

FPGA-based LTE TURBO Decoder Coprocessor
Interfaced with a GPP through PCIe

By

Mohamed Adel Hafez Emeash

Mohamed Mahmoud Mostafa Talha

Mohamed Mohamed Mahrous Amer

Mahmoud Mohamed Ali Asy

Mahmoud Mostafa Mohamed Bahnasy

Mostafa Tharwat Abdelhalim Salama

Under supervision of

Dr. Hassan Mostafa

Dr. Yasmin Fahmy

A Graduation Project Report Submitted to
the Faculty of Engineering at Cairo University
in Partial Fulfillment of the Requirements for the
Degree of
Bachelor of Science
in
Electronics and Communications Engineering

Faculty of Engineering, Cairo University

Giza, Egypt

July 2017

Table of Contents

List of Tables	viii
List of Figures	ix
List of Symbols and Abbreviations.....	xiii
Acknowledgments.....	xiv
Abstract.....	xv
Chapter 1: Introduction.....	16
1.1 C-RAN (Cloud/Centralized Radio Access Network).....	16
1.2 Baseband Processing	19
1.3 Forward error correction (FEC)	20
Chapter 2: Convolutional Codes and Decoding Methods	23
2.1 Overview	23
2.2 Parity Equations	24
2.3 Views of the Convolutional Encoder	25
2.3.1 Block Diagram View	26
2.3.2 State Machine View	26
2.4 Trellis Structure.....	27
2.5 Channel Models.....	28
2.5.1 Binary Symmetric Channel.....	28
2.5.2 Additive White Gaussian Noise Channel.....	29
2.6 Decoding Convolutional Codes.....	30
2.6.1 The Viterbi Decoder	32
2.7 Soft-Decision Decoding	36
Chapter 3: Turbo Codes.....	41
3.1 Turbo Encoder.....	41
3.1.1 Recursive Systematic Convolutional (RSC) Encoder.....	41
3.1.2 Trellis Termination	44

3.1.3	Concatenation of Codes	45
3.1.4	Interleaver Design	47
3.1.5	LTE Turbo Encoder in the 3GPP Standard.....	49
3.1.6	Interleaver	51
3.2	Decoder	52
3.2.1	Turbo Decoder	52
3.2.2	Principle of the General Soft-Output Viterbi Decoder	54
3.2.3	Likelihood Functions	54
3.2.4	Reliability of the General SOVA Decoder	58
3.2.5	SOVA Component Decoder for a Turbo Code.....	63
3.2.6	SOVA Iterative Turbo Code Decoder.....	68
Chapter 4:	Matlab Results and Analysis.....	71
4.1	Matlab Implementation	71
4.2	BER Performance over AWGN Channel.....	72
4.3	Interpretation of Results	75
Chapter 5:	Hardware Architecture for SOVA	76
5.1	SOVA Component	76
5.1.1	Trellis Stage	76
5.2	Trace back and Updating Depths	79
5.2.1	Merge Stage	80
5.2.2	Decode Stage	81
5.3	Block diagram of the hardware architecture for a SOVA decoder	81
5.4	Sliding Window.....	85
5.5	Quantization	88
5.5.1	Quantization Process.....	88
5.5.2	Choosing the proper word size	91
Chapter 6:	Hardware Implementation and Results	92

6.1	Design Flow in ISE	93
6.1.1	Design Entry	93
6.1.2	Synthesis	93
6.1.3	Implementation	93
6.1.4	Verification	94
6.1.5	Device Configuration.....	94
6.2	Testing framework for SOVA Block	95
6.3	Simulation Results of Behavioral RTL design.....	97
6.4	Comparison with Software Reference.....	97
6.5	Hardware Performance.....	98
6.6	Turbo Decoder implementation	99
6.6.1	Scheduling of Computation at block level.....	99
6.6.2	Turbo Decoder as a black box	101
6.6.3	Additional Feature:	103
6.6.4	Generic Parameters	104
6.7	Simulation Results of Behavioral RTL design.....	104
6.7.1	Initialization and Data Input to Decoder.....	104
6.7.2	Decoder Output When Processing is Done.....	105
6.8	Performance and Resource Usage.....	106
6.9	BER Performance.....	107
Chapter 7:	PCI Express Interconnect.....	108
7.1	PCIe Link	109
7.2	PCIe Clock Recovery	110
7.3	PCIe Fabric Topology	111
7.4	PCIe Layering Overview.....	112
7.4.1	Transaction layer.....	114
7.4.2	Data link layer.....	115

