo)
- [BMD2003] S. Benedetto, G. Montorsi and D. Divsalar, "Concatenated Convolutional Codes with Interleavers," in *IEEE Commun. Magazine*, pp. 102-109, Aug. 2003
- [DDP1998] S. Dolinar, D. Divsalar, and F. Pollara, "Code Performance as a Function of Block Size," in *Telecommun. and Mission Ops. Progress Report 42-133*, NASA, May 1998. (Frame size)
- [DM1995] H. Dawid and H. Meyr, "Real-time algorithms and VLSI architectures for soft output MAP convolutional decoding," in *Proc. Personal, Indoor, and Mobile Radio Communications, PIMRC'95. Wireless: Merging Onto the Information Superhighway*, pp. 193–197, Sep. 1995
- [EPG1994] J. A. Erfanian, S. Pasupathy, and G. Gulak, "Reduced complexity symbol detectors with parallel structures for ISI channels," in *IEEE Trans. Commun.*, vol. 42, pp. 1661–1671, 1994. (Max Log MAP)
- [Fei2000] J. Fei, "On a turbo decoder design for low power dissipation," in *Virginia Polytechnic Institute and State University M.Sc in E.E Thesis*, Jul. 2000.
- [For1966] G.D. Forney, Jr., "Concatenated Codes," *MIT Press*, 1966.

- [GG2003] V.C. Gaudet and P.G. Gulak, "A 13.3-Mb/s 0.35mm CMOS Analog Turbo Decoder IC With a Configurable Interleaver," in *IEEE J. Solid-State Circuits*, vol. 38, no. 11, pp. 2010-2015, Nov. 2003.
- [GTO+2005] H. Gambe, Y. Tanaka, K. Ohbuchi, T. Ishihara and J. Li, "An Improved Sliding Window Algorithm for Max-Log-MAP Turbo Decoder and Its Programmable LSI Implementation," in *IEICE Trans. Electron.*, vol. E88-C, no.3, pp. 403-412, Mar. 2005.
- [[HH1989] J. Hagenauer and P. Hoeher, "A Viterbi algorithm with soft-decision outputs and its applications," in *IEEE Globecom*, vol.3, pp. 1680-1686, 1989. (SOVA)
- [KB1990] W. Koch and A. Baier, "Optimum and sub-optimum detection of coded data disturbed by time-varying inter-symbol interference," in *IEEE Globecom*, vol. 3, pp. 1679-1684, Dec. 1990. (Max Log MAP)
- [KC2004] J. Kaza and C. Chakrabarti, "Design and Implementation of Low-Energy Turbo Decoders," in *IEEE Trans. VLSI Systems*, vol. 12, no. 9, pp. 968-977, Sep. 2004
- [LAJ2003] K.K. Loo, T. Alukaidey, S.A. Jimaa, "High Performance Parallelised 3GPP Turbo Decoder", in *Proc., 5th European Personal Mobile Communication Conference (EMPCC2003)*, pp. 337-342, Apr. 2003
- [LSS2003] S. Lee, N.R. Shanbhag, and A.C. Singer, "A low-power VLSI architecture for Turbo decoding," in *IEEE Proc. International Symposium on Low Power Electronics Design*, pp. 366- 371, 2003.
- [MS2003] M. Mansour and N. R. Shanbhag, "VLSI architectures for SISO-APP decoders," in *IEEE Trans. VLSI Syst.*, vol. 11, no. 2, pp. 627-650, Apr. 2003.
- [[RVH1995] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," in *Proc. Int. Conf. Communications*, vol. 2, pp. 1009-1013, Jun. 1995. (Log MAP)
- [SCE2001] C. Schurgers, F. Catthoor, and M. Engels, "Memory optimization of MAP Turbo decoder algorithms," in *IEEE Transaction on VLSI Syst*, vol. 9, no. 2, pp. 305-312, Apr. 2001.
- [Sha1948] C. E. Shannon, "A mathematical theory of communication," in *Bell System Technical Journal*, vol. 27, pp. 379-423 and 623-656, Jul. and Oct., 1948.
- [Skl1997] B. Sklar, "A primer on turbo code concepts," in *IEEE Commun. Mag.*, pp. 94-102, Dec. 1997. (General Paper)
- [TGV+2002] M. J. Thul, F. Gilbert, T. Vogt, G. Kreiselmaier, and N. Wehn, "A scalable system architecture for high-throughput Turbo decoders," in *IEEE Workshop on Sig. Pro. Syst.*, pp. 152-158, 2002.
- [TZC2004] M. Tiwari, Y. Zhu and C. Chakrabarti, "Memory sub-banking scheme for high

throughput Turbo decoder”, in *ICASSP*, vol. 5, pp 29-32, May 2004.

- [Vit1967] A. J. Viterbi, “Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm,” in *IEEE Transactions on Information Theory*, vol. IT-13, pp. 260-269, Apr. 1967. (Viterbi)
- [Vit1971] A.J. Viterbi, “Convolutional codes and their performance in communication systems.” in *IEEE Trans. Commun.*, vol. 19, pp. 751-772, Oct. 1971. (Conv Performance)
- [Vit1998] A. J. Viterbi, “An intuitive justification of the MAP decoder for convolutional codes,” in *IEEE J. Sel. Areas Commun.*, vol. 16, no. 2, pp. 260–264, Feb. 1998.
- [WB2000] J. Woodward and L. Hanzo, “Comparative Study of Turbo Decoding Techniques: An Overview,” in *IEEE Trans. Vehic. Technol.*, vol. 49, no. 6, pp. 2208-2233, Nov. 2000. (General Paper)
- [WLW2001] A. Worm, H. Lamm, N. Wehn, “Design of low-power high-speed maximum a priori decoder architectures,” in *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*, pp. 258-265, Mar. 2001 (Fixed Windowing)
- [WP2002] Z.Wang, Z.Chi, and K.K.Parhi, “Area-efficient high-speed decoding schemes for Turbo decoders,” in *IEEE VLSI Systems*, vol. 10, no. 6, pp. 902– 912, Dec. 2002.
- [Xil2004] Xilinx, “ChipScope Pro Software and Cores User Manual for v6.3i,” in *Xilinx ChipScope Documentation*, Oct. 2004 .

Appendix A: Verilog Codes

A.1 Top Level Emulator

```
/*
-----
Title   : Turbo Decoder Emulator
-----
File    : turbo.v
Created : 2005/02/19
Last Update: 2005/02/19
-----
Uses    : interleaver.v, create_data.v, decoder.v, Memory_Interface.v
          CLOCK_GEN.v, defines.v
-----
Description: This module integrate encoder for turbo code and turbo decoder
              using logMAP algorithm. It measure the performance of the decoder
              by tracking the number of error.

Conventions:
All EXTERNAL port signals are UPPER CASE.

All internal signals are LOWER CASE and are active HIGH.

-----
*/
`include "../rtl/Mem_Inf/DEFINES.v"

//-----
//
// Main module declaration
//
//-----

module Turbo(
usr_rst, usr_en, usr_start, num_frame, status, iteration, num_error, SNR_select,

MASTER_CLOCK_P, ALTERNATE_CLOCK_P, MEM_CLK_FBOUT_P, MEM_CLK_FBIN_P, EXTEND_DCM_RESET_P,

MEMORY_BANK0_CLK_P, MEMORY_BANK0_CLKEN_N, MEMORY_BANK0_WEN_N,
MEMORY_BANK0_WENA_N, MEMORY_BANK0_WENB_N, MEMORY_BANK0_WENC_N, MEMORY_BANK0_WEND_N,
MEMORY_BANK0_ADV_LD_N, MEMORY_BANK0_OEN_N, MEMORY_BANK0_CEN_N, MEMORY_BANK0_ADDR_P,
MEMORY_BANK0_DATA_A_P, MEMORY_BANK0_DATA_B_P, MEMORY_BANK0_DATA_C_P, MEMORY_BANK0_DATA_D_P,

MEMORY_BANK1_CLK_P, MEMORY_BANK1_CLKEN_N, MEMORY_BANK1_WEN_N,
MEMORY_BANK1_WENA_N, MEMORY_BANK1_WENB_N, MEMORY_BANK1_WENC_N, MEMORY_BANK1_WEND_N,
MEMORY_BANK1_ADV_LD_N, MEMORY_BANK1_OEN_N, MEMORY_BANK1_CEN_N, MEMORY_BANK1_ADDR_P,
MEMORY_BANK1_DATA_A_P, MEMORY_BANK1_DATA_B_P, MEMORY_BANK1_DATA_C_P, MEMORY_BANK1_DATA_D_P,

MEMORY_BANK2_CLK_P, MEMORY_BANK2_CLKEN_N, MEMORY_BANK2_WEN_N,
MEMORY_BANK2_WENA_N, MEMORY_BANK2_WENB_N, MEMORY_BANK2_WENC_N, MEMORY_BANK2_WEND_N,
MEMORY_BANK2_ADV_LD_N, MEMORY_BANK2_OEN_N, MEMORY_BANK2_CEN_N, MEMORY_BANK2_ADDR_P,
MEMORY_BANK2_DATA_A_P, MEMORY_BANK2_DATA_B_P, MEMORY_BANK2_DATA_C_P, MEMORY_BANK2_DATA_D_P,

MEMORY_BANK3_CLK_P, MEMORY_BANK3_CLKEN_N, MEMORY_BANK3_WEN_N,
MEMORY_BANK3_WENA_N, MEMORY_BANK3_WENB_N, MEMORY_BANK3_WENC_N, MEMORY_BANK3_WEND_N,
MEMORY_BANK3_ADV_LD_N, MEMORY_BANK3_OEN_N, MEMORY_BANK3_CEN_N, MEMORY_BANK3_ADDR_P,
MEMORY_BANK3_DATA_A_P, MEMORY_BANK3_DATA_B_P, MEMORY_BANK3_DATA_C_P, MEMORY_BANK3_DATA_D_P,
```

```

MEMORY_BANK4_CLK_P, MEMORY_BANK4_CLKEN_N, MEMORY_BANK4_WEN_N,
MEMORY_BANK4_WENA_N, MEMORY_BANK4_WENB_N, MEMORY_BANK4_WENC_N, MEMORY_BANK4_WEND_N,
MEMORY_BANK4_ADV_LD_N, MEMORY_BANK4_OEN_N, MEMORY_BANK4_CEN_N, MEMORY_BANK4_ADDR_P,
MEMORY_BANK4_DATA_A_P, MEMORY_BANK4_DATA_B_P, MEMORY_BANK4_DATA_C_P, MEMORY_BANK4_DATA_D_P,

debug_clk, turbo_debug, enc_debug, dec_debug, dec_mux2_1b, mem_debug, mem_mux23_01b);

parameter a_priori_width = 8;
parameter LLR_width = 16;

parameter frame_index = 19;
parameter end_address = -1;

// User I/O pins
input usr_rst, usr_en, usr_start, SNR_select;
input[9:0] num_frame;
output[1:0] status;
input[3:0] iteration;
output[7:0] num_error;

// Clock pins
input MASTER_CLOCK_P; // master clock input (27MHz)
input ALTERNATE_CLOCK_P; // alternate clock input (50MHz)
input EXTEND_DCM_RESET_P; // signal to extend the DCM reset
input MEM_CLK_FBIN_P; // memory DCM feedback loop input
output MEM_CLK_FBOUT_P; // memory DCM feedback loop output

// Memory pins
output MEMORY_BANK0_CLK_P; // ZBT RAM clock
output MEMORY_BANK0_CLKEN_N; // ZBT RAM clock enable
output MEMORY_BANK0_WEN_N; // ZBT RAM write enable
output MEMORY_BANK0_WENA_N; // ZBT RAM byte A write enable
output MEMORY_BANK0_WENB_N; // ZBT RAM byte B write enable
output MEMORY_BANK0_WENC_N; // ZBT RAM byte C write enable
output MEMORY_BANK0_WEND_N; // ZBT RAM byte D write enable
output MEMORY_BANK0_ADV_LD_N; // ZBT RAM address load-advance
output MEMORY_BANK0_OEN_N; // ZBT RAM output enable
output MEMORY_BANK0_CEN_N; // ZBT RAM chip enable
output [(C_ADDR_BITS-1):0] MEMORY_BANK0_ADDR_P; // ZBT RAM address bus
inout [7:0] MEMORY_BANK0_DATA_A_P; // ZBT RAM data bus byte "A"
inout [7:0] MEMORY_BANK0_DATA_B_P; // ZBT RAM data bus byte "B"
inout [7:0] MEMORY_BANK0_DATA_C_P; // ZBT RAM data bus byte "C"
inout [7:0] MEMORY_BANK0_DATA_D_P; // ZBT RAM data bus byte "D"

output MEMORY_BANK1_CLK_P; // ZBT RAM clock
output MEMORY_BANK1_CLKEN_N; // ZBT RAM clock enable
output MEMORY_BANK1_WEN_N; // ZBT RAM write enable
output MEMORY_BANK1_WENA_N; // ZBT RAM byte A write enable
output MEMORY_BANK1_WENB_N; // ZBT RAM byte B write enable
output MEMORY_BANK1_WENC_N; // ZBT RAM byte C write enable
output MEMORY_BANK1_WEND_N; // ZBT RAM byte D write enable
output MEMORY_BANK1_ADV_LD_N; // ZBT RAM address load-advance
output MEMORY_BANK1_OEN_N; // ZBT RAM output enable
output MEMORY_BANK1_CEN_N; // ZBT RAM chip enable
output [(C_ADDR_BITS-1):0] MEMORY_BANK1_ADDR_P; // ZBT RAM address bus
inout [7:0] MEMORY_BANK1_DATA_A_P; // ZBT RAM data bus byte "A"
inout [7:0] MEMORY_BANK1_DATA_B_P; // ZBT RAM data bus byte "B"
inout [7:0] MEMORY_BANK1_DATA_C_P; // ZBT RAM data bus byte "C"
inout [7:0] MEMORY_BANK1_DATA_D_P; // ZBT RAM data bus byte "D"

output MEMORY_BANK2_CLK_P; // ZBT RAM clock
output MEMORY_BANK2_CLKEN_N; // ZBT RAM clock enable
output MEMORY_BANK2_WEN_N; // ZBT RAM write enable
output MEMORY_BANK2_WENA_N; // ZBT RAM byte A write enable
output MEMORY_BANK2_WENB_N; // ZBT RAM byte B write enable