7.4.3	Physical layer	115
7.5	Types of PCI Express Protocol	116
7.6	Why PCI Express Interface?	117
Chapter 8:	PCI Express Linux Driver and RIFFA Framework	119
8.1	PCI Addressing	119
8.2	Direct Memory Access.....	120
8.2.1	DMA mappings.....	122
8.2.2	How a PCIe driver works.....	122
8.3	RIFFA Framework	122
8.3.1	RIFFA 1.0	123
8.3.2	RIFFA 2.1	123
8.3.3	Design	124
8.3.4	RIFFA Software.....	125
8.3.5	Hardware Interface.....	128
8.3.6	Architecture.....	132
8.3.7	RIFFA 2.1 FPGA Support	137
8.3.8	RIFFA 2.1 Bandwidth.....	137
Chapter 9:	Turbo Interfacing	138
9.1	Hardware Interface	139
9.1.1	PCIe Endpoint Core	139
9.1.2	Turbo Timing Constraints.....	148
9.1.3	Design	151
9.1.4	Synthesizing the Design.....	153
9.1.5	Configure Target Device.....	153
9.2	Software Interface	154
9.2.1	Installing the RIFFA Driver.....	154
9.2.2	User Space Application.....	155

Chapter 10: Conclusion.....	158
References.....	159
Appendix A: Interleaver Table	162
Appendix B: Path metric derivation	163
Appendix C: MATLAB Code.....	165

List of Tables

Table 3.1: Input and Output Sequences for Encoder in Figure 3-8	48
Table 4.1: BER values for frame K=64 over AWGN Channel	72
Table 4.2: BER values for frame K=1024 over AWGN Channel	73
Table 4.3: BER values for frame K=6144 over AWGN Channel	74
Table 5.1: BER values for different word sizes using frame size K=1024.....	92
Table 6.1: Description of SOVA Signals.....	95
Table 6.2: FPGA Resources of SOVA Module	98
Table 6.3: Description of Turbo Decoder Signals	102
Table 6.4: Description of Turbo Decoder module parameters	104
Table 6.5: FPGA Resources used by Turbo Decoder	107
Table 7.1:PCI Express different generations speed comparison	110
Table 8.1: PCI Configuration Space	120
Table 8.2: Functions of RIFFA API	126
Table 8.3: Hardware Interface Receive Ports	129
Table 8.4: Hardware Interface Transmit Ports.....	130

List of Figures

Figure 1-1 C-RAN Architecture	17
Figure 2-1 Convolutional code with two parity bits per message bit ($r = 2, K=3$).	24
Figure 2-2 Block diagram view of convolutional coding with shift registers.	25
Figure 2-3 State machine view of convolutional coding.	27
Figure 2-4 Trellis Diagram.	28
Figure 2-5 Binary Symmetric Channel Model.....	29
Figure 2-6 Trellis Structure.....	32
Figure 2-7 The branch metric for hard decision decoding.....	34
Figure 2-8 The Viterbi Decoder in Action.....	37
Figure 2-9 The Viterbi decoder in Action continued.....	38
Figure 2-10 Branch Metric For Soft-Decision Decoding	39
Figure 3-1: Fundamental turbo code encoder.	41
Figure 3-2: Conventional convolutional encoder with $r=1/2$ and $K=3$	42
Figure 3-3: The RSC encoder obtained from Figure 3.2 with $r=1/2$ and $K=3$	43
Figure 3-4: Trellis termination strategy for RSC encoder	44
Figure 3-5: Serial concatenated code.	45
Figure 3-6: Parallel concatenated code.	45
Figure 3-7: The interleaver increases the code weight for Encoder 2 compared to Encoder 1.	47
Figure 3-8: An illustrative example of an interleaver's capability.	48
Figure 3-9: Structure of LTE Turbo Encoder.	49
Figure 3-10: Turbo Decoder	53
Figure 3-11: A concatenated SOVA decoder where y represents the received channel values, u represents the hard decision output values, and L represents the associated reliability values.....	54
Figure 3-12: Example of survivor and competing paths for reliability estimation at time t	58
Figure 3-13: Example that shows the weakness of reliability assignment using metric	60
Figure 3-14: Updating process for time $t-4$ ($MEM_{low}=4$).	62
Figure 3-15: Updating process for time $t-2$ ($MEM_{low}=2$).	63
Figure 3-16: SOVA component decoder	63

Figure 3-17: Source reliability for SOVA metric computation	65
Figure 3-18: Example of SOVA survivor and competing paths for reliability estimation.....	66
Figure 3-19: SOVA iterative turbo code decoder.....	69
Figure 4-1 BER for frame K=64 over AWGN Channel	72
Figure 4-2 BER for frame K=1024 over AWGN Channel	73
Figure 4-3: BER for frame K=6144 over AWGN Channel.....	74
Figure 4-4: BER Comparison between different frame sizes over AWGN using 3 iterations.....	75
Figure 5-1: BMC module in the trellis unit	78
Figure 5-2: Trellis unit for LTE consisting of 8 ACS.....	79
Figure 5-3: Block diagram of register exchange unit	81
Figure 5-4: System architecture of SOVA decoder	82
Figure 5-5: Block diagram of SMU module	83
Figure 5-6: Encoder to determine state s_{k-L}	83
Figure 5-7: PCU for SOVA	84
Figure 5-8: Block diagram of UPD module.....	85
Figure 5-9: One bit releases sliding window decoding.....	86
Figure 5-10 BER for frame size K=1024 over AWGN Channel using 1 iteration.....	87
Figure 5-11 BER for frame size K=1024 over AWGN channel using 2 iterations	87
Figure 5-12 Scaling the recieved value in quantization process.....	89
Figure 5-13 Scaling Factor and Limits for quantization process.....	90
Figure 5-14 BER for frame size K=1024 over AWGN Channel for different word sizes.....	92
Figure 6-1: Design and verification process of the FPGA implementations	94
Figure 6-2: Interface of SOVA Block.....	95
Figure 6-3: RTL Schematic of SOVA Module.....	96
Figure 6-4: Loading SOVA Block with Input Data.....	97
Figure 6-5: Output of SOVA Block.....	97
Figure 6-6: Performance of SOVA	98
Figure 6-7: Turbo Decoder Archeticture	99
Figure 6-8: Implementation of Turbo Decoder.....	100
Figure 6-9: Control Unit of Turbo Decoder.....	101
Figure 6-10: RTL Schematic of Turbo Decoder	101