```

```

output    MEMORY_BANK2_WENC_N;// ZBT RAM byte C write enable
output    MEMORY_BANK2_WEND_N;// ZBT RAM byte D write enable
output    MEMORY_BANK2_ADV_LD_N;// ZBT RAM address load-advance
output    MEMORY_BANK2_OEN_N;// ZBT RAM output enable
output    MEMORY_BANK2_CEN_N;// ZBT RAM chip enable
output [(ADDR_BITS-1):0]MEMORY_BANK2_ADDR_P;// ZBT RAM address bus
inout [7:0] MEMORY_BANK2_DATA_A_P;// ZBT RAM data bus byte "A"
inout [7:0] MEMORY_BANK2_DATA_B_P;// ZBT RAM data bus byte "B"
inout [7:0] MEMORY_BANK2_DATA_C_P;// ZBT RAM data bus byte "C"
inout [7:0] MEMORY_BANK2_DATA_D_P;// ZBT RAM data bus byte "D"

output    MEMORY_BANK3_CLK_P;// ZBT RAM clock
output    MEMORY_BANK3_CLKEN_N;// ZBT RAM clock enable
output    MEMORY_BANK3_WEN_N;// ZBT RAM write enable
output    MEMORY_BANK3_WENA_N;// ZBT RAM byte A write enable
output    MEMORY_BANK3_WENB_N;// ZBT RAM byte B write enable
output    MEMORY_BANK3_WENC_N;// ZBT RAM byte C write enable
output    MEMORY_BANK3_WEND_N;// ZBT RAM byte D write enable
output    MEMORY_BANK3_ADV_LD_N;// ZBT RAM address load-advance
output    MEMORY_BANK3_OEN_N;// ZBT RAM output enable
output    MEMORY_BANK3_CEN_N;// ZBT RAM chip enable
output [(ADDR_BITS-1):0]MEMORY_BANK3_ADDR_P;// ZBT RAM address bus
inout [7:0] MEMORY_BANK3_DATA_A_P;// ZBT RAM data bus byte "A"
inout [7:0] MEMORY_BANK3_DATA_B_P;// ZBT RAM data bus byte "B"
inout [7:0] MEMORY_BANK3_DATA_C_P;// ZBT RAM data bus byte "C"
inout [7:0] MEMORY_BANK3_DATA_D_P;// ZBT RAM data bus byte "D"

output    MEMORY_BANK4_CLK_P;// ZBT RAM clock
output    MEMORY_BANK4_CLKEN_N;// ZBT RAM clock enable
output    MEMORY_BANK4_WEN_N;// ZBT RAM write enable
output    MEMORY_BANK4_WENA_N;// ZBT RAM byte A write enable
output    MEMORY_BANK4_WENB_N;// ZBT RAM byte B write enable
output    MEMORY_BANK4_WENC_N;// ZBT RAM byte C write enable
output    MEMORY_BANK4_WEND_N;// ZBT RAM byte D write enable
output    MEMORY_BANK4_ADV_LD_N;// ZBT RAM address load-advance
output    MEMORY_BANK4_OEN_N;// ZBT RAM output enable
output    MEMORY_BANK4_CEN_N;// ZBT RAM chip enable
output [(ADDR_BITS-1):0]MEMORY_BANK4_ADDR_P;// ZBT RAM address bus
inout [7:0] MEMORY_BANK4_DATA_A_P;// ZBT RAM data bus byte "A"
inout [7:0] MEMORY_BANK4_DATA_B_P;// ZBT RAM data bus byte "B"
inout [7:0] MEMORY_BANK4_DATA_C_P;// ZBT RAM data bus byte "C"
inout [7:0] MEMORY_BANK4_DATA_D_P;// ZBT RAM data bus byte "D"

output debug_clk;
output [63:1] turbo_debug;
output [63:1] enc_debug;
output [158:0] dec_debug;
input dec_mux2_1b;
output [158:0] mem_debug;
input mem_mux23_01b;

////////////////////////////////////
// USER I/O
wire[1:0] status;
reg[7:0] num_error;

// CLK GENERATOR SIGNAL
wire    MEM_CLK_FBOUT_P;
wire    clk_27mhz;           // internal 27MHz clock
wire    clk_54mhz;           // internal 54MHz clock
wire    clk_108mhz;          // internal 108MHz clock
wire    pixel_clock;         // pixel clock for the SVGA controller
wire    memory_27mhz;        // 27MHz clock for the ZBT memories
wire    memory_54mhz;        // 54MHz clock for the ZBT memories
wire    memory_108mhz;       // 108MHz clock for the ZBT memories

```

```

wire    pal_mode;           // selects PAL or NTSC clock as pixel_clock
wire    fpga_reset;        // reset to fpga until all DCMs are locked

// INTERLEAVER SIGNALS
reg int_A_en, int_B_en;
reg int_A_clr, int_B_clr;
reg int_A_mode, int_B_mode;
wire [18:0] addr_A, addr_B;
wire [18:0] rand_addr_A, rand_addr_B;

// User Memory Interface
reg [8:0]  ctrl_mem0_bus;
reg [18:0] addr_mem0_bus;
wire [31:0] read_mem0_bus;
reg [31:0] write_mem0_bus;

reg [8:0]  ctrl_mem1_bus;
reg [18:0] addr_mem1_bus;
wire [31:0] read_mem1_bus;
reg [31:0] write_mem1_bus;

reg [8:0]  ctrl_mem2_bus;
reg [18:0] addr_mem2_bus;
wire [31:0] read_mem2_bus;
reg [31:0] write_mem2_bus;

wire [8:0] ctrl_mem3_bus;
wire [18:0] addr_mem3_bus;
wire [31:0] read_mem3_bus;
wire [31:0] write_mem3_bus;

wire [8:0] ctrl_mem4_bus;
wire [18:0] addr_mem4_bus;
wire [31:0] read_mem4_bus;
wire [31:0] write_mem4_bus;

// Comparator Signals
reg comp_memory_bank0_clocken;
reg comp_memory_bank0_load_addr;
reg comp_memory_bank0_read;
reg comp_memory_bank0_ce;

reg comp_memory_bank1_clocken;
reg comp_memory_bank1_load_addr;
reg comp_memory_bank1_read;
reg comp_memory_bank1_ce;

wire [8:0] ctrl_mem0_bus_cmp;
wire [18:0] addr_mem0_bus_cmp;
wire [31:0] write_mem0_bus_cmp;

wire [8:0] ctrl_mem1_bus_cmp;
wire [18:0] addr_mem1_bus_cmp;
wire [31:0] write_mem1_bus_cmp;

wire [8:0] ctrl_mem2_bus_cmp;
wire [18:0] addr_mem2_bus_cmp;
wire [31:0] write_mem2_bus_cmp;

assign ctrl_mem0_bus_cmp[8] = comp_memory_bank0_clocken;
assign ctrl_mem0_bus_cmp[7] = comp_memory_bank0_load_addr;
assign ctrl_mem0_bus_cmp[6] = comp_memory_bank0_read;
assign ctrl_mem0_bus_cmp[5:1] = 5'b00000;
assign ctrl_mem0_bus_cmp[0] = comp_memory_bank0_ce;

```

```

assign ctrl_mem0_bus_cmp[8] = comp_memory_bank0_clocken;
assign ctrl_mem0_bus_cmp[7] = comp_memory_bank0_load_addr;
assign ctrl_mem0_bus_cmp[6] = comp_memory_bank0_read;
assign ctrl_mem0_bus_cmp[5:1] = 5'b00000;
assign ctrl_mem0_bus_cmp[0] = comp_memory_bank0_ce;

assign ctrl_mem0_bus_cmp[8] = comp_memory_bank0_clocken;
assign ctrl_mem0_bus_cmp[7] = comp_memory_bank0_load_addr;
assign ctrl_mem0_bus_cmp[6] = comp_memory_bank0_read;
assign ctrl_mem0_bus_cmp[5:1] = 5'b00000;
assign ctrl_mem0_bus_cmp[0] = comp_memory_bank0_ce;

// ENCODER SIGNALS
reg enc_en;
wire enc_done;
wire [2:0] int_A_ctrl_enc, int_B_ctrl_enc;

wire [8:0] ctrl_mem0_bus_enc;
wire [18:0] addr_mem0_bus_enc;
wire [31:0] write_mem0_bus_enc;

wire [8:0] ctrl_mem1_bus_enc;
wire [18:0] addr_mem1_bus_enc;
wire [31:0] write_mem1_bus_enc;

wire [8:0] ctrl_mem2_bus_enc;
wire [18:0] addr_mem2_bus_enc;
wire [31:0] write_mem2_bus_enc;

// DECODER SIGNALS
reg dec_en;
wire dec_done;
wire [2:0] int_A_ctrl_dec, int_B_ctrl_dec;

wire [8:0] ctrl_mem0_bus_dec;
wire [18:0] addr_mem0_bus_dec;
wire [31:0] write_mem0_bus_dec;

wire [8:0] ctrl_mem1_bus_dec;
wire [18:0] addr_mem1_bus_dec;
wire [31:0] write_mem1_bus_dec;

wire [8:0] ctrl_mem2_bus_dec;
wire [18:0] addr_mem2_bus_dec;
wire [31:0] write_mem2_bus_dec;

// INTERFACE TO ZBT RAM
wire MEMORY_BANK0_CLK_P; // ZBT RAM clock
wire MEMORY_BANK0_CLKEN_N; // ZBT RAM clock enable
wire MEMORY_BANK0_WEN_N; // ZBT RAM write enable
wire MEMORY_BANK0_WENA_N; // ZBT RAM byte A write enable
wire MEMORY_BANK0_WENB_N; // ZBT RAM byte B write enable
wire MEMORY_BANK0_WENC_N; // ZBT RAM byte C write enable
wire MEMORY_BANK0_WEND_N; // ZBT RAM byte D write enable
wire MEMORY_BANK0_ADV_LD_N; // ZBT RAM address load-advance
wire MEMORY_BANK0_OEN_N; // ZBT RAM wire enable
wire MEMORY_BANK0_CEN_N; // ZBT RAM chip enable
wire [0:31] MEMORY_BANK0_ADDR_P; // ZBT RAM address bus
wire [7:0] MEMORY_BANK0_DATA_A_P; // ZBT RAM data bus byte "A"
wire [7:0] MEMORY_BANK0_DATA_B_P; // ZBT RAM data bus byte "B"
wire [7:0] MEMORY_BANK0_DATA_C_P; // ZBT RAM data bus byte "C"
wire [7:0] MEMORY_BANK0_DATA_D_P; // ZBT RAM data bus byte "D"

wire MEMORY_BANK1_CLK_P; // ZBT RAM clock
wire MEMORY_BANK1_CLKEN_N; // ZBT RAM clock enable

```

```

wire    MEMORY_BANK1_WEN_N;// ZBT RAM write enable
wire    MEMORY_BANK1_WENA_N;// ZBT RAM byte A write enable
wire    MEMORY_BANK1_WENB_N;// ZBT RAM byte B write enable
wire    MEMORY_BANK1_WENC_N;// ZBT RAM byte C write enable
wire    MEMORY_BANK1_WEND_N;// ZBT RAM byte D write enable
wire    MEMORY_BANK1_ADV_LD_N;// ZBT RAM address load-advance
wire    MEMORY_BANK1_OEN_N;// ZBT RAM wire enable
wire    MEMORY_BANK1_CEN_N;// ZBT RAM chip enable
wire [(C_ADDR_BITS-1):0]MEMORY_BANK1_ADDR_P;// ZBT RAM address bus
wire [7:0] MEMORY_BANK1_DATA_A_P;// ZBT RAM data bus byte "A"
wire [7:0] MEMORY_BANK1_DATA_B_P;// ZBT RAM data bus byte "B"
wire [7:0] MEMORY_BANK1_DATA_C_P;// ZBT RAM data bus byte "C"
wire [7:0] MEMORY_BANK1_DATA_D_P;// ZBT RAM data bus byte "D"

wire    MEMORY_BANK2_CLK_P;// ZBT RAM clock
wire    MEMORY_BANK2_CLKEN_N;// ZBT RAM clock enable
wire    MEMORY_BANK2_WEN_N;// ZBT RAM write enable
wire    MEMORY_BANK2_WENA_N;// ZBT RAM byte A write enable
wire    MEMORY_BANK2_WENB_N;// ZBT RAM byte B write enable
wire    MEMORY_BANK2_WENC_N;// ZBT RAM byte C write enable
wire    MEMORY_BANK2_WEND_N;// ZBT RAM byte D write enable
wire    MEMORY_BANK2_ADV_LD_N;// ZBT RAM address load-advance
wire    MEMORY_BANK2_OEN_N;// ZBT RAM wire enable
wire    MEMORY_BANK2_CEN_N;// ZBT RAM chip enable
wire [(C_ADDR_BITS-1):0]MEMORY_BANK2_ADDR_P;// ZBT RAM address bus
wire [7:0] MEMORY_BANK2_DATA_A_P;// ZBT RAM data bus byte "A"
wire [7:0] MEMORY_BANK2_DATA_B_P;// ZBT RAM data bus byte "B"
wire [7:0] MEMORY_BANK2_DATA_C_P;// ZBT RAM data bus byte "C"
wire [7:0] MEMORY_BANK2_DATA_D_P;// ZBT RAM data bus byte "D"

wire    MEMORY_BANK3_CLK_P;// ZBT RAM clock
wire    MEMORY_BANK3_CLKEN_N;// ZBT RAM clock enable
wire    MEMORY_BANK3_WEN_N;// ZBT RAM write enable
wire    MEMORY_BANK3_WENA_N;// ZBT RAM byte A write enable
wire    MEMORY_BANK3_WENB_N;// ZBT RAM byte B write enable
wire    MEMORY_BANK3_WENC_N;// ZBT RAM byte C write enable
wire    MEMORY_BANK3_WEND_N;// ZBT RAM byte D write enable
wire    MEMORY_BANK3_ADV_LD_N;// ZBT RAM address load-advance
wire    MEMORY_BANK3_OEN_N;// ZBT RAM wire enable
wire    MEMORY_BANK3_CEN_N;// ZBT RAM chip enable
wire [(C_ADDR_BITS-1):0]MEMORY_BANK3_ADDR_P;// ZBT RAM address bus
wire [7:0] MEMORY_BANK3_DATA_A_P;// ZBT RAM data bus byte "A"
wire [7:0] MEMORY_BANK3_DATA_B_P;// ZBT RAM data bus byte "B"
wire [7:0] MEMORY_BANK3_DATA_C_P;// ZBT RAM data bus byte "C"
wire [7:0] MEMORY_BANK3_DATA_D_P;// ZBT RAM data bus byte "D"

wire    MEMORY_BANK4_CLK_P;// ZBT RAM clock
wire    MEMORY_BANK4_CLKEN_N;// ZBT RAM clock enable
wire    MEMORY_BANK4_WEN_N;// ZBT RAM write enable
wire    MEMORY_BANK4_WENA_N;// ZBT RAM byte A write enable
wire    MEMORY_BANK4_WENB_N;// ZBT RAM byte B write enable
wire    MEMORY_BANK4_WENC_N;// ZBT RAM byte C write enable
wire    MEMORY_BANK4_WEND_N;// ZBT RAM byte D write enable
wire    MEMORY_BANK4_ADV_LD_N;// ZBT RAM address load-advance
wire    MEMORY_BANK4_OEN_N;// ZBT RAM wire enable
wire    MEMORY_BANK4_CEN_N;// ZBT RAM chip enable
wire [(C_ADDR_BITS-1):0]MEMORY_BANK4_ADDR_P;// ZBT RAM address bus
wire [7:0] MEMORY_BANK4_DATA_A_P;// ZBT RAM data bus byte "A"
wire [7:0] MEMORY_BANK4_DATA_B_P;// ZBT RAM data bus byte "B"
wire [7:0] MEMORY_BANK4_DATA_C_P;// ZBT RAM data bus byte "C"
wire [7:0] MEMORY_BANK4_DATA_D_P;// ZBT RAM data bus byte "D"

// Internal Signals
wire clk, rst;
reg clr;

```

```

reg [2:0] uc;
reg uc_en;
reg int_cmp_en, int_cmp_clr;
reg [9:0] num_frame_reg;

localparam [2:0] c1 = 3'b100;
localparam [2:0] c2 = 3'b111;

reg [1:0] state, next_state;
localparam [1:0] IDLE = 2'b00;
localparam [1:0] ENCODE = 2'b01;
localparam [1:0] DECODE = 2'b11;
localparam [1:0] COMPARE = 2'b10;

assign status = state;

wire enc_in, dec_out;
reg error, frame_inc;

assign enc_in = read_mem0_bus[16];
assign dec_out = read_mem1_bus[31];

assign clk = clk_27mhz;
assign zbt_clk = memory_27mhz;
assign rst = fpga_reset | usr_rst;

always @ (posedge clk or posedge rst) begin
    if(rst) begin
        state <= IDLE;
        num_error <= 0;
        num_frame_reg <= 0;
        uc <= 0;
    end
    else if (clr) begin
        state <= next_state;
        num_error <= 0;
        num_frame_reg <= 0;
        uc <= 0;
    end
    else if (usr_en) begin
        state <= next_state;
        if(error) num_error <= num_error + 1;
        if(frame_inc) num_frame_reg <= num_frame_reg + 1;
        if(uc_en) uc <= uc + 1;
    end
end

always @ (state, usr_start, enc_done, dec_done, num_frame, num_frame_reg) begin
    next_state = state;
    case(state)
        IDLE: if(usr_start) next_state = ENCODE;
        ENCODE: if(enc_done) next_state = DECODE;
        DECODE: if(dec_done) next_state = COMPARE;
        COMPARE: begin
            if(num_frame_reg <= num_frame) next_state = ENCODE;
            else next_state = IDLE;
        end
    endcase
end

always @ (state, usr_start, enc_in, dec_out, addr_A, uc) begin
    enc_en = 0;
    dec_en = 0;
    error = 0;
    frame_inc = 0;