Figure 6-11: Interface of Turbo Decoder	102
Figure 6-12: handshaking signals: Start.....	104
Figure 6-13: handshaking signals: Finished	105
Figure 6-14: handshaking signals: Ready_To_Get_Output signal	105
Figure 6-15: handshaking signals: Ready_To_Process signal.....	106
Figure 6-16: handshaking signals: Valid_Output signal.....	106
Figure 6-17: BER Comparision for frame size K=1024 using 1 iteration.....	108
Figure 7-1 PCI Express Link	109
Figure 7-2 PCI Express Topology	112
Figure 7-3 PCI Express Layering Diagram	113
Figure 7-4 PCI Express soft and hardened implementation	116
Figure 7-5 The Spartan-6 FPGA SP605 Evaluation Kit [3]	117
Figure 7-6 Spartan-6 FPGA IP Core Specifications [4]	118
Figure 8-1:Receive Timing Diagram	130
Figure 8-2: Transmit Timing Diagram.	132
Figure 8-3: RIFFA Hardware Architecture.....	133
Figure 8-4: Upstream Data Transfer	136
Figure 8-5: RIFFA bandwidth.	138
Figure 9-1: Xilinx CORE Generator.....	140
Figure 9-2: Xilinx CORE Generator Project Options.....	141
Figure 9-3: Xilinx CORE Generator Design Entry.....	142
Figure 9-4: Xilinx CORE Generator generating IP CORE.....	142
Figure 9-5: Spartan-6 Integrated Block for PCI Express.....	143
Figure 9-6: PCI Express CORE Base Address Registers.	144
Figure 9-7: PCI Express CORE ID Initial Values.	145
Figure 9-8: PCI Express CORE Max Payload Size.	146
Figure 9-9: PCI Express Xilinx Reference Boards.	147
Figure 9-10: PCIe IP CORE Directory.	147
Figure 9-11: Turbo decoder critical path delay.....	148
Figure 9-12: Timing Constraints met.....	148
Figure 9-13: Clocking Features for Spartan-6 FPGA	149
Figure 9-14: Output Clock Settings Screen	149
Figure 9-15: Clock Summary Screen.....	150
Figure 9-16: Turbo Interfacing design	151

Figure 9-17: FPGA Resources Summary	153
Figure 9-18: sp605 Configuration options	153
Figure 9-19: FPGA Startup Options	154
Figure 9-20: Turbo performance on Hardware	157

List of Symbols and Abbreviations

3GPP	3rd Generation Partnership Project
ACS	Add, Compare, and Select
API	Application Programming Interface
APP	Aposteriori probability
BER	Bit Error Rate
BSC	Binary Symmetric Channel
CLI	Command line interface.
C-RAN	Cloud Radio Access Network
DMA	Direct memory access.
DSP	Digital Signal Processor
FEC	Forward Error Correction
FIFO	First In First Out
GPP	General Purpose Processor
IP	Intellectual Property
KLM	Kernel loadable module.
L1	Layer 1 (Physical layer in the OSI Model)
LTE	Long Term Evolution
MAP	Maximum Aposteriori Probability
ML	Maximum Likelihood
PCIe	Peripheral Component Interconnect Express
PDF	Probability Density Function
POSIX	Portable Operating System Interface
QPI	Quick Path Interconnect
RIFFA	Reusable Integration Framework for FPGA Accelerators
RSC	Recursive Systematic Code
SOVA	Soft Output Viterbi Algorithm

Acknowledgments

We would like to thank Dr. Hassan Mustafa for giving us the opportunity to work with him on this project and supplying us with the needed hardware. We would also like to thank Dr. Yasmin Fahmy for her support in the Turbo Decoder part.

Finally, we would like to thank Eng. Mohammed Soliman for supporting us in the communication concepts. Also we would like to thank Eng. Mohammed Farag for his role in supporting us.

Abstract

In the era of mobile Internet, mobile operators are facing pressure on ever-increasing capital expenditures and operating expenses with much less growth of income. Cloud Radio Access Network (C-RAN) is expected to be a candidate of next generation access network techniques that can solve the operators' puzzle. The main idea of C-RAN is the cloud processing for multiple cells. One of the major trends towards achieving such a cloud is to make the L1 DSP computationally intensive processing on General Purpose Processor (GPP) based architectures. For this cloud to be realized it is required to offload the most complex L1 processing portions (like FEC decoders) to a highly parallel platform (FPGA) giving more powerful options to the cloud.

In this project, different LTE Turbo decoder algorithms were explored and the SOVA (Soft-Viterbi Algorithm) was chosen owing to its relative simplicity, a MATLAB model was constructed for the decoder and tested for performance, an RTL code for the built model was then written in Verilog hardware description language and tested on the target XILINX SPARTAN-6 FPGA, finally, a Linux driver was developed for the PCIe with the help of the RIFFA framework using C programming language to interface the FPGA with a GPP located in a PC.

Chapter 1: Introduction

1.1 C-RAN (Cloud/Centralized Radio Access Network)

Nowadays mobile operators are facing a serious situation. With the introduction of various air interface standards and the prevalence of smart devices, mobile Internet traffic is surging, and operators are forced to increase capital expenditure (CAPEX) and operating expense (OPEX) in order to meet users' requirements. On the other hand the average revenue per user (ARPU) cannot catch up with the increasing expenses. It is predicted that the traffic will double every year in the next few years till 2020 [1], which will require more cost to build, operate, and upgrade the network infrastructure, while only a small increase on the revenue is expected. The operators have to find new solutions to maintain a healthy profit and provide better services for customers.

On the other hand, the proliferation of mobile broadband internet also presents a unique opportunity for developing an evolved network architecture that will enable new applications and services, and become more energy efficient.

The RAN is the most important asset for mobile operators to provide high data rate, high quality, and 24x7 services to mobile users. Traditional RAN architecture has the following characteristics: first, each Base Station (BS) only connects to a fixed number of sector antennas that cover a small area and only handle transmission/reception signals in its coverage area; second, the system capacity is limited by interference, making it difficult to improve spectrum capacity; and last but not least, BSs are built on proprietary platforms as a vertical solution. These characteristics have resulted in many challenges. For example, the large number of BSs requires corresponding initial investment, site support, site rental and management support. Building more BS sites means increasing CAPEX and OPEX. Usually, BS's utilization rate is low because the average network load is usually far lower than that in peak load; while the BS' processing power can't be shared with other BSs. Isolated BSs prove costly and difficult to improve spectrum capacity. Lastly, to meet the fast increasing data services, mobile operators need to upgrade their network frequently and operate multiple-standard

network, including GSM, WCDMA/TD-SCDMA and LTE. However, the proprietary platform means mobile operators lack the flexibility in network upgrade, or the ability to add services beyond simple upgrades [2].

In summary, traditional RAN will become far too expensive for mobile operators to keep competitive in the future mobile internet world.

Nowadays multi-core processors are becoming increasingly powerful, and the cloud computing-based open IT platform is a promising alternative for both IT service providers and mobile operators. It is time for mobile operators to consider using the cloud computing facility to form a much larger processing resource pool shared in a large geographical area to achieve low-cost operation.

Cloud Radio Access Network (C-RAN) is a new paradigm proposed by a few operators that features centralized processing, collaborative radio, real-time cloud computing, and power efficient infrastructure. This novel architecture aggregates all BS computational resources into a central pool; the radio frequency signals from geographically distributed antennas are collected by remote radio heads (RRHs) and transmitted to the cloud platform through an optical transmission network (OTN). It aims to reduce the number of cell sites while maintaining similar coverage, and reducing capital expenditures and operating expenses while offering better services.

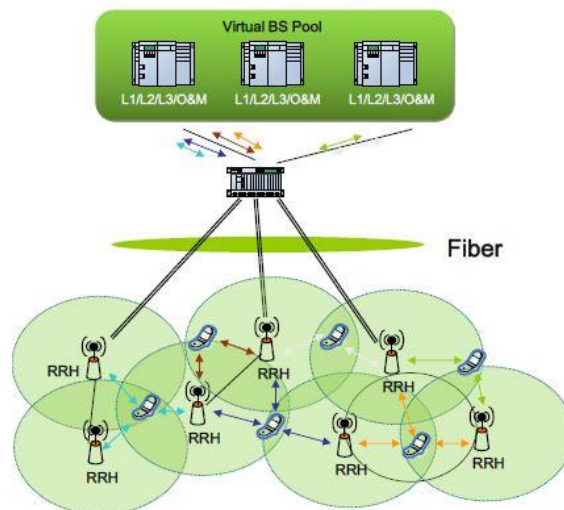


Figure 1-1 C-RAN Architecture

C-RAN is designed to be applicable to most typical RAN scenarios, from macro cell to femtocell. As shown in Fig. 1-1, it is composed of the baseband unit (BBU), optical