```

```

clr = 0;
uc_en = 0;
int_cmp_en = 0;
int_cmp_clr = 0;

comp_memory_bank0_clocken = 1'b0;
comp_memory_bank0_load_addr = 1'b0;
comp_memory_bank0_read = 1'b0;
comp_memory_bank0_ce = 1'b0;

comp_memory_bank1_clocken = 1'b0;
comp_memory_bank1_load_addr = 1'b0;
comp_memory_bank1_read = 1'b0;
comp_memory_bank1_ce = 1'b0;

case(state)
  IDLE: if(usr_start) clr = 1;
  ENCODE: enc_en = 1;
  DECODE: dec_en = 1;
  COMPARE: begin
    if (uc < c1) begin // fetch address
      uc_en = 1;
      int_cmp_en = 1;

      comp_memory_bank0_clocken = 1'b1;
      comp_memory_bank0_load_addr = 1'b1;
      comp_memory_bank0_read = 1'b1;
      comp_memory_bank0_ce = 1'b1;

      comp_memory_bank1_clocken = 1'b1;
      comp_memory_bank1_load_addr = 1'b1;
      comp_memory_bank1_read = 1'b1;
      comp_memory_bank1_ce = 1'b1;
    end
    else if(uc == c1) begin // compare input to output
      int_cmp_en = 1;

      comp_memory_bank0_clocken = 1'b1;
      comp_memory_bank0_load_addr = 1'b1;
      comp_memory_bank0_read = 1'b1;
      comp_memory_bank0_ce = 1'b1;

      comp_memory_bank1_clocken = 1'b1;
      comp_memory_bank1_load_addr = 1'b1;
      comp_memory_bank1_read = 1'b1;
      comp_memory_bank1_ce = 1'b1;

      if (enc_in != dec_out) error = 1;
      if (addr_A == end_address) uc_en = 1;
    end
    else if (uc <= c2) begin // flush data
      uc_en = 1;

      comp_memory_bank0_clocken = 1'b1;
      comp_memory_bank0_load_addr = 1'b0;
      comp_memory_bank0_read = 1'b1;
      comp_memory_bank0_ce = 1'b1;

      comp_memory_bank1_clocken = 1'b1;
      comp_memory_bank1_load_addr = 1'b0;
      comp_memory_bank1_read = 1'b1;
      comp_memory_bank1_ce = 1'b1;

      if (enc_in != dec_out) error = 1;
      if (uc == c2) int_cmp_clr = 1;
    end
  end
endcase

```



```

        end

    end

endcase

end

// Connecting user memory interface
always @ (enc_en, dec_en,
    int_cmp_en, int_cmp_clr,
    ctrl_mem0_bus_cmp, ctrl_mem1_bus_cmp, ctrl_mem2_bus_cmp,
    addr_mem0_bus_cmp, addr_mem1_bus_cmp, addr_mem2_bus_cmp,
    write_mem0_bus_cmp, write_mem1_bus_cmp, write_mem2_bus_cmp,
    int_A_ctrl_enc, int_B_ctrl_enc,
    ctrl_mem0_bus_enc, ctrl_mem1_bus_enc, ctrl_mem2_bus_enc,
    addr_mem0_bus_enc, addr_mem1_bus_enc, addr_mem2_bus_enc,
    write_mem0_bus_enc, write_mem1_bus_enc, write_mem2_bus_enc,
    int_A_ctrl_dec, int_B_ctrl_dec,
    ctrl_mem0_bus_dec, ctrl_mem1_bus_dec, ctrl_mem2_bus_dec,
    addr_mem0_bus_dec, addr_mem1_bus_dec, addr_mem2_bus_dec,
    write_mem0_bus_dec, write_mem1_bus_dec, write_mem2_bus_dec)
begin
    casex ({enc_en, dec_en})
        2'b1x: begin
            int_A_en = int_A_ctrl_enc[2];
            int_A_clr = int_A_ctrl_enc[1];
            int_A_mode = int_A_ctrl_enc[0];

            int_B_en = int_B_ctrl_enc[2];
            int_B_clr = int_B_ctrl_enc[1];
            int_B_mode = int_B_ctrl_enc[0];

            ctrl_mem0_bus = ctrl_mem0_bus_enc;
            ctrl_mem1_bus = ctrl_mem1_bus_enc;
            ctrl_mem2_bus = ctrl_mem2_bus_enc;

            addr_mem0_bus = addr_mem0_bus_enc;
            addr_mem1_bus = addr_mem1_bus_enc;
            addr_mem2_bus = addr_mem2_bus_enc;

            write_mem0_bus = write_mem0_bus_enc;
            write_mem1_bus = write_mem1_bus_enc;
            write_mem2_bus = write_mem2_bus_enc;
        end
        2'b0x1: begin
            int_A_en = int_A_ctrl_dec[2];
            int_A_clr = int_A_ctrl_dec[1];
            int_A_mode = int_A_ctrl_dec[0];

            int_B_en = int_B_ctrl_dec[2];
            int_B_clr = int_B_ctrl_dec[1];
            int_B_mode = int_B_ctrl_dec[0];

            ctrl_mem0_bus = ctrl_mem0_bus_dec;
            ctrl_mem1_bus = ctrl_mem1_bus_dec;
            ctrl_mem2_bus = ctrl_mem2_bus_dec;

            addr_mem0_bus = addr_mem0_bus_dec;
            addr_mem1_bus = addr_mem1_bus_dec;
            addr_mem2_bus = addr_mem2_bus_dec;

            write_mem0_bus = write_mem0_bus_dec;
            write_mem1_bus = write_mem1_bus_dec;
            write_mem2_bus = write_mem2_bus_dec;
        end
        default: begin

```

```

        int_A_en = int_cmp_en;
        int_A_clr = int_cmp_clr;
        int_A_mode = 1;

        int_B_en = 0;
        int_B_clr = 0;
        int_B_mode = 0;

        ctrl_mem0_bus = ctrl_mem0_bus_cmp;
        ctrl_mem1_bus = ctrl_mem1_bus_cmp;
        ctrl_mem2_bus = ctrl_mem2_bus_cmp;

        addr_mem0_bus = addr_mem0_bus_cmp;
        addr_mem1_bus = addr_mem1_bus_cmp;
        addr_mem2_bus = addr_mem2_bus_cmp;

        write_mem0_bus = write_mem0_bus_cmp;
        write_mem1_bus = write_mem1_bus_cmp;
        write_mem2_bus = write_mem2_bus_cmp;
    end
endcase
end

/*// Connecting user memory interface
assign ctrl_mem0_bus[8] = (oe_comp)? comp_memory_bank0_clocken : 1'bz;
assign ctrl_mem0_bus[7] = (oe_comp)? comp_memory_bank0_load_addr : 1'bz;
assign ctrl_mem0_bus[6] = (oe_comp)? comp_memory_bank0_read : 1'bz;
assign ctrl_mem0_bus[0] = (oe_comp)? comp_memory_bank0_ce : 1'bz;

assign ctrl_mem1_bus[8] = (oe_comp)? comp_memory_bank1_clocken : 1'bz;
assign ctrl_mem1_bus[7] = (oe_comp)? comp_memory_bank1_load_addr : 1'bz;
assign ctrl_mem1_bus[6] = (oe_comp)? comp_memory_bank1_read : 1'bz;
assign ctrl_mem1_bus[0] = (oe_comp)? comp_memory_bank1_ce : 1'bz;

assign addr_mem0_bus = (oe_comp) ? addr_A : 19'hzzzzz;
assign addr_mem1_bus = (oe_comp) ? addr_A : 19'hzzzzz;

assign write_mem0_bus = (enc_en) ? enc_write_mem0_bus : 32'hzzzz_zzzz;
assign write_mem0_bus = (dec_en) ? dec_write_mem0_bus : 32'hzzzz_zzzz;
assign write_mem1_bus = (enc_en) ? enc_write_mem1_bus : 32'hzzzz_zzzz;
assign write_mem1_bus = (dec_en) ? dec_write_mem1_bus : 32'hzzzz_zzzz;
assign write_mem2_bus = (enc_en) ? enc_write_mem2_bus : 32'hzzzz_zzzz;
assign write_mem2_bus = (dec_en) ? dec_write_mem2_bus : 32'hzzzz_zzzz;
*/

// DEBUG SIGNALS

wire debug_clk;
wire [63:1] turbo_debug;
wire [63:1] enc_debug;
wire [158:0] dec_debug;
wire [158:0] mem_debug;

assign turbo_debug[63] = clk;
assign turbo_debug[62] = rst;
assign turbo_debug[61] = usr_start;
assign turbo_debug[60] = SNR_select;
assign turbo_debug[59] = enc_en;
assign turbo_debug[58] = dec_en;
assign turbo_debug[57] = frame_inc;
assign turbo_debug[56] = uc_en;
assign turbo_debug[55] = int_cmp_en;
assign turbo_debug[54] = int_cmp_clr;
assign turbo_debug[53] = 0;
assign turbo_debug[52] = error;

```

```

assign turbo_debug[51:49] = uc;
assign turbo_debug[48:45] = iteration;
//assign turbo_debug[44:37] = num_error;
assign turbo_debug[44:41] = addr_A[3:0];
assign turbo_debug[40:37] = addr_B[3:0];

assign turbo_debug[36:33] = num_frame[9:6];

assign turbo_debug[32] = int_A_en;
assign turbo_debug[31] = int_A_clr;
assign turbo_debug[30] = int_A_mode;
assign turbo_debug[29] = int_B_en;
assign turbo_debug[28] = int_B_clr;
assign turbo_debug[27] = int_B_mode;

assign turbo_debug[26:18] = num_frame_reg;
assign turbo_debug[17:16] = state;
assign turbo_debug[15:1] = read_mem1_bus[31:17]; //LLR
//turbo_debug[0] will be disconnect for the use of trigger_in

assign debug_clk = clk_54mhz;

CLOCKGEN clock_i(
MASTER_CLOCK_P, ALTERNATE_CLOCK_P, MEM_CLK_FBOUT_P, MEM_CLK_FBIN_P, EXTEND_DCM_RESET_P,
clk_27mhz, clk_54mhz, clk_108mhz, pixel_clock, memory_27mhz, memory_54mhz, memory_108mhz,
pal_mode, fpga_reset
);

interleaver #(.size(frame_index))
int_A(clk, rst, int_A_en, int_A_clr, int_A_mode, addr_A, rand_addr_A);

interleaver #(.size(frame_index))
int_B(clk, rst, int_B_en, int_B_clr, int_B_mode, addr_B, rand_addr_B);

create_data #(.frame_index(frame_index), .end_address(end_address))
create_data_i(
clk, rst, enc_en, enc_done,
ctrl_mem0_bus_enc, addr_mem0_bus_enc, read_mem0_bus, write_mem0_bus_enc,
ctrl_mem1_bus_enc, addr_mem1_bus_enc, read_mem1_bus, write_mem1_bus_enc,
ctrl_mem2_bus_enc, addr_mem2_bus_enc, read_mem2_bus, write_mem2_bus_enc,
int_A_ctrl_enc, addr_A, rand_addr_A,
int_B_ctrl_enc, addr_B, rand_addr_B,
enc_debug);

decoder #(.a_priori_width(a_priori_width), .LLR_width(LLR_width), .frame_index(frame_index), .end_address(end_address))
dec_i(
clk, rst, dec_en, dec_done, SNR_select,
ctrl_mem0_bus_dec, addr_mem0_bus_dec, read_mem0_bus, write_mem0_bus_dec,
ctrl_mem1_bus_dec, addr_mem1_bus_dec, read_mem1_bus, write_mem1_bus_dec,
ctrl_mem2_bus_dec, addr_mem2_bus_dec, read_mem2_bus, write_mem2_bus_dec,
ctrl_mem3_bus, addr_mem3_bus, read_mem3_bus, write_mem3_bus,
ctrl_mem4_bus, addr_mem4_bus, read_mem4_bus, write_mem4_bus,
int_A_ctrl_dec, addr_A, rand_addr_A,
int_B_ctrl_dec, addr_B,
iteration, dec_mux2_1b, dec_debug);

Memory_Interface mem_i(
clk, zbt_clk,
ctrl_mem0_bus, addr_mem0_bus, read_mem0_bus, write_mem0_bus,
ctrl_mem1_bus, addr_mem1_bus, read_mem1_bus, write_mem1_bus,
ctrl_mem2_bus, addr_mem2_bus, read_mem2_bus, write_mem2_bus,
ctrl_mem3_bus, addr_mem3_bus, read_mem3_bus, write_mem3_bus,
ctrl_mem4_bus, addr_mem4_bus, read_mem4_bus, write_mem4_bus,

MEMORY_BANK0_CLK_P, MEMORY_BANK0_CLKEN_N, MEMORY_BANK0_WEN_N,

```

```

MEMORY_BANK0_WENA_N, MEMORY_BANK0_WENB_N, MEMORY_BANK0_WENC_N, MEMORY_BANK0_WEND_N,
MEMORY_BANK0_ADV_LD_N, MEMORY_BANK0_OEN_N, MEMORY_BANK0_CEN_N, MEMORY_BANK0_ADDR_P,
MEMORY_BANK0_DATA_A_P, MEMORY_BANK0_DATA_B_P, MEMORY_BANK0_DATA_C_P, MEMORY_BANK0_DATA_D_P,

MEMORY_BANK1_CLK_P, MEMORY_BANK1_CLKEN_N, MEMORY_BANK1_WEN_N,
MEMORY_BANK1_WENA_N, MEMORY_BANK1_WENB_N, MEMORY_BANK1_WENC_N, MEMORY_BANK1_WEND_N,
MEMORY_BANK1_ADV_LD_N, MEMORY_BANK1_OEN_N, MEMORY_BANK1_CEN_N, MEMORY_BANK1_ADDR_P,
MEMORY_BANK1_DATA_A_P, MEMORY_BANK1_DATA_B_P, MEMORY_BANK1_DATA_C_P, MEMORY_BANK1_DATA_D_P,

MEMORY_BANK2_CLK_P, MEMORY_BANK2_CLKEN_N, MEMORY_BANK2_WEN_N,
MEMORY_BANK2_WENA_N, MEMORY_BANK2_WENB_N, MEMORY_BANK2_WENC_N, MEMORY_BANK2_WEND_N,
MEMORY_BANK2_ADV_LD_N, MEMORY_BANK2_OEN_N, MEMORY_BANK2_CEN_N, MEMORY_BANK2_ADDR_P,
MEMORY_BANK2_DATA_A_P, MEMORY_BANK2_DATA_B_P, MEMORY_BANK2_DATA_C_P, MEMORY_BANK2_DATA_D_P,

MEMORY_BANK3_CLK_P, MEMORY_BANK3_CLKEN_N, MEMORY_BANK3_WEN_N,
MEMORY_BANK3_WENA_N, MEMORY_BANK3_WENB_N, MEMORY_BANK3_WENC_N, MEMORY_BANK3_WEND_N,
MEMORY_BANK3_ADV_LD_N, MEMORY_BANK3_OEN_N, MEMORY_BANK3_CEN_N, MEMORY_BANK3_ADDR_P,
MEMORY_BANK3_DATA_A_P, MEMORY_BANK3_DATA_B_P, MEMORY_BANK3_DATA_C_P, MEMORY_BANK3_DATA_D_P,

MEMORY_BANK4_CLK_P, MEMORY_BANK4_CLKEN_N, MEMORY_BANK4_WEN_N,
MEMORY_BANK4_WENA_N, MEMORY_BANK4_WENB_N, MEMORY_BANK4_WENC_N, MEMORY_BANK4_WEND_N,
MEMORY_BANK4_ADV_LD_N, MEMORY_BANK4_OEN_N, MEMORY_BANK4_CEN_N, MEMORY_BANK4_ADDR_P,
MEMORY_BANK4_DATA_A_P, MEMORY_BANK4_DATA_B_P, MEMORY_BANK4_DATA_C_P, MEMORY_BANK4_DATA_D_P,

mem_mux23_01b,
mem_debug
);

endmodule

```

A.2 Data Generator

```

module create_data(
clk, rst, en, done,

ctrl_mem0_bus,
addr_mem0_bus,
read_mem0_bus,
write_mem0_bus,

ctrl_mem1_bus,
addr_mem1_bus,
read_mem1_bus,
write_mem1_bus,

ctrl_mem2_bus,
addr_mem2_bus,
read_mem2_bus,
write_mem2_bus,

int_A_ctrl,
addr_A,
rand_addr_A,

int_B_ctrl,
addr_B,
rand_addr_B,

debug_port);

parameter frame_index = 19;
parameter end_address = -1;

```

```

localparam fi = frame_index - 1;

input clk, rst, en;
output done;

output [8:0] ctrl_mem0_bus;
output [18:0] addr_mem0_bus;
input [31:0] read_mem0_bus;
output [31:0] write_mem0_bus;

output [8:0] ctrl_mem1_bus;
output [18:0] addr_mem1_bus;
input [31:0] read_mem1_bus;
output [31:0] write_mem1_bus;

output [8:0] ctrl_mem2_bus;
output [18:0] addr_mem2_bus;
input [31:0] read_mem2_bus;
output [31:0] write_mem2_bus;

output [2:0] int_A_ctrl;
input [18:0] addr_A;
input [18:0] rand_addr_A;

output [2:0] int_B_ctrl;
input [18:0] addr_B;
input [18:0] rand_addr_B;

output [63:1] debug_port;

wire done;

wire [8:0] ctrl_mem0_bus;
wire [18:0] addr_mem0_bus;
wire [31:0] write_mem0_bus;

wire [8:0] ctrl_mem1_bus;
wire [18:0] addr_mem1_bus;
wire [31:0] write_mem1_bus;

wire [8:0] ctrl_mem2_bus;
wire [18:0] addr_mem2_bus;
wire [31:0] write_mem2_bus;

wire data_input;
wire sys1, par1, sys2, par2;
wire [3:0] sf_sys1, sf_par1, sf_par2;

wire enc_in;
wire [3:0] sf_par2_mem;
wire enc_in_pi;
wire [3:0] sf_sys_mem;
wire sys_mem, par1_mem, par2_mem;

wire [2:0] int_A_ctrl, int_B_ctrl;
wire int_A_mode, int_A_en, int_A_clr, int_B_mode, int_B_en, int_B_clr;

assign int_A_ctrl = {int_A_en, int_A_clr, int_A_mode};
assign int_B_ctrl = {int_B_en, int_B_clr, int_B_mode};

wire gen_en, enc_en, enc_term, map_en;

// Output enable and some inputs of tristate bus
wire oe_ctrl0, oe_ctrl1, oe_ctrl2, oe_a_mem0, oe_ra_mem0,

```

```

        oe_a_mem1, oe_ra_mem1, oe_a_mem2, oe_ra_mem2,
        oe_wr_mem0, oe_wr_mem1a, oe_wr_mem1b, oe_wr_mem2;

wire [8:0] ctrl_mem0_bus_i, ctrl_mem1_bus_i, ctrl_mem2_bus_i;

assign ctrl_mem0_bus = ctrl_mem0_bus_i;
assign ctrl_mem1_bus = ctrl_mem1_bus_i;
assign ctrl_mem2_bus = ctrl_mem2_bus_i;

assign addr_mem0_bus = (oe_a_mem0 & ~oe_ra_mem0)? addr_A : rand_addr_A;

assign addr_mem1_bus = (oe_a_mem1 & ~oe_ra_mem1)? addr_A : rand_addr_A;

assign addr_mem2_bus = (oe_a_mem2 & ~oe_ra_mem2)? addr_B : rand_addr_B;

assign enc_in = read_mem0_bus[16];
assign sf_par2_mem = read_mem0_bus[27:24];
assign enc_in_pi = read_mem1_bus[0];
assign sf_sys_mem = read_mem1_bus[7:4];
assign {sys_mem, par1_mem, par2_mem} = read_mem2_bus[26:24];

assign write_mem0_bus[31:16] = {4'h0, sf_par2, 7'h00, data_input};
assign write_mem1_bus[7:0] = (oe_wr_mem1a & ~oe_wr_mem1b)? {7'h00, data_input} : {sf_sys1, sf_par1};
assign write_mem2_bus[31:24] = {5'h00, sys1, par1, par2};
assign write_mem2_bus[7:0] = {sf_sys_mem, sf_par2_mem};

// DEBUG PORT
wire [63:1] debug_port;
wire [28:0] ctrl_debug;
wire [3:0] enc_debug;

assign debug_port[63] = done;
assign debug_port[62] = enc_in;
assign debug_port[61] = enc_in_pi;
assign debug_port[60] = data_input;
assign debug_port[59:57] = {sys_mem, par1_mem, par2_mem};
assign debug_port[56:49] = {sf_sys1, sf_par1};
assign debug_port[48:45] = enc_debug;
assign debug_port[44:42] = {sys1, par1, par2};
assign debug_port[41:34] = {sf_sys_mem, sf_par2_mem};
assign debug_port[33:5] = ctrl_debug;
assign debug_port[4] = enc_term;
assign debug_port[3] = en;
assign debug_port[2] = rst;
assign debug_port[1] = clk;
//debug_port[0] will be use for trigger_in

generator gen_i(clk, rst, gen_en, data_input);
encoder enc_i(clk, rst, enc_en, enc_term, enc_in, enc_in_pi, sys1, par1, sys2, par2, enc_debug);
mapper map_i(clk, rst, map_en, sys_mem, par1_mem, par2_mem, sf_sys1, sf_par1, sf_par2);

enc_control #(frame_index(10), end_address(-1))
    enc_ctrl_i (clk, rst, en, done, addr_A, addr_B,
        gen_en, enc_en, enc_term, map_en,
        int_A_mode, int_A_en, int_A_clr,
        int_B_mode, int_B_en, int_B_clr,
        oe_ctrl0, oe_ctrl1, oe_ctrl2,
        oe_a_mem0, oe_ra_mem0, oe_a_mem1, oe_ra_mem1, oe_a_mem2, oe_ra_mem2,
        oe_wr_mem0, oe_wr_mem1a, oe_wr_mem1b, oe_wr_mem2,
        ctrl_mem0_bus_i, ctrl_mem1_bus_i, ctrl_mem2_bus_i,
        ctrl_debug);

endmodule

```

A.3 Component Encoder

```
module encoder(clk, rst, en, terminate, ordered_data, interleaved_data, sys1, par1, sys2, par2, debug);
    input clk, rst, en, terminate, ordered_data, interleaved_data;
    output sys1, par1, sys2, par2;
    output [3:0] debug;

    wire rst, clk, en, terminate, ordered_data, interleaved_data;
    wire sys1, sys2, par1, par2;
    wire [3:0] debug;

    rsc rsc1(rst, clk, en, terminate, ordered_data, sys1, par1, debug[3:2]);
    rsc rsc2(rst, clk, en, terminate, interleaved_data, sys2, par2, debug[1:0]);
endmodule
```

A.4 RSC Encoder

```
module rsc(rst, clk, en, terminate, data_in, sys, par, shift_reg);
    input rst, clk, en, terminate, data_in;
    output sys, par;
    output [1:0] shift_reg;

    wire rst, clk, en, terminate, data_in, sys, par;
    wire x2, x10;
    reg [1:0] shift_reg;

    assign x10 = shift_reg[0] ^ shift_reg[1];
    assign sys = (terminate)? x10 : data_in;
    assign x2 = sys ^ x10;
    assign par = x2;

    always @(posedge clk or posedge rst)
    begin
        if (rst) shift_reg <= 0;
        else if (en) begin
            shift_reg <= {x2, shift_reg[1]};
        end
    end
endmodule
```

A.5 Address Generator and Interleaver

```
module interleaver (clk, rst, enable, clr, mode, addr, rand_addr);

    parameter size = 19;
    localparam s = size;

    input clk;
    input rst;
    input enable;
    input clr;
    input mode; // mode 0 is decrementer, mode 1 is interleaver
    output [18:0] addr;
    output [1:19] rand_addr;

    wire [18:0] addr;
```

```

        reg [s-1:0] addr_r;
        reg [1:19] rand_addr;
//      reg [1:19] rand_addr19;
//      reg [1:16] rand_addr16;
//      reg [1:15] rand_addr15;
//      reg [1:14] rand_addr14;
//      reg [1:13] rand_addr13;
//      reg [1:12] rand_addr12;
//      reg [1:11] rand_addr11;
//      reg [1:10] rand_addr10;

//      wire special_clr19, special_set19, magic_bit19;
//      wire special_clr16, special_set16, magic_bit16;
//      wire special_clr15, special_set15, magic_bit15;
//      wire special_clr14, special_set14, magic_bit14;
//      wire special_clr13, special_set13, magic_bit13;
//      wire special_clr12, special_set12, magic_bit12;
//      wire special_clr11, special_set11, magic_bit11;
//      wire special_clr10, special_set10, magic_bit10;

// Special set and clear to make the LFSR to include state zero in the state chain
//      assign special_clr19 = (rand_addr19 == 19'b100_0000_0000_0000_0000);
//      assign special_set19 = ~|rand_addr19;
//      assign magic_bit19 = (rand_addr19[1] & ~special_clr19) | special_set19;

//      assign special_clr16 = (rand_addr16 == 16'b1000_0000_0000_0000);
//      assign special_set16 = ~|rand_addr16;
//      assign magic_bit16 = (rand_addr16[1] & ~special_clr16) | special_set16;
//
//      assign special_clr15 = (rand_addr15 == 15'b100_0000_0000_0000);
//      assign special_set15 = ~|rand_addr15;
//      assign magic_bit15 = (rand_addr15[1] & ~special_clr15) | special_set15;
//
//      assign special_clr14 = (rand_addr14 == 14'b10_0000_0000_0000);
//      assign special_set14 = ~|rand_addr14;
//      assign magic_bit14 = (rand_addr14[1] & ~special_clr14) | special_set14;
//
//      assign special_clr13 = (rand_addr13 == 13'b1_0000_0000_0000);
//      assign special_set13 = ~|rand_addr13;
//      assign magic_bit13 = (rand_addr13[1] & ~special_clr13) | special_set13;
//
//      assign special_clr12 = (rand_addr12 == 12'b1000_0000_0000);
//      assign special_set12 = ~|rand_addr12;
//      assign magic_bit12 = (rand_addr12[1] & ~special_clr12) | special_set12;
//
//      assign special_clr11 = (rand_addr11 == 11'b100_0000_0000);
//      assign special_set11 = ~|rand_addr11;
//      assign magic_bit11 = (rand_addr11[1] & ~special_clr11) | special_set11;
//
//      assign special_clr10 = (rand_addr10 == 10'b10_0000_0000);
//      assign special_set10 = ~|rand_addr10;
//      assign magic_bit10 = (rand_addr10[1] & ~special_clr10) | special_set10;

        assign addr = {(19-s){1'b0}},addr_r;

// LUT for preventing termination bit from overwritten
always @(addr_r, rand_addr10) begin
        rand_addr = 0;
//      // rand_addr19
//      case (addr_r)
//          19'h00000: rand_addr = 19'h19a2a;
//          19'h19a2a: rand_addr = 19'h00000;
//          19'h7fffe: rand_addr = addr_r;
//          19'h7fff: rand_addr = addr_r;
//          default: rand_addr = rand_addr19;

```



```

//          endcase
//          // rand_addr16
//          case (addr_r)
//              16'h0000: rand_addr[4:19] = 16'h5248;
//              16'h5248: rand_addr[4:19] = 16'h0000;
//              16'hfff: rand_addr[4:19] = addr_r;
//              16'hfff: rand_addr[4:19] = addr_r;
//              default: rand_addr[4:19] = rand_addr16;
//          endcase
//          // rand_addr15
//          case (addr_r)
//              15'h0000: rand_addr[5:19] = 15'h0001;
//              15'h0001: rand_addr[5:19] = 15'h0000;
//              15'h7ffe: rand_addr[5:19] = addr_r;
//              15'h7fff: rand_addr[5:19] = addr_r;
//              default: rand_addr[5:19] = rand_addr15;
//          endcase
//          // rand_addr14
//          case (addr_r)
//              14'h0000: rand_addr[6:19] = 14'h069d;
//              14'h069d: rand_addr[6:19] = 14'h0000;
//              14'h3ffe: rand_addr[6:19] = addr_r;
//              14'h3fff: rand_addr[6:19] = addr_r;
//              default: rand_addr[6:19] = rand_addr14;
//          endcase
//          // rand_addr13
//          case (addr_r)
//              13'h0000: rand_addr[7:19] = 13'h0b3a;
//              13'h0b3a: rand_addr[7:19] = 13'h0000;
//              13'h1ffe: rand_addr[7:19] = addr_r;
//              13'h1fff: rand_addr[7:19] = addr_r;
//              default: rand_addr[7:19] = rand_addr13;
//          endcase
//          // rand_addr12
//          case (addr_r)
//              12'h000: rand_addr[8:19] = 12'h946;
//              12'h946: rand_addr[8:19] = 12'h000;
//              12'hffe: rand_addr[8:19] = addr_r;
//              12'hfff: rand_addr[8:19] = addr_r;
//              default: rand_addr[8:19] = rand_addr12;
//          endcase
//          // rand_addr11
//          case (addr_r)
//              11'h000: rand_addr[9:19] = 11'h229;
//              11'h229: rand_addr[9:19] = 11'h000;
//              11'h7fe: rand_addr[9:19] = addr_r;
//              11'h7ff: rand_addr[9:19] = addr_r;
//              default: rand_addr[9:19] = rand_addr11;
//          endcase
//          // rand_addr10
//          case (addr_r)
//              10'h000: rand_addr[10:19] = 10'h2bd;
//              10'h2bd: rand_addr[10:19] = 10'h000;
//              10'h3fe: rand_addr[10:19] = addr_r;
//              10'h3ff: rand_addr[10:19] = addr_r;
//              default: rand_addr[10:19] = rand_addr10;
//          endcase
//          end
//          always @(posedge clk or posedge rst) begin
//              if (rst) begin
//                  addr_r <= 0;
//                  rand_addr19 <= -1;
//                  rand_addr16 <= -1;
//                  rand_addr15 <= -1;
//                  rand_addr14 <= -1;

```

```

//          rand_addr13 <= -1;
//          rand_addr12 <= -1;
//          rand_addr11 <= -1;
//          rand_addr10 <= -1;

end
else begin

    if (clr) begin
        addr_r <= 0;
        rand_addr19 <= -1;
        rand_addr16 <= -1;
        rand_addr15 <= -1;
        rand_addr14 <= -1;
        rand_addr13 <= -1;
        rand_addr12 <= -1;
        rand_addr11 <= -1;
        rand_addr10 <= -1;
    end
    else if (enable) begin
        if (mode == 1'b0) addr_r <= addr_r - 1;
        else begin
            addr_r <= addr_r + 1;
        end
        // for 2^19 frame size[19,14,13,12]
        //          rand_addr19 <= {rand_addr19[19],magic_bit19,rand_addr19[2:11],rand_addr19[12]^rand_addr19[19],
        //          rand_addr19[13]^rand_addr19[19],rand_addr19[14]^rand_addr19[19],rand_addr19[15:18]};
        //
        // for 2^16 frame size[16,13,12,11]
        //          rand_addr16 <= {rand_addr16[16],magic_bit16,rand_addr16[2:10],rand_addr16[11]^rand_addr16[16],
        //          rand_addr16[12]^rand_addr16[16],rand_addr16[13]^rand_addr16[16],rand_addr16[14:15]};
        //
        // for 2^15 frame size[15,14]
        //          rand_addr15 <= {rand_addr15[15],magic_bit15,rand_addr15[2:13],rand_addr15[14]^rand_addr15[15]};
        //
        // for 2^14 frame size[14, 11, 10, 9]
        //          rand_addr14 <= {rand_addr14[14],magic_bit14,rand_addr14[2:8],rand_addr14[9]^rand_addr14[14],
        //          rand_addr14[10]^rand_addr14[14],rand_addr14[11]^rand_addr14[14],rand_addr14[12:13]};
        //
        // for 2^13 frame size[13, 8, 7, 6]
        //          rand_addr13 <= {rand_addr13[13],magic_bit13,rand_addr13[2:5],rand_addr13[6]^rand_addr13[13],
        //          rand_addr13[7]^rand_addr13[13],rand_addr13[8]^rand_addr13[13],rand_addr13[9:12]};
        //
        // for 2^12 frame size[12, 10, 9, 3]
        //          rand_addr12 <= {rand_addr12[12],magic_bit12,rand_addr12[2],rand_addr12[3]^rand_addr12[12],rand_addr12[4:8],
        //          rand_addr12[9]^rand_addr12[12],rand_addr12[10]^rand_addr12[12],rand_addr12[11]};
        //
        // for 2^11 frame size[11, 9]
        //          rand_addr11 <= {rand_addr11[11],magic_bit11,rand_addr11[2:8],rand_addr11[9]^rand_addr11[11],rand_addr11[10]};
        //
        // for 2^10 frame size[10, 7]
        rand_addr10 <= {rand_addr10[10],magic_bit10,rand_addr10[2:6],rand_addr10[7]^rand_addr10[10],rand_addr10[8:9]};
    end
end

end
endmodule