transmission network (OTN), and remote radio head (RRH). The BBU acts as a digital unit implementing the base station functionality from baseband processing to packet processing, while the RRHs perform radio functions, including frequency conversion, amplification, and A/D and D/A conversion. The RRHs send/receive digitalized signals to/from the BBU pool via optical fiber, and antennas are equipped with RRHs to transmit/ receive radio frequency (RF) signals. By placing numerous BBUs in a central physical pool while distributing RRHs according to RF strategies, operators can dynamically employ a real-time virtualization technology that maps radio signals from/to one RRH to any BBU processing entity in the pool. The benefits of C-RAN lie in the following four areas:

Reduced Cost: C-RAN aggregates computation resources in a few big rooms and leaves simpler functions in RRHs, thus saving a lot of operation and management cost. C-RAN makes equipment more effectively shared, such as GPS and transmission devices, thus reducing capital expenditure. Load balancing and scalability can be well achieved through virtualization, thus reducing waste of resources.

Better Energy Efficiency: C-RAN frees up individual BSs from the commitment of providing 24/7 services. All processing functionalities are implemented in a remote data-center. Power consumption and load congestion can be reduced by dynamically allocating processing capability and migrating tasks in the BS pool, and several BSs can be turned to low power or even be shut down selectively. Operators only need to install new RRHs connecting with the BBU pool to cover more service areas or split the cell for higher capacity.

Capacity Improvement: In C-RAN, BSs can work together in a large physical BBU pool and they can easily share the signaling, traffic data and channel state information (CSI) of active UE's in the system. It is much easier to implement joint processing & scheduling to mitigate inter-cell interference (ICI) and improve spectral efficiency.

Smart Internet Traffic Offload: Through enabling the smart breakout technology in C-RAN, the growing internet traffic from smart phones and other portable devices, can be offloaded from the core network of operators. The benefits are as follows: reduced back-haul traffic and cost; reduced core network traffic and gateway upgrade cost; reduced

latency to the users; differentiating service delivery quality for various applications. The service overlapping the core network also supplies a better experience to users.

Based on the recent developments in cloud computing and software defined radio (SDR) techniques, C-RAN is able to use general-purpose processors (**GPPs**) with multicore and multithread techniques to implement virtualized and centralized baseband and protocol processing.

In order to reduce power consumption and improve processing capability, hardware accelerators are preferred for computation-intensive tasks even in C-RAN, e.g. **Turbo decoders** (which are our devices of concern), FFT, and MIMO decoders. In order to use these hardware accelerators efficiently and flexibly in the C-RAN environment, challenging problems need to be addressed. One is a high-throughput interface to facilitate data exchange between the cloud platform and the accelerators pool. The **PCIe** interface is a good candidate and is the one used in our project [3].

1.2 Baseband Processing

Baseband refers to the original frequency range of a transmission signal before it is modulated. A **baseband unit (BBU)** is a unit that processes baseband in telecomm systems. A BBU has the following characteristics: modular design, small size, low power consumption and can be easily deployed.

A BBU in a cellular telephone cell site is comprised of a **digital signal processor** to process forward voice signals for transmission to a mobile unit and to process reverse voice signals received from the mobile unit.

In Baseband processing, discrete information is communicated with specific symbols selected from a finite set of symbols. A series of pulses forms a pulse train that carries the full message. Prior to transmission, especially in radio systems, these pulses are shaped to limit their high frequency content so as to minimize crosstalk with adjacent communication channels. During transmission through a bandlimited channel, pulses are dispersed (spread) in time and can overlap with each other giving rise to intersymbol interference (ISI). When the RF modulated pulses reach the receiver, dispersion and other distortions can be partially compensated with an equalizer.