```

A.6 Turbo Decoder

```
module decoder(
  clk, rst, en, done,
  SNR_select,

  ctrl_mem0_bus,
  addr_mem0_bus,
  read_mem0_bus,
  write_mem0_bus,

  ctrl_mem1_bus,
  addr_mem1_bus,
  read_mem1_bus,
  write_mem1_bus,

  ctrl_mem2_bus,
  addr_mem2_bus,
  read_mem2_bus,
  write_mem2_bus,

  ctrl_mem3_bus,
  addr_mem3_bus,
  read_mem3_bus,
  write_mem3_bus,

  ctrl_mem4_bus,
  addr_mem4_bus,
  read_mem4_bus,
  write_mem4_bus,

  int_A_ctrl,
  addr_A,
  rand_addr_A,

  int_B_ctrl,
  addr_B,

  iteration,

  dec_mux2_1b,
  debug_port);

  parameter a_priori_width = 8;
  parameter LLR_width = 16;

  parameter frame_index = 19;
  parameter end_address = -1;

  localparam fi = frame_index - 1;
  localparam w1 = a_priori_width - 1;
  localparam w2 = LLR_width - 1;

  input clk, rst, en, SNR_select;
  output done;

  output [8:0] ctrl_mem0_bus;
  output [18:0] addr_mem0_bus;
  input [31:0] read_mem0_bus;
  output [31:0] write_mem0_bus;

  output [8:0] ctrl_mem1_bus;
  output [18:0] addr_mem1_bus;
  input [31:0] read_mem1_bus;
```

```

output [31:0] write_mem1_bus;

output [8:0] ctrl_mem2_bus;
output [18:0] addr_mem2_bus;
input [31:0] read_mem2_bus;
output [31:0] write_mem2_bus;

output [8:0] ctrl_mem3_bus;
output [18:0] addr_mem3_bus;
input [31:0] read_mem3_bus;
output [31:0] write_mem3_bus;

output [8:0] ctrl_mem4_bus;
output [18:0] addr_mem4_bus;
input [31:0] read_mem4_bus;
output [31:0] write_mem4_bus;

output [2:0] int_A_ctrl;
input [18:0] addr_A;
input [18:0] rand_addr_A;

output [2:0] int_B_ctrl;
input [18:0] addr_B;

input [3:0] iteration;

input dec_mux2_1b;
output [158:0] debug_port;

wire init, done;

wire [8:0] ctrl_mem0_bus;
wire [18:0] addr_mem0_bus;
wire [31:0] write_mem0_bus;
wire [8:0] ctrl_mem1_bus;
reg [18:0] addr_mem1_bus;
wire [31:0] write_mem1_bus;
wire [8:0] ctrl_mem2_bus;
reg [18:0] addr_mem2_bus;
wire [31:0] write_mem2_bus;
wire [8:0] ctrl_mem3_bus;
wire [18:0] addr_mem3_bus;
wire [31:0] write_mem3_bus;
wire [8:0] ctrl_mem4_bus;
wire [18:0] addr_mem4_bus;
wire [31:0] write_mem4_bus;

wire [18:0] de_int_address;
wire [15:0] de_int_addr_low;
wire [2:0] de_int_addr_high;

// Pipeline Signals
reg [2:0] de_int_addr_high1;
reg [2:0] de_int_addr_high2;
reg [2:0] de_int_addr_high3;
reg [2:0] de_int_addr_high4;
reg [2:0] de_int_addr_high5;
reg [2:0] de_int_addr_high6;
reg [2:0] de_int_addr_high7;

wire [w1:0] a_priori1, a_priori2;
wire [3:0] sys1, par1, sys2, par2;
wire [w2:0] beta_a1, beta_b1, beta_c1, beta_d1;
wire [w2:0] beta_a1_out, beta_b1_out, beta_c1_out, beta_d1_out;
wire [w2:0] beta_a2, beta_b2, beta_c2, beta_d2;

```

```

wire [w2:0] beta_a2_out, beta_b2_out, beta_c2_out, beta_d2_out;

wire dec1_valid, dec2_valid;
wire [w1:0] a_posteriori1, a_posteriori2;
wire [w2:0] LLR1, LLR2;

wire [1:0] dec1_mode;
wire [1:0] dec2_mode;

wire [2:0] int_A_ctrl, int_B_ctrl;
wire int_A_mode, int_A_en, int_A_clr, int_B_mode, int_B_en, int_B_clr;

assign int_A_ctrl = {int_A_en, int_A_clr, int_A_mode};
assign int_B_ctrl = {int_B_en, int_B_clr, int_B_mode};

// Output enable and some inputs of tristate bus
wire oe_ctrl0, oe_ctrl1, oe_ctrl2, oe_ctrl3, oe_ctrl4;
wire oe_a_mem0, oe_ra_mem0, oe_a_mem1, oe_b_mem1, oe_int_mem1;
wire oe_a_mem2, oe_ra_mem2, oe_b_mem2, oe_mem34;
wire oe_wr_mem0, oe_wr_mem1, oe_wr_mem2, oe_beta_1, oe_beta_2;
wire [8:0] ctrl_mem0_bus_i, ctrl_mem1_bus_i, ctrl_mem2_bus_i;
wire [8:0] ctrl_mem3_bus_i, ctrl_mem4_bus_i;

wire [158:0] debug_port;
wire [31:0] ctrl_debug;
wire [126:0] dec1_debug, dec2_debug;

assign debug_port[158:32] = (dec_mux2_1b)? dec2_debug : dec1_debug;
assign debug_port[31:0] = ctrl_debug;

assign de_int_address[18:16] = (dec2_mode[0])? de_int_addr_high7 : de_int_addr_high6;
assign de_int_address[15:0] = de_int_addr_low;

assign ctrl_mem0_bus = ctrl_mem0_bus_i;
assign ctrl_mem1_bus = ctrl_mem1_bus_i;
assign ctrl_mem2_bus = ctrl_mem2_bus_i;
assign ctrl_mem3_bus = ctrl_mem3_bus_i;
assign ctrl_mem4_bus = ctrl_mem4_bus_i;

assign addr_mem0_bus = (oe_a_mem0 & ~oe_ra_mem0)? addr_A : rand_addr_A;

always@(oe_a_mem1, oe_b_mem1, oe_int_mem1, addr_A, addr_B, de_int_address) begin
    casex({oe_a_mem1, oe_b_mem1, oe_int_mem1})
        3'b1xx: addr_mem1_bus = addr_A;
        3'bx1x: addr_mem1_bus = addr_B;
        3'bxx1: addr_mem1_bus = de_int_address;
        default: addr_mem1_bus = 0;
    endcase
end

always@(oe_a_mem2, oe_ra_mem2, oe_b_mem2, addr_A, rand_addr_A, addr_B) begin
    casex({oe_a_mem2, oe_ra_mem2, oe_b_mem2})
        3'b1xx: addr_mem2_bus = addr_A;
        3'bx1x: addr_mem2_bus = rand_addr_A;
        3'bxx1: addr_mem2_bus = addr_B;
        default: addr_mem2_bus = 0;
    endcase
end

assign addr_mem3_bus = addr_B;
assign addr_mem4_bus = addr_B;

assign de_int_addr_low = read_mem0_bus[15:0];
assign a_priori1 = (init)? 0 : read_mem1_bus[15:8];
assign sys1 = read_mem1_bus[7:4];

```

```

assign par1 = read_mem1_bus[3:0];
assign de_int_addr_high = read_mem2_bus[18:16];
assign {a_priori2, sys2, par2} = read_mem2_bus[15:0];
assign {beta_a1, beta_c1} = read_mem3_bus;
assign {beta_a2, beta_c2} = read_mem3_bus;
assign {beta_b1, beta_d1} = read_mem4_bus;
assign {beta_b2, beta_d2} = read_mem4_bus;

assign write_mem0_bus[15:0] = addr_A[15:0];
assign write_mem1_bus[31:8] = {LLR2,a_posteriori2};
assign write_mem2_bus[18:8] = {addr_A[18:16],a_posteriori1};
assign write_mem3_bus = (oe_beta_1 & ~oe_beta_2)? {beta_a1_out, beta_c1_out} : {beta_a2_out, beta_c2_out};
assign write_mem4_bus = (oe_beta_1 & ~oe_beta_2)? {beta_b1_out, beta_d1_out} : {beta_b2_out, beta_d2_out};

// Pipeline for top three bit of de_int_address when read
always @ (posedge clk or posedge rst) begin
    if (rst) begin
        de_int_addr_high1 <= 0;
        de_int_addr_high2 <= 0;
        de_int_addr_high3 <= 0;
        de_int_addr_high4 <= 0;
        de_int_addr_high5 <= 0;
        de_int_addr_high6 <= 0;
        de_int_addr_high7 <= 0;
    end
    else if (dec2_mode[1]) begin
        de_int_addr_high1 <= de_int_addr_high;
        de_int_addr_high2 <= de_int_addr_high1;
        de_int_addr_high3 <= de_int_addr_high2;
        de_int_addr_high4 <= de_int_addr_high3;
        de_int_addr_high5 <= de_int_addr_high4;
        de_int_addr_high6 <= de_int_addr_high5;
        de_int_addr_high7 <= de_int_addr_high6;
    end
end

dec_control #(.frame_index(frame_index),.end_address(end_address))
    dec_ctrl_i(clk,rst,en,init,done,dec1_valid,dec2_valid,
        addr_A[fi:0],addr_B[fi:0],dec1_mode,dec2_mode,
        int_A_mode,int_A_en,int_A_clr,
        int_B_mode,int_B_en,int_B_clr,
        oe_ctrl0, oe_ctrl1, oe_ctrl2, oe_ctrl3, oe_ctrl4,
        oe_a_mem0, oe_ra_mem0, oe_a_mem1, oe_b_mem1,
        oe_int_mem1, oe_a_mem2, oe_ra_mem2, oe_b_mem2,oe_mem34,
        oe_wr_mem0, oe_wr_mem1, oe_wr_mem2, oe_beta_1, oe_beta_2,
        ctrl_mem0_bus_i,ctrl_mem1_bus_i,ctrl_mem2_bus_i,
        ctrl_mem3_bus_i,ctrl_mem4_bus_i,iteration,ctrl_debug);

log_MAP #(.trellis_term(1'b1))
    decoder_1(clk, rst, dec1_mode, a_priori1, sys1, par1,
        beta_a1, beta_b1, beta_c1, beta_d1, SNR_select,
        beta_a1_out, beta_b1_out, beta_c1_out, beta_d1_out,
        dec1_valid, a_posteriori1, LLR1, dec1_debug);

log_MAP #(.trellis_term(1'b1))
    decoder_2(clk, rst, dec2_mode, a_priori2, sys2, par2,
        beta_a2, beta_b2, beta_c2, beta_d2, SNR_select,
        beta_a2_out, beta_b2_out, beta_c2_out, beta_d2_out,
        dec2_valid, a_posteriori2, LLR2, dec2_debug);

endmodule

```

A.7 Log-MAP Decoder

```
`timescale 1 ns / 100 ps // time unit = 1 ns; precision = 1/10 ns;

module log_MAP(clk, rst, mode, a_priori, sys, par, beta_a, beta_b, beta_c, beta_d,
              SNR_select, beta_a_out, beta_b_out, beta_c_out, beta_d_out,
              data_valid, a_posteriori, LLR, debug);

    parameter a_priori_width = 8;
    parameter LLR_width = 16;
    parameter trellis_term = 1'b1; // assume trellis are terminated

    localparam w1 = a_priori_width - 1;
    localparam w2 = LLR_width - 1;
    localparam t = trellis_term;

    localparam [1:0] Reset_ST = 2'b00;
    localparam [1:0] Beta_ST = 2'b01;
    localparam [1:0] Alpha_ST = 2'b11;
    localparam [1:0] LLR_ST = 2'b10;

    reg [6:0] lfsr_counter; // count up to 127 cycles for window size.

    input clk, rst, SNR_select;
    input [1:0] mode; // mode 00:Idle 01:Beta 10:Alpha 11:Last run
    input [w1:0] a_priori;
    input [3:0] sys, par;
    input [w2:0] beta_a, beta_b, beta_c, beta_d;
    output [w2:0] beta_a_out, beta_b_out, beta_c_out, beta_d_out;
    output data_valid;
    output [w1:0] a_posteriori;
    output [w2:0] LLR;
    output [126:0] debug;

    wire [w2:0] beta_a_out, beta_b_out, beta_c_out, beta_d_out;
    reg data_valid;
    wire [w1:0] a_posteriori;
    wire [w2:0] LLR;

    reg bm_en, pm_B_set, pm_A_set;
    wire bm_en_out, pm_B_en, pm_A_en;
    wire pm_B_cal_valid, pm_B_init_valid;
    wire LLR_busy;

    wire [w1:0] r_aa, r_ba, r_cb, r_db, r_ac, r_bc, r_cd, r_dd;
    wire [w2:0] new_pm_B_a, new_pm_B_b, new_pm_B_c, new_pm_B_d, pm_B_a_init, pm_B_b_init, pm_B_c_init, pm_B_d_init;
    wire [w2:0] ps_B_a_up1, ps_B_a_dn1, ps_B_b_up1, ps_B_b_dn1, ps_B_c_up1, ps_B_c_dn1, ps_B_d_up1, ps_B_d_dn1;
    wire [w2:0] ps_B_a_up2, ps_B_a_dn2, ps_B_b_up2, ps_B_b_dn2, ps_B_c_up2, ps_B_c_dn2, ps_B_d_up2, ps_B_d_dn2;

    wire pm_A_cal_valid, pm_A_init_valid;
    wire pm_A_cal_busy, pm_A_init_busy;
    wire [w2:0] new_pm_A_a, new_pm_A_b, new_pm_A_c, new_pm_A_d, pm_A_a_init, pm_A_b_init, pm_A_c_init, pm_A_d_init;
    wire [w2:0] ps_A_a_up1, ps_A_a_dn1, ps_A_b_up1, ps_A_b_dn1, ps_A_c_up1, ps_A_c_dn1, ps_A_d_up1, ps_A_d_dn1;
    wire [w2:0] ps_A_a_up2, ps_A_a_dn2, ps_A_b_up2, ps_A_b_dn2, ps_A_c_up2, ps_A_c_dn2, ps_A_d_up2, ps_A_d_dn2;

    reg [1:0] state, next_state;
    reg lfsr_clr, lfsr_en;

    reg [w1:0] ap_delay1, ap_delay2, ap_delay3, ap_delay4, ap_delay5, ap_delay6;
    reg [3:0] sys_delay1, sys_delay2, sys_delay3, sys_delay4, sys_delay5, sys_delay6;
    reg [w2:0] beta_a1, beta_b1, beta_c1, beta_d1, beta_a2, beta_b2, beta_c2, beta_d2;
    reg [1:0] mode_delay;

    wire LLR_valid, a_posteriori_valid, overflow;
```

```

assign pm_B_en = bm_en_out & ~mode_delay[1] & mode_delay[0];
assign pm_A_en = bm_en_out & mode_delay[1];

// Attach output beta and valid signals
assign beta_a_out = new_pm_B_a;
assign beta_b_out = new_pm_B_b;
assign beta_c_out = new_pm_B_c;
assign beta_d_out = new_pm_B_d;
assign beta_valid = pm_B_cal_valid;

// Attach debug port
wire [126:0] debug;
assign debug[126] = 1'b0; // A1_max_valid;
assign debug[125:124] = state;
assign debug[123:122] = mode;
assign debug[121] = data_valid;
assign debug[120] = pm_B_set;
assign debug[119:116] = LLR[15:12];
assign debug[115:112] = sys;
assign debug[111:104] = a_posteriori;
assign debug[103:88] = new_pm_A_a;
assign debug[87:72] = new_pm_A_b;
assign debug[71:56] = pm_B_a_init;
assign debug[55:40] = new_pm_B_a;
assign debug[39:24] = new_pm_B_b;
assign debug[23:16] = r_ac;
assign debug[15:8] = r_ba;
assign debug[7:0] = r_aa;

bm_bundler_cal(clk, rst, bm_en, bm_en_out, a_priori, sys, par, SNR_select,
               r_aa, r_ba, r_cb, r_db, r_ac, r_bc, r_cd, r_dd);

pm_bundle #(.branch_met_width(w1+1), .path_met_width(w2+1), .trellis_term(t))
    B_cal(clk, rst, pm_B_en, SNR_select, pm_B_cal_valid, pm_B_set,
          pm_B_a_init, pm_B_c_init, new_pm_B_a, new_pm_B_c, r_aa, r_ac, ps_B_a_up1, ps_B_a_dn1, new_pm_B_a,
          pm_B_a_init, pm_B_c_init, new_pm_B_a, new_pm_B_c, r_ba, r_bc, ps_B_b_up1, ps_B_b_dn1, new_pm_B_b,
          pm_B_b_init, pm_B_d_init, new_pm_B_b, new_pm_B_d, r_cb, r_cd, ps_B_c_up1, ps_B_c_dn1, new_pm_B_c,
          pm_B_b_init, pm_B_d_init, new_pm_B_b, new_pm_B_d, r_db, r_dd, ps_B_d_up1, ps_B_d_dn1, new_pm_B_d);

pm_bundle #(.branch_met_width(w1+1), .path_met_width(w2+1), .trellis_term(t))
    B_init(clk, rst, pm_B_en, SNR_select, pm_B_init_valid, pm_B_set,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_B_a_init, pm_B_c_init, r_aa, r_ac, ps_B_a_up2, ps_B_a_dn2, pm_B_a_init,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_B_a_init, pm_B_c_init, r_ba, r_bc, ps_B_b_up2, ps_B_b_dn2, pm_B_b_init,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_B_b_init, pm_B_d_init, r_cb, r_cd, ps_B_c_up2, ps_B_c_dn2, pm_B_c_init,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_B_b_init, pm_B_d_init, r_db, r_dd, ps_B_d_up2, ps_B_d_dn2, pm_B_d_init);

pm_bundle #(.branch_met_width(w1+1), .path_met_width(w2+1), .trellis_term(1'b1))
    A_cal(clk, rst, pm_A_en, SNR_select, pm_A_cal_valid, pm_A_set,
          pm_A_a_init, pm_A_b_init, new_pm_A_a, new_pm_A_b, r_aa, r_ba, ps_A_a_up1, ps_A_a_dn1, new_pm_A_a,
          pm_A_c_init, pm_A_d_init, new_pm_A_c, new_pm_A_d, r_cb, r_db, ps_A_b_up1, ps_A_b_dn1, new_pm_A_b,
          pm_A_a_init, pm_A_b_init, new_pm_A_a, new_pm_A_b, r_ac, r_bc, ps_A_c_up1, ps_A_c_dn1, new_pm_A_c,
          pm_A_c_init, pm_A_d_init, new_pm_A_c, new_pm_A_d, r_cd, r_dd, ps_A_d_up1, ps_A_d_dn1, new_pm_A_d);

pm_bundle #(.branch_met_width(w1+1), .path_met_width(w2+1), .trellis_term(1'b1))
    A_init(clk, rst, pm_A_en, SNR_select, pm_A_init_valid, pm_A_set,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_A_a_init, pm_A_b_init, r_aa, r_ba, ps_A_a_up2, ps_A_a_dn2, pm_A_a_init,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_A_c_init, pm_A_d_init, r_cb, r_db, ps_A_b_up2, ps_A_b_dn2, pm_A_b_init,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_A_a_init, pm_A_b_init, r_ac, r_bc, ps_A_c_up2, ps_A_c_dn2, pm_A_c_init,
           {(w2+1){1'b0}}, {(w2+1){1'b0}}, pm_A_c_init, pm_A_d_init, r_cd, r_dd, ps_A_d_up2, ps_A_d_dn2, pm_A_d_init);

aposteriori_cal #(.a_priori_width(w1+1), .path_met_width(w2+1))
    LLR_cal(clk, rst, pm_A_cal_valid, LLR_busy, sys_delay6, ap_delay6,
            beta_a2, ps_A_a_up1, ps_A_a_dn1, beta_b2, ps_A_b_dn1, ps_A_b_up1,
            beta_c2, ps_A_c_dn1, ps_A_c_up1, beta_d2, ps_A_d_up1, ps_A_d_dn1,
            LLR, a_posteriori, LLR_valid, a_posteriori_valid, overflow);

```



```

//delays sys, apriori and beta for a-posteriori calculation
always @ (posedge clk or posedge rst) begin
    if(rst) begin
        mode_delay <= 0;
        ap_delay1 <= 0;
        ap_delay2 <= 0;
        ap_delay3 <= 0;
        ap_delay4 <= 0;
        ap_delay5 <= 0;
        ap_delay6 <= 0;
        sys_delay1 <= 0;
        sys_delay2 <= 0;
        sys_delay3 <= 0;
        sys_delay4 <= 0;
        sys_delay5 <= 0;
        sys_delay6 <= 0;
        beta_a1 <= 0;
        beta_b1 <= 0;
        beta_c1 <= 0;
        beta_d1 <= 0;
        beta_a2 <= 0;
        beta_b2 <= 0;
        beta_c2 <= 0;
        beta_d2 <= 0;
    end
    else begin
        mode_delay <= mode;
        if(mode[1] || LLR_busy) begin
            ap_delay1 <= a_priori;
            ap_delay2 <= ap_delay1;
            ap_delay3 <= ap_delay2;
            ap_delay4 <= ap_delay3;
            ap_delay5 <= ap_delay4;
            ap_delay6 <= ap_delay5;
            sys_delay1 <= sys;
            sys_delay2 <= sys_delay1;
            sys_delay3 <= sys_delay2;
            sys_delay4 <= sys_delay3;
            sys_delay5 <= sys_delay4;
            sys_delay6 <= sys_delay5;
            beta_a1 <= beta_a;
            beta_b1 <= beta_b;
            beta_c1 <= beta_c;
            beta_d1 <= beta_d;
            beta_a2 <= beta_a1;
            beta_b2 <= beta_b1;
            beta_c2 <= beta_c1;
            beta_d2 <= beta_d1;
        end
    end
end

always @ (posedge clk or posedge rst) begin
    if(rst) lfsr_counter <= 1;
    else if(lfsr_clr) lfsr_counter <= 1;
    else if(lfsr_en) begin
        lfsr_counter <= lfsr_counter >> 1;
        lfsr_counter[6] <= lfsr_counter[1] ^ lfsr_counter[0];
    end
end

always @ (posedge clk or posedge rst) begin
    if(rst) state <= Reset_ST;
    else state <= next_state;
end

```

```

always @ (state, mode, LLR_busy) begin
    case (state)
        Reset_ST: begin
            casez (mode) 2'b01: next_state = Beta_ST ;
                2'b10: next_state = Alpha_ST;
                2'b11: next_state = LLR_ST;
                default: next_state = Reset_ST;
            endcase
        end
        Beta_ST: begin
            casez (mode) 2'b00: if next_state = Reset_ST ;
                2'b10: if next_state = Alpha_ST;
                2'b11: if next_state = LLR_ST;
                default: next_state = Beta_ST;
            endcase
        end
        Alpha_ST: if(~mode[1] & ~LLR_busy) next_state = Reset_ST;
        LLR_ST: if(~mode[1] & ~LLR_busy) next_state = Reset_ST;
    endcase
end
always @ (state, mode, lfsr_counter, pm_B_cal_valid, a_posteriori_valid, LLR_valid) begin
    bm_en = 1'b0;
    lfsr_en = 1'b0;
    lfsr_clr = 1'b0;
    pm_B_set = 1'b0;
    pm_A_set = 1'b0;
    data_valid = 1'b0;
    case (state)
        Reset_ST: if(mode[0] | mode[1]) begin
            bm_en = 1'b1;
            lfsr_en = 1'b1;
        end
        Beta_ST: begin
            if(mode[0] | mode[1]) begin
                bm_en = 1'b1;
                lfsr_en = 1'b1;
            end
            if (pm_B_cal_valid) data_valid = 1'b1; // This allow the B init be written into the memory
            if (lfsr_counter==7'b00000011) begin
                pm_B_set = 1'b1;
                lfsr_clr = 1'b1;
            end
        end
        Alpha_ST: begin
            if(mode[0] | mode[1]) begin
                bm_en = 1'b1;
                lfsr_en = 1'b1;
            end
            if (a_posteriori_valid) data_valid = 1'b1;
            if (lfsr_counter==7'b00000011) begin
                pm_A_set = 1'b1;
                lfsr_clr = 1'b1;
            end
        end
        LLR_ST: begin
            if(mode[0] | mode[1]) begin
                bm_en = 1'b1;
                lfsr_en = 1'b1;
            end
            if (LLR_valid) data_valid = 1'b1;
            if (lfsr_counter==7'b00000011) begin
                pm_A_set = 1'b1;
                lfsr_clr = 1'b1;
            end
        end
    endcase
end
endmodule

```

A.8 Branch Metrics Calculation Units

```

`timescale 1 ns / 100 ps // time unit = 1ns; precision = 1/10 ns;

module bm_bundle(clk, rst, en, en_out, a_priori, sys, par, SNR_select, r_aa, r_ba, r_cb, r_db, r_ac, r_bc, r_cd, r_dd);
    parameter apriori_width = 8;
    localparam w = apriori_width - 1;

    input clk, rst, en, SNR_select;
    input [w:0] a_priori;
    input [3:0] sys, par;
    output en_out;
    output [w:0] r_aa, r_ba, r_cb, r_db, r_ac, r_bc, r_cd, r_dd;

    wire clk, rst, en, SNR_select; // 0 Low SNR, 1 Mid/High SNR
    wire [w:0] a_priori, r_aa, r_ba, r_cb, r_db, r_ac, r_bc, r_cd, r_dd;
    wire [3:0] sys, par;
    reg en_out;

    //Expect data setting
    //exp_data[2:0]
    //bit2: Expect EncoderInput
    //      Input 0 (default) : Expect Data = +1
    //      Input 1          : Expect Data = -1
    //bit1: Expect Systematic
    //      Input 0 (default) : Expect Sys = +1
    //      Input 1          : Expect Sys = -1
    //bit0: Expect Parity
    //      Input 0 (default) : Expect Par = +1
    //      Input 1          : Expect Par = -1
    branch_met_cal #(apriori_width(w+1), exp_data(3'b000)) r_aa_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_aa);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b110)) r_ba_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_ba);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b110)) r_cb_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_cb);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b000)) r_db_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_db);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b111)) r_ac_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_ac);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b001)) r_bc_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_bc);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b001)) r_cd_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_cd);
    branch_met_cal #(apriori_width(w+1), exp_data(3'b111)) r_dd_cal(clk, rst, en, a_priori, sys, par, SNR_select, r_dd);

    // daisy chain of enable
    always @ (posedge clk or posedge rst) begin
        if (rst) en_out <= 0;
        else en_out <= en;
    end
endmodule

module branch_met_cal(clk, rst, en, a_priori, sys, par, SNR_select, branch_met);
    parameter apriori_width = 8;
    //Expect data setting
    //exp_data[2:0]
    //bit2: Expect EncoderInput Sign
    //      Input 0 (default) : Expect Data = +1
    //      Input 1          : Expect Data = -1
    //bit1: Expect Systematic Sign
    //      Input 0 (default) : Expect Sys = +1
    //      Input 1          : Expect Sys = -1
    //bit0: Expect Parity Sign
    //      Input 0 (default) : Expect Par = +1
    //      Input 1          : Expect Par = -1
    parameter [2:0] exp_data = 3'b111;
    localparam w = apriori_width - 1;

    input clk, rst, en, SNR_select;
    input [w:0] a_priori;

```

```

input [3:0] sys, par;
output [w:0] branch_met;

wire clk, rst, en, SNR_select; // 0 Low SNR, 1 Mid/High SNR
wire [w:0] a_priori;
wire [3:0] sys, par;
reg [w:0] branch_met;

wire [3:0] sys_comp_out, par_comp_out;
wire [w:0] apriori_comp_out;
wire [3:0] sys_reg_shift, par_reg_shift;
wire [1:0] comp_adj; //2's complement adjustment
reg [w+1:0] branch_sum;

assign apriori_comp_out = {(w+1){exp_data[2]}} ^ a_priori;
assign sys_comp_out = {4{exp_data[1]}} ^ sys;
assign par_comp_out = {4{exp_data[0]}} ^ par;

always @ (posedge clk or posedge rst)
begin
    if (rst)begin
        branch_met <= 0;
    end
    else if (en)begin
        branch_met <= branch_sum[(w+1):1]; // branch_met = branch_sum / 2
    end
end

//channel sum decimal precision select base on SNR setting
assign sys_reg_shift = (SNR_select)? {sys_comp_out[3],sys_comp_out[3:1]} : sys_comp_out;
assign par_reg_shift = (SNR_select)? {par_comp_out[3],par_comp_out[3:1]} : par_comp_out;

//2's complement sum adjustment
assign comp_adj[1] = exp_data[2] & (exp_data[1] | exp_data[0]);
assign comp_adj[0] = ^exp_data;

//branch sum adder
always @ (apriori_comp_out, sys_reg_shift, par_reg_shift, comp_adj) branch_sum = {{(w-2){sys_reg_shift[3]}},sys_reg_shift}
+ {{(w-2){par_reg_shift[3]}},par_reg_shift}
+ {apriori_comp_out[w],apriori_comp_out}
+ comp_adj;

endmodule

```

A.9 Path Metrics Calculation Units

`timescale 1 ns /100 ps // time unit = 1ns; precision = 1/10 ns;

```

module pm_bundle (clk, rst, en, SNR_select, valid, pm_set,
    pm_a_init_up, pm_a_init_dn, pm_a_up, pm_a_dn, r_a_up, r_a_dn, ps_a_up, ps_a_dn, new_pm_a,
    pm_b_init_up, pm_b_init_dn, pm_b_up, pm_b_dn, r_b_up, r_b_dn, ps_b_up, ps_b_dn, new_pm_b,
    pm_c_init_up, pm_c_init_dn, pm_c_up, pm_c_dn, r_c_up, r_c_dn, ps_c_up, ps_c_dn, new_pm_c,
    pm_d_init_up, pm_d_init_dn, pm_d_up, pm_d_dn, r_d_up, r_d_dn, ps_d_up, ps_d_dn, new_pm_d);

    parameter branch_met_width = 8;
    parameter path_met_width = 16;
    parameter trellis_term = 1'b1; // assume trellis are terminated

    localparam w1 = branch_met_width - 1;
    localparam w2 = path_met_width - 1;
    localparam t = trellis_term;

    input clk, rst, en, SNR_select, pm_set;