An information source generates messages bearing information to be transmitted. The messages can be words, code symbols etc. The output of the information source is converted to a sequence of symbols from a certain alphabet. Most often binary symbols are transmitted.

The output of the information source is in general not suitable for transmission as it might contain too much redundancy. For efficiency reasons, the source encoder is designed to convert the source output sequence into a sequence of binary digits with minimum redundancy. If the source encoder generates R_b bits per second (bps), R_b is called the data rate.

The channel impairments cause errors in the received signal. The **channel encoder** is incorporated in the system to add redundancy to the information sequence. This redundancy is used to minimize transmission errors. The channel encoder assigns to each message of k digits a longer message of n digits called a codeword.

1.3 Forward error correction (FEC)

In 1948, Shannon [4] demonstrated in a landmark paper that, by proper encoding of the information, errors induced by a noisy channel or storage medium can be reduced to any desired level without sacrificing the rate of information transmission or storage, as long as the information rate is less than the capacity of the channel. Since Shannon's work much effort has been expended on the problem of devising efficient encoding and decoding methods for error control in a noisy environment. Recent developments have contributed toward achieving the reliability required by today's high-speed digital systems, and the use of coding for error control has become an integral part in the design of modern communication and digital storage systems. [5].

The primary function of an error control encoder-decoder pair (also known as a codec) is to enhance the reliability of message during transmission of information carrying symbols through a communication channel. An error control code can also ease the design process of a digital transmission system in multiple ways such as the following:

a) The transmission power requirement of a digital transmission scheme can be reduced by the use of an error control codec. This aspect is exploited in the design of most of the

modern wireless digital communication systems such as a cellular mobile communication system.

b) Even the size of a transmitting or receiving antenna can be reduced by the use of an error control codec while maintaining the same level of end-to-end performance

c) Access of more users to same radio frequency in a multi-access communication system can be ensured by the use of error control technique [example: cellular CDMA].

d) Jamming margin in a spread spectrum communication system can be effectively increased by using suitable error control technique. Increased jamming margin allows signal transmission to a desired receiver in battlefield and elsewhere even if the enemy tries to drown the signal by transmitting high power in-band noise.

Forward error correction (FEC) codes have long been a powerful tool in the advancement of information storage and transmission. By introducing meaningful redundancy (FEC codes) into a stream of information, systems gain the ability to not only detect data errors, but also correct them. With this ability, the systems can run on less power, operate at longer distances, and decrease the need for costly retransmissions

.

The encoding operation for a (n,k) error control code is a kind of mapping of sequences, chosen from a k-dimensional subspace to a larger, n-dimensional vector space of n-tuples defined over a finite field and with $n > k$.

Decoding refers to a reverse mapping operation for estimating the probable information sequence from the knowledge of the received coded sequence. The code rate or 'coding efficiency' R of the code is defined as:

$$R = \frac{I_{in}}{I_{out}}$$

Where I_{in} and I_{out} denote the lengths of input and output sequences respectively. The code rate is a dimensionless proper fraction.

A (7, 4) Hamming code is an example of a binary block code whose rate $R = 4/7$. For an error correction code, $R < 1.0$ and this implies that some additional information (in the form of 'parity symbol' or 'redundant symbol') is added during the process of encoding.

The two main categories of FEC codes are block codes and convolutional codes. The main difference between the two of them is memory of the encoder. In block codes each encoding operation depends on the current input message and is independent on previous encodings. That is, the encoder has no memory of history of past encodings. In contrast, for a convolutional code, each encoder output sequence depends not only on the current input message, but also on a number of past message blocks [6].

Chapter 2: Convolutional Codes and Decoding Methods

Most of this chapter is taken from an online course by MIT University [7] and [9].

2.1 Overview

Convolutional codes are a bit like the block codes in that they involve the transmission of parity bits that are computed from message bits. Unlike block codes in systematic form, however, the sender does not send the message bits followed by (or interspersed with) the parity bits; in a convolutional code, the sender sends only the parity bits. The encoder uses a sliding window to calculate $r > 1$ parity bits by combining various subsets of bits in the window. The combining is a simple addition in F_2 (i.e., modulo-2 addition, or equivalently, an exclusive-or operation).

Unlike a block code, the windows overlap and slide by 1, as shown in Figure 2.1 The size of the window, in bits, is called the code's constraint length. The longer the constraint length, the larger the number of parity bits that are influenced by any given message bit.

Because the parity bits are the only bits sent over the channel, a larger constraint length generally implies a greater resilience to bit errors. The trade-off, though, is that it will take considerably longer to decode codes of long constraint length, so one can't increase the constraint length arbitrarily and expect fast decoding. If a convolutional code that produces r parity bits per window and slides the window forward by one bit at a time, its rate (when calculated over long messages) is $1/r$. The greater the value of r , the higher the resilience of bit errors, but the trade-off is that a proportionally higher amount of communication bandwidth is devoted to coding overhead. In practice, we would like to pick r and the constraint length to be as small as possible while providing a low enough resulting probability of a bit error.

We will use K (upper case) to refer to the constraint length, a somewhat unfortunate choice because we have used k (lower case) to refer to the number of message bits that get encoded to produce coded bits.

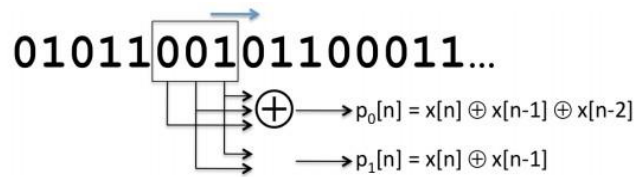


Figure 2-1 Convolutional code with two parity bits per message bit ($r = 2$, $K=3$)

Because we will rarely refer to a “block” of size k while talking about convolutional codes, we hope that this notation won’t cause confusion.

Armed with this notation, we can describe the encoding process succinctly. The encoder looks at K bits at a time and produces r parity bits according to carefully chosen functions that operate over various subsets of the K bits. One example is shown in Figure 2-1, which shows a scheme with $K = 3$ and $r = 2$ (the rate of this code, $1/r = 1/2$). The encoder spits out r bits, which are sent sequentially, slides the window by 1 to the right, and then repeats the process.

2.2 Parity Equations

The example in Figure 2-1 shows one example of a set of parity equations, which govern the way in which parity bits are produced from the sequence of message bits x . In this example, the equations are as follows (all additions are in F_2):

$$p_0[n] = x[n] + x[n-1] + x[n-2]$$

$$p_1[n] = x[n] + x[n-1]$$

An example of parity equations for a rate $1/3$ code is:

$$p_0[n] = x[n] + x[n-1] + x[n-2]$$

$$p_1[n] = x[n] + x[n-1]$$

$$p_2[n] = x[n] + x[n-2]$$

In general, one can view each parity equation as being produced by composing the message bits, X , and a generator polynomial, g . In the first example above, the generator polynomial coefficients are $(1, 1, 1)$ and $(1, 1, 0)$, while in the second, they are $(1, 1, 1)$, $(1, 1, 0)$, and $(1, 0, 1)$.

We denote by g_i the K -element generator polynomial for parity bit p_i . We can then write p_i as follows:

$$p_i[n] = \left(\sum_{j=0}^{K-1} g_i[j]x[n-j] \right) \text{ mod } 2$$

The form of the above equation is a convolution of g and x hence the term “convolutional code”. The number of generator polynomials is equal to the number of generated parity bits, r , in each sliding window.

2.3 Views of the Convolutional Encoder

We now describe two views of the convolutional encoder, which we will find useful in better understanding convolutional codes and in implementing the encoding and decoding procedures. The first view is in terms of a block diagram, where one can construct the mechanism using shift registers that are connected together. The second is in terms of a state machine, which corresponds to a view of the encoder as a set of states with well-defined transitions between them. The state machine view will turn out to be extremely useful in figuring out how to decode a set of parity