```

```

input [w2:0] pm_a_init_up, pm_a_init_dn, pm_a_up, pm_a_dn,
           pm_b_init_up, pm_b_init_dn, pm_b_up, pm_b_dn,
           pm_c_init_up, pm_c_init_dn, pm_c_up, pm_c_dn,
           pm_d_init_up, pm_d_init_dn, pm_d_up, pm_d_dn;
input [w1:0] r_a_up, r_a_dn, r_b_up, r_b_dn, r_c_up, r_c_dn, r_d_up, r_d_dn;

output [w2:0] ps_a_up, ps_a_dn, ps_b_up, ps_b_dn, ps_c_up, ps_c_dn, ps_d_up, ps_d_dn;
output [w2:0] new_pm_a, new_pm_b, new_pm_c, new_pm_d;
output valid; // pre valid allow the initial value of Beta to be written
               // into the memory one cycle before the actual Beta output.
               // This allow correct timing when calculating a-priori and LLR.

wire [w2:0] ps_a_up, ps_a_dn, ps_b_up, ps_b_dn, ps_c_up, ps_c_dn, ps_d_up, ps_d_dn;
wire [w2:0] new_pm_a, new_pm_b, new_pm_c, new_pm_d;

wire valid, valid_a;
reg path_diff_en, new_met_en;

assign valid = valid_a;

path_met_cal #(branch_met_width(w1+1), .path_met_width(w2+1), .trellis_term(t))
  PMC_A(clk, rst, set, en, SNR_select,
        r_a_up, r_a_dn,
        pm_a_init, pm_a_up, pm_a_dn,
        ps_a_up, ps_a_dn, new_pm_a,
        valid_a, debug_a);

path_met_cal #(branch_met_width(w1+1), .path_met_width(w2+1))
  PMC_B(clk, rst, set, en, SNR_select,
        r_b_up, r_b_dn,
        pm_b_init, pm_b_up, pm_b_dn,
        ps_b_up, ps_b_dn, new_pm_b,
        valid_b, debug_b);

path_met_cal #(branch_met_width(w1+1), .path_met_width(w2+1))
  PMC_C(clk, rst, set, en, SNR_select,
        r_c_up, r_c_dn,
        pm_c_init, pm_c_up, pm_c_dn,
        ps_c_up, ps_c_dn, new_pm_c,
        valid_c, debug_c);

path_met_cal #(branch_met_width(w1+1), .path_met_width(w2+1))
  PMC_D(clk, rst, set, en, SNR_select,
        r_d_up, r_d_dn,
        pm_d_init, pm_d_up, pm_d_dn,
        ps_d_up, ps_d_dn, new_pm_d,
        valid_d, debug_d);

endmodule

`timescale 1 ns / 100 ps // time unit = 1ns; precision = 1/10 ns;

module path_met_cal(clk, rst, set, en, SNR_select,
                  branch_met_up, branch_met_dn,
                  pm_init, path_met_up, path_met_dn,
                  path_sum_up, path_sum_dn, new_path_met,
                  valid, debug);

parameter branch_met_width = 8;
parameter path_met_width = 16;
parameter trellis_term = 1'b0; // This parameter is use when the start/end state is known(1) or unknown(0)

localparam w1 = branch_met_width - 1;
localparam w2 = path_met_width - 1;
localparam i = trellis_term;

```

```

input clk, rst, set, en, SNR_select;
input [w1:0] branch_met_up, branch_met_dn;
input [w2:0] pm_init, path_met_up, path_met_dn;
output [w2:0] path_sum_up, path_sum_dn, new_path_met;
output valid;
output [1:0] debug;

reg [w2:0] new_path_met;
reg valid, carry;
wire overflow;
wire [1:0] debug;

reg [w2:0] path_sum_up, path_sum_dn, new_path_sum, path_corr, max_path;
reg [3:0] path_difference;
reg [1:0] correction_factor, compare_result;

assign debug = correction_factor;

//Overflow detection
assign overflow = ~new_path_met[w2] & max_path[w2];
always @ (clk) valid <= en;
always @ (posedge clk or posedge rst)
begin
    if (rst)begin
        new_path_met <= 0;
        new_path_met[3] <= i;
    end
    else if (set) new_path_met <= pm_init;
    else new_path_met <= max_path;
end

always @ (path_difference[3:0], SNR_select)
begin
    if (SNR_select)
        case (path_difference)
            4'h2: correction_factor = 2'b01; // path_difference = 1
            4'h1: correction_factor = 2'b01; // path_difference = 0.5
            4'h0: correction_factor = 2'b01; // path_difference = 0
            4'hF: correction_factor = 2'b01; // path_difference = -0.5
            4'hE: correction_factor = 2'b01; // path_difference = -1
            default: correction_factor = 2'b00;
        endcase
    else
        case (path_difference)
            4'h8: correction_factor = 2'b01; // path_difference = 2 or -2
            4'h7: correction_factor = 2'b01; // path_difference = 1.75
            4'h6: correction_factor = 2'b01; // path_difference = 1.5
            4'h5: correction_factor = 2'b01; // path_difference = 1.25
            4'h4: correction_factor = 2'b01; // path_difference = 1
            4'h3: correction_factor = 2'b10; // path_difference = 0.75
            4'h2: correction_factor = 2'b10; // path_difference = 0.5
            4'h1: correction_factor = 2'b10; // path_difference = 0.25
            4'h0: correction_factor = 2'b11; // path_difference = 0
            4'hF: correction_factor = 2'b10; // path_difference = -0.25
            4'hE: correction_factor = 2'b10; // path_difference = -0.5
            4'hD: correction_factor = 2'b10; // path_difference = -0.75
            4'hC: correction_factor = 2'b01; // path_difference = -1
            4'hB: correction_factor = 2'b01; // path_difference = -1.25
            4'hA: correction_factor = 2'b01; // path_difference = -1.5
            4'h9: correction_factor = 2'b01; // path_difference = -1.75
            default: correction_factor = 2'b00;
        endcase
    end
end

//path sum up adder

```

```

always @ (path_met_up, branch_met_up) path_sum_up = path_met_up + {{(w2-w1){branch_met_up[w1]}},branch_met_up};
//path sum dn adder
always @ (path_met_dn, branch_met_dn) path_sum_dn = path_met_dn + {{(w2-w1){branch_met_dn[w1]}},branch_met_dn};
//path comparator
always @ (path_sum_up,path_sum_dn)
begin
    compare_result = 2'b00;
    if(path_sum_up[w2:4] == path_sum_dn[w2:4]) compare_result[0] = 1'b1;
    if ({~path_sum_up[w2],path_sum_up} > {~path_sum_dn[w2],path_sum_dn}) compare_result[1] = 1'b1;
end
//path difference subtractor
always @ (path_sum_up[3:0], path_sum_dn[3:0]) path_difference = path_sum_up[3:0] - path_sum_dn[3:0];
//new path metric adder
always @ (path_sum_up[3:0], correction_factor) {carry_up, path_corr[3:0]} = {1'b0, path_sum_up[3:0]} + correction_factor;
//incrementer
always @ (carry_up, path_sum_up) path_corr[15:4] = path_sum_up[15:4] + carry_up;
//MUX for path with maximum path metric
always @ (compare_result, path_corr, path_sum_up, path_sum_dn) begin
    case(compare_result)
        2'b00: max_path = path_sum_dn;
        2'b10: max_path = path_sum_up;
        2'b11: max_path = path_corr;
    endcase
end
endmodule

```

A.10 LLR and A-Posteriori Data Calculation Unit

`timescale 1 ns /100 ps // time unit = 1ns; precision = 1/10 ns;

```

module aposteriori_cal( clk, rst, en, busy, sys, a_priori,
    beta_a, ps_pa, ps_na, beta_b, ps_pb, ps_nb,
    beta_c, ps_pc, ps_nc, beta_d, ps_pd, ps_nd,
    LLR_reg, a_posteriori_reg, LLR_valid, valid, overflow);

    parameter a_priori_width = 8;
    parameter path_met_width = 16;

    localparam w1 = a_priori_width - 1;
    localparam w2 = path_met_width - 1;

    input clk, rst, en;
    input [3:0] sys;
    input [w1:0] a_priori;
    input [w2:0] beta_a, ps_pa, ps_na, beta_b, ps_pb, ps_nb,
        beta_c, ps_pc, ps_nc, beta_d, ps_pd, ps_nd;
    output busy;
    output [w2:0] LLR_reg;
    output [w1:0] a_posteriori_reg;
    output LLR_valid, valid, overflow;

    reg [w2:0] LLR,LLR_reg;
    reg [w1+1:0] a_posteriori;
    reg [w1:0] a_posteriori_reg;
    wire busy, overflow_internal;
    reg local_max_en, max_sum_en, LLR_en, a_pos_en, valid, overflow;
    reg [w2:0] sum_a_p, sum_a_n, sum_b_p, sum_b_n, sum_c_p, sum_c_n, sum_d_p, sum_d_n,
        sum_a_p_r, sum_a_n_r, sum_b_p_r, sum_b_n_r, sum_c_p_r, sum_c_n_r, sum_d_p_r, sum_d_n_r,
        max_ab_p_r, max_ab_n_r, max_cd_p_r, max_cd_n_r, max_sum_p_reg, max_sum_n_reg;
    wire [w2:0] max_ab_p, max_ab_n, max_cd_p, max_cd_n, max_sum_p, max_sum_n;
    wire LLR_valid;
    reg mux_ab_p, mux_ab_n, mux_cd_p, mux_cd_n, mux_sum_p, mux_sum_n;

```

```

assign busy = local_max_en | max_sum_en | LLR_en | a_pos_en;
assign LLR_valid = a_pos_en;

always @ (posedge clk or posedge rst)
    begin
        if (rst)begin
            LLR_reg <= 0;
            a_posteriori_reg <= 0;

            local_max_en <= 0;
            max_sum_en <= 0;
            LLR_en <= 0;
            a_pos_en <= 0;
            valid <= 0;

            overflow <= 0;

            sum_a_p_r <= 0;
            sum_a_n_r <= 0;
            sum_b_p_r <= 0;
            sum_b_n_r <= 0;
            sum_c_p_r <= 0;
            sum_c_n_r <= 0;
            sum_d_p_r <= 0;
            sum_d_n_r <= 0;

            max_ab_p_r <= 0;
            max_ab_n_r <= 0;
            max_cd_p_r <= 0;
            max_cd_n_r <= 0;

            max_sum_p_reg <= 0;
            max_sum_n_reg <= 0;
        end
        else
            begin
                local_max_en <= en;
                max_sum_en <= local_max_en;
                LLR_en <= max_sum_en;
                a_pos_en <= LLR_en;
                valid <= a_pos_en;
                if (en | a_pos_en) begin
                    overflow <= overflow_internal;
                end
                if (en) begin
                    sum_a_p_r <= sum_a_p;
                    sum_a_n_r <= sum_a_n;
                    sum_b_p_r <= sum_b_p;
                    sum_b_n_r <= sum_b_n;
                    sum_c_p_r <= sum_c_p;
                    sum_c_n_r <= sum_c_n;
                    sum_d_p_r <= sum_d_p;
                    sum_d_n_r <= sum_d_n;
                end
                if (local_max_en) begin
                    max_ab_p_r <= max_ab_p;
                    max_ab_n_r <= max_ab_n;
                    max_cd_p_r <= max_cd_p;
                    max_cd_n_r <= max_cd_n;
                end
                if (max_sum_en) begin
                    max_sum_p_reg <= max_sum_p;
                    max_sum_n_reg <= max_sum_n;
                end
                if (LLR_en) LLR_reg <= LLR;
            end
        end
    end

```



```

        if(a_pos_en) a_posteriori_reg <= a_posteriori[w1:0];
    end

    //Overflow detection
    assign overflow_internal=
        // detect overflow in adder when +add+=- or -add-=+
        ps_pa[w2] & beta_a[w2] & ~sum_a_p[w2] | // adder1
        ~ps_pa[w2] & ~beta_a[w2] & sum_a_p[w2] |
        ps_na[w2] & beta_a[w2] & ~sum_a_n[w2] | // adder2
        ~ps_na[w2] & ~beta_a[w2] & sum_a_n[w2] |
        ps_pb[w2] & beta_b[w2] & ~sum_b_p[w2] | // adder3
        ~ps_pb[w2] & ~beta_b[w2] & sum_b_p[w2] |
        ps_nb[w2] & beta_b[w2] & ~sum_b_n[w2] | // adder4
        ~ps_nb[w2] & ~beta_b[w2] & sum_b_n[w2] |
        ps_pc[w2] & beta_c[w2] & ~sum_c_p[w2] | // adder5
        ~ps_pc[w2] & ~beta_c[w2] & sum_c_p[w2] |
        ps_nc[w2] & beta_c[w2] & ~sum_c_n[w2] | // adder6
        ~ps_nc[w2] & ~beta_c[w2] & sum_c_n[w2] |
        ps_pd[w2] & beta_d[w2] & ~sum_d_p[w2] | // adder7
        ~ps_pd[w2] & ~beta_d[w2] & sum_d_p[w2] |
        ps_nd[w2] & beta_d[w2] & ~sum_d_n[w2] | // adder8
        ~ps_nd[w2] & ~beta_d[w2] & sum_d_n[w2] |

        // detect overflow in subtractor when +sub=- or -sub+=+
        ~LLR[w2] & max_sum_p_reg[w2] & ~max_sum_n_reg[w2] |
        LLR[w2] & ~max_sum_p_reg[w2] & max_sum_n_reg[w2] |
        // detect overflow in a_posteriori when 2 msbs not the same
        (a_posteriori[w1+1]^a_posteriori[w1]);

    //path sum adder for a positive
    always @ (beta_a, ps_pa) sum_a_p = beta_a + ps_pa;
    //path sum adder for a negative
    always @ (beta_a, ps_na) sum_a_n = beta_a + ps_na;
    //path sum adder for b positive
    always @ (beta_b, ps_pb) sum_b_p = beta_b + ps_pb;
    //path sum adder for b negative
    always @ (beta_b, ps_nb) sum_b_n = beta_b + ps_nb;
    //path sum adder for c positive
    always @ (beta_c, ps_pc) sum_c_p = beta_c + ps_pc;
    //path sum adder for c negative
    always @ (beta_c, ps_nc) sum_c_n = beta_c + ps_nc;
    //path sum adder for d positive
    always @ (beta_d, ps_pd) sum_d_p = beta_d + ps_pd;
    //path sum adder for d negative
    always @ (beta_d, ps_nd) sum_d_n = beta_d + ps_nd;

    //path comparator for a and b positive
    always @ (sum_a_p_r, sum_b_p_r)
    begin
        if ({~sum_a_p_r[w2],sum_a_p_r} > {~sum_b_p_r[w2],sum_b_p_r}) mux_ab_p = 1'b1;
        else mux_ab_p = 1'b0;
    end

    //mux for selecting max A,B,r sum for a and b positive
    assign max_ab_p = (mux_ab_p)? sum_a_p_r:sum_b_p_r;

    //path comparator for a and b negative
    always @ (sum_a_n_r, sum_b_n_r)
    begin
        if ({~sum_a_n_r[w2],sum_a_n_r} > {~sum_b_n_r[w2],sum_b_n_r}) mux_ab_n = 1'b1;
        else mux_ab_n = 1'b0;
    end

    //mux for selecting max A,B,r sum for a and b negative
    assign max_ab_n = (mux_ab_n)? sum_a_n_r:sum_b_n_r;

    //path comparator for c and d positive

```

```

always @ (sum_c_p_r, sum_d_p_r)
begin
    if ({~sum_c_p_r[w2], sum_c_p_r} > {~sum_d_p_r[w2], sum_d_p_r}) mux_cd_p = 1'b1;
    else mux_cd_p = 1'b0;
end
//mux for selecting max A,B,r sum for c and d positive
assign max_cd_p = (mux_cd_p)? sum_c_p_r:sum_d_p_r;

//path comparator for c and d negative
always @ (sum_c_n_r, sum_d_n_r)
begin
    if ({~sum_c_n_r[w2], sum_c_n_r} > {~sum_d_n_r[w2], sum_d_n_r}) mux_cd_n = 1'b1;
    else mux_cd_n = 1'b0;
end
//mux for selecting max A,B,r sum for c and d negative
assign max_cd_n = (mux_cd_n)? sum_c_n_r:sum_d_n_r;

//path comparator for ab and cd postive
always @ (max_ab_p_r, max_cd_p_r)
begin
    if ({~max_ab_p_r[w2], max_ab_p_r} > {~max_cd_p_r[w2], max_cd_p_r}) mux_sum_p = 1'b1;
    else mux_sum_p = 1'b0;
end
//mux for selecting max A,B,r sum for ab and cd postive
assign max_sum_p = (mux_sum_p)? max_ab_p_r:max_cd_p_r;

//path comparator for ab and cd negative
always @ (max_ab_n_r, max_cd_n_r)
begin
    if ({~max_ab_n_r[w2], max_ab_n_r} > {~max_cd_n_r[w2], max_cd_n_r}) mux_sum_n = 1'b1;
    else mux_sum_n = 1'b0;
end
//mux for selecting max A,B,r sum for ab and cd negative
assign max_sum_n = (mux_sum_n)? max_ab_n_r:max_cd_n_r;

//LLR subtractor
always @ (max_sum_p_reg, max_sum_n_reg) LLR = max_sum_p_reg - max_sum_n_reg;

//A-posteriori subtractor
always @ (LLR_reg, a_priori, sys) a_posteriori = LLR_reg - {(w2-w1){a_priori[w1]}},a_priori} - {(w2-3){sys[3]}},sys};

endmodule

```

Appendix B: Matlab Codes

B.1 Top Level Simulator

```
function sim_turbo(messageWidth, numIteration, numFrame, SNR_low, SNR_high, precision);

%Initialization-----
BER_Result = [];
FER_Result = [];
rand('state', 0); %reset random generator
randn('state', 0); %reset random generator

%numInputSymbols (Scalar)
%Number of input symbols to the encoder: 2k
numInputSymbols = 2; %input are either 0 or 1

%numOutputsymbols (Scalar)
%Number of output symbols from the encoder: 2n
%MUST be INTEGER after taking log2(2n)
numOutputSymbols = 4; %output [sys, par] are either 00,01,10,11

%numStates (Scalar)
%Number of states in the encoder
numStates = 8;
termination_length = log2(numStates);

%quantizeWidth (Scalar)
%The width of quantize input for each bit of output symbols from encoder
%The MSB determines the sign
quantizeWidth = 3;

%nextStates (numStates-by-2k matrix)
%Next states for all combinations of current state and current input
%Collum 1 is for first input branch (eg.0) and
%Collum 2 is for second input branch (eg.1) etc.
nextStates = [0 4; 4 0; 5 1; 1 5; 2 6; 6 2; 7 3; 3 7];

%outputs (numStates-by-2*2k matrix)
%Outputs for all combinations of current state and current input
outputs = [-1 -1 1 1; -1 -1 1 1; -1 1 1 -1; -1 1 1 -1; -1 1 1 -1; -1 1 1 -1; -1 1 1 1; -1 1 1 1];

%find reverse value
[nextInputs, preInputs, preOutputs, preStates] = findReverse(outputs, nextStates);

trellis = struct('numInputSymbols',numInputSymbols,'numOutputSymbols',numOutputSymbols,...
    'numStates',numStates,'quantizeWidth',quantizeWidth,...
    'nextStates',nextStates,'preStates',preStates,...
    'nextInputs',nextInputs,'preInputs',preInputs,...
    'outputs',outputs,'preOutputs',preOutputs);

% Create Interleaver Matrix
Permute = randperm(messageWidth);
InvPermute(Permute) = 1:messageWidth;
%End Initialization-----

%generate inputs

Input1 = [round(rand(1,messageWidth))]; %Input1 is a random stream of 0s and 1s with the 1st bit being a 0
Input2 = Input1(Permute); %Input2 is the permutation of Input1
```

```

%encode messages
t1 = poly2trellis(4,[13 15], 13); % create trellis1 based on polynomial of encoder1
[Message1, endstate1] = convenc(Input1,t1,0); % Encode message by encoder1, init state = 0
Message1 = [Message1 findTermination(endstate1, t1)]; % attach termination
[Message2, endstate2] = convenc(Input2,t1,0); % Encode message by encoder2, init state = 0
Message2 = [Message2 findTermination(endstate2, t1)]; % attach termination

for SNR=SNR_low:precision:SNR_high,
    bit_error = 0;
    frame_error = 0;
    No = 3*10^-(SNR/10);
    for frameNumber=1:numFrame,
        frameNumber
        Outputs = turbo(No, SNR, numIteration, Permute, InvPermute, Message1, Message2, trellis, trellis);
        Errors = (Input1 ~= Outputs);
        if sum(Errors,2) ~= 0
            bit_error = sum(Errors,2) + bit_error;
            frame_error = frame_error + 1;
        end;
    end;
    format long;
    diary on;
    SNR
    BER = bit_error/(messageWidth*numFrame)
    FER = frame_error/numFrame
    diary off;
    BER_Result = [BER_Result ; SNR BER];
    FER_Result = [FER_Result ; SNR FER];

    %finish simulation if we see FER equal zero
    if FER == 0
        break;
    end;
end;

format long;
diary;
messageWidth
numIteration
numFrame
SNR_low
SNR_high
precision
FER_Result
BER_Result
diary off;
format;

```

B.2 Turbo Decoder

```

%Filename: turbo.m
%Version: 3.2
%Author: Henry Wong
%Create: September 5, 2003
%Last Modify: October 5, 2004
%Descriptions: Top level file for k/n = 1/3 Turbo Code
%Version 0: Initial code
%Version 1: Implement turbo as a function that return BER
%Version 2: Reimplement the two encoders to two eight states RSC encoders.
%Version 3: Move the encoder out to speed up simulation
% 3.1: Add trellis termination
% 3.2: Improve runtime of interleaving

```

```

%Version 4: Try quantization for Lc.
%Inputs: No = single side psd of noise
% numIteration = number of loops in turbo decoder iteration
% Permute = interleaver vector
% InvPermute = deinterleaver vector
% Message1,2 = encoder outputs
% trellis_A,B = decoder trellis with soft value
%Outputs: Outputs = hard decision decoder output

function Outputs = turbo(No, SNR, numIteration, Permute, InvPermute, Message1, Message2, trellis_A, trellis_B)

%Initialization-----
noise_delta = sqrt(No/2); %standard deviation of gaussian noise 1.5~3dB SNR 2.38~1dB SNR
msg_width = length(Message1); %real message does not included the final 6 bits
data_width = (msg_width/2)-3;
A_Priori = zeros(1,data_width);
%End Initialization-----

%step 1 create gaussian noise channel
Noise = randn(size(Message1))*noise_delta; %Gaussian Noise of mean 3 and standard deviation of noise_delta
Receive1 = ((Message1 * 2 - 1) + Noise); %received waveform
Noise = randn(size(Message2))*noise_delta; %Gaussian Noise of mean 3 and standard deviation of noise_delta
Receive2 = ((Message2 * 2 - 1) + Noise); %received waveform

% %step 2 samples channel inputs to be received inputs
% Sample1 = (Receive1<-3)*(-3) + (Receive1>3)*3 + (abs(Receive1)<=3) .* Receive1; %sample of received waveform
% Sample2 = (Receive2<-3)*(-3) + (Receive2>3)*3 + (abs(Receive2)<=3) .* Receive2; %sample of received waveform

%step 2 samples channel inputs to be received inputs
Sample1 = Receive1; %sample of received waveform
Sample2 = Receive2; %sample of received waveform

%set sys of Sample2 equal sys of Sample1
Sys1 = Sample1(1:2:msg_width-6);
Sys2 = Sys1(Permute);
Sample2(1:2:msg_width-6) = Sys2;

%Calculate Channel Reliability
% Lc = 2/noise_delta^2;

Lc = 1;
% if (SNR <= 1.75)
%   Lc = 1;
% elseif (SNR <= 4.75)
%   Lc = 2;
% elseif (SNR <= 7.75)
%   Lc = 4;
% else
%   Lc = 8;
% end;

Sys1 = Lc * Sys1;
Sys2 = Lc * Sys2;

%Calculate LLR of Sys bits
% LLR_Sys1 = log((abs(Sys + trellis_A.quntizeWidth)+1)/(abs(Sys - trellis_A.quntizeWidth)+1));
% LLR_Sys2 = LLR_Sys1(Permute);
% LLR_Sys1 = zeros(size(Sys));
% LLR_Sys2 = LLR_Sys1;

%step 3 decode using trellis with logMAP algorithm

Tail_A_Priori = zeros(1,3);
LLR2 = zeros(1,data_width+3);
A_Priori = LLR2(1:data_width);

```

```

for n=1:numIteration,
    LLR1 = APP_decoder(Sample1, [A_Priori Tail_A_Priori], Lc, trellis_A);
    % A_Priori = LLR1(1:data_width) - A_Priori - LLR_Sys1;
    A_Priori = LLR1(1:data_width) - Sys1 - A_Priori;
    max_A_Priori = max(abs(A_Priori))
    A_Priori = A_Priori(Permute);
    LLR2 = APP_decoder(Sample2, [A_Priori Tail_A_Priori], Lc, trellis_B);
    % A_Priori = LLR2(1:data_width) - A_Priori - LLR_Sys2;
    A_Priori = LLR2(1:data_width) - Sys2 - A_Priori;
    A_Priori = A_Priori(InvPermute);
end;

%step 4 make hard decision about output
Outputs = (sign(sign(A_Priori) + 0.1) + 1)/2; %if LLR is zero it will be bias to one due to the factor of 0.1

```

B.3 Log-MAP Decoder

```

%Filename: APP_decoder.m
%Version: 2.1
%Author: Henry Wong
%Create: September 5, 2003
%Last Modify: December 5, 2003
%Descriptions: Soft Trellis decoder using logMAP algorithm and fix windowing
%Version1: Use fix windowing
%Version2: Move out creation of reverse matrix to speed up simulation
% 2.1: Trellis termination introduced
%Inputs: Channel Memory
% Trellis: {numInputSymbols, numOutputSymbols, numStates,
%          quntizeWidth, nextStates, outputs}
%Output: LLR
function LLR = APP_decoder(Channel, A_Priori, Lc, trellis)

%Initialization-----
channel_size = size(Channel,2);
memory_size = channel_size/2;
W = (log2(trellis.numStates)+1) * 10; %window size
Alpha_Memory = zeros(trellis.numStates, memory_size);
Alpha_Memory(:,1) = 0; %initial values of Alphas at head of message
Alpha_Memory(1,1) = 31;
LLR = zeros(1,memory_size);

nextInputs = trellis.nextInputs;
Previous_Inputs = trellis.preInputs;
Previous_Outputs = trellis.preOutputs;
Previous_States = trellis.preStates;
max_beta = 0;
max_alpha = 0;

%End Initialization-----
%Step1 Going forward and filled up the Alpha memory
for a=1:memory_size-1,
    [Alpha_Memory(:,a+1), Reverse_Gamma] = logMAP_sum(Channel(1,2*a-1:2*a), A_Priori(1,a), Previous_Inputs, ...
                                                    Lc, Previous_Outputs, ...
                                                    Previous_States, Alpha_Memory(:,a));
end;
max_alpha = max(max(abs(Alpha_Memory)))

%Step2 Backtracing and calculated LLR ASSUSME with have 2bit inputs
Beta = zeros(trellis.numStates,1); %initialize Beta memory
Beta(1,1) = 31;

```

```

%No Window*****
for q=memory_size:-1:1,
    Previous_Beta = Beta;
    if max(abs(Beta)) > max_beta
        max_beta = max(abs(Beta));
    end
    [Beta, Gamma] = logMAP_sum(Channel(1,2*q-1:2*q), A_Priori(1,q), nextInputs, Lc, ...
        trellis.outputs,trellis.nextStates, Previous_Beta);
    LLR(1,q) = calculate_LLRL(Alpha_Memory(:,q), Previous_Beta, Gamma, trellis.nextStates);
end;

max_beta

%Fix Windowing*****
% %Get the first W output
% for b=memory_size:-1:memory_size-W+1,
%     Previous_Beta = Beta;
%     [Beta, Gamma] = logMAP_sum(Channel(1,2*b-1:2*b), A_Priori(1,b), nextInputs, Lc, ...
%         trellis.outputs,trellis.nextStates, Previous_Beta);
%     LLR(1,b) = calculate_LLRL(Alpha_Memory(:,b), Previous_Beta, Gamma, trellis.nextStates);
% end;
%
% Beta_init = zeros(trellis.numStates,1); %initialize Beta_init memory
%
% %Get the second W output
% for c=memory_size-W:-1:memory_size-2*W+1,
%     Previous_Beta = Beta;
%     %first Beta unit is for initialization
%     [Beta_init, Gamma] = logMAP_sum(Channel(1,2*c-1:2*c), A_Priori(1,c), nextInputs, Lc, ...
%         trellis.outputs,trellis.nextStates, Beta_init);
%     %second Beta unit is for output
%     [Beta, Gamma] = logMAP_sum(Channel(1,2*c-1:2*c), A_Priori(1,c), nextInputs, Lc, ...
%         trellis.outputs,trellis.nextStates, Previous_Beta);
%     LLR(1,c) = calculate_LLRL(Alpha_Memory(:,c), Previous_Beta, Gamma, trellis.nextStates);
% end;
%
% %Get all the remaining output
%
% for d=2:floor(memory_size/W),
%     Beta = Beta_init;
%     Beta_init = zeros(trellis.numStates,1); %reset Beta to 0
%
%     for e=memory_size-d*W:-1:memory_size-(d+1)*W+1,
%         Previous_Beta = Beta;
%         %first Beta unit is for initialization
%         [Beta_init, Gamma] = logMAP_sum(Channel(1,2*e-1:2*e), A_Priori(1,e), nextInputs, Lc, ...
%             trellis.outputs,trellis.nextStates, Beta_init);
%         %second Beta unit is for output
%         [Beta, Gamma] = logMAP_sum(Channel(1,2*e-1:2*e), A_Priori(1,e), nextInputs, Lc, ...
%             trellis.outputs,trellis.nextStates, Previous_Beta);
%         LLR(1,e) = calculate_LLRL(Alpha_Memory(:,e), Previous_Beta, Gamma, trellis.nextStates);
%         if (e==1)
%             break;
%         end;
%     end;
% end;
% end;

```

B.4 Path Metric Unit

```
%Create: September 10, 2003
%Last Modify: Janary 8, 2004
%Descriptions: Caculate the path sum based on the expected values, inputs
%               and previous values.
%***NOTE*** This function only works with channel input = (sys,par)
%Inputs: Channel Memory
%         Expected Inputs to decoder (trellis.outputs)
%         Previous States
%         Previous values
%Output: Sum of path metrics
%        Gamma value
function [Sum, Gamma] = logMAP_sum(Inputs, A_Priori, Encoder_Inputs, Lc,...
    Expected_Outputs,Previous_States,Old_Sum);

%Assume only two next branches per state

% Full penalize/rewarding scheme *****
Gamma = 0.5 * A_Priori * Encoder_Inputs + 0.5 * Lc * ( Inputs(1) * Expected_Outputs(:,1:2:4)...
    + Inputs(2) * Expected_Outputs(:,2:2:4) );

%Vectorize code
All_Sum = Gamma + Old_Sum(Previous_States + 1);
Sum = max(All_Sum)' + log(1+exp(-1*abs(All_Sum(:,1)-All_Sum(:,2))));
```

B.5 LLR and A-Posteriori Data Calculation Unit

```
%Create: September 10, 2003
%Last Modify: Jan 8, 2004
%Descriptions: Caculate the LLR based on the Alpha, Beta and Gamma values (Rewarding scheme).
%***NOTE*** This function only works with channel input = (sys,par)
%Inputs: Alpha, Beta, Gamma, Next_States
%        Previous values
%Output: LLR
function LLR = calculate_LLR(Alpha,Beta,Gamma,Next_States)

% Vectorize code
Zero_Branches = Alpha + Gamma(:,1) + Beta(Next_States(:,1)+1);
One_Branches = Alpha + Gamma(:,2) + Beta(Next_States(:,2)+1);

LLR = max(One_Branches) - max(Zero_Branches);
```


Vita

Henry Wong was born in Hong Kong, China where he finished his primary and secondary education.

In 1996, he was admitted to the Electrical Engineering Program in Carleton University, Ottawa, Ontario, Canada. He worked as a co-op software designer in Newbridge Networks in the summer of 1998 and 1999 and finished his B.Eng.(Elec.) with high distinction in 2000.

From 2000 to 2002, he worked as a FPGA/PCB designer in Alcatel Canada developing telecommunication equipment for high speed data network such as ATM and OC-192.

From 2002 to 2003, he worked as an instructor for an Intel teaching course related to network processors. He was also an application specialist supporting Intel network processor designs.

From 2003 to 2005, he became a full time graduate student of Prof. John Knight in the Department of Electronics in Carleton University. He was a member of VLSI group performing research on wireless applications.

He will receive his M.A.Sc degree in 2005. After graduation, he will join Micron Technology in Boise, Idaho, USA. as a FPGA/PCB designer, and develop hardware testers for performance and competitive analysis of the latest memory products.

Henry Wong can speak and write in five languages, and he really enjoys travelling and participating in group sport activities, such as basketball and soccer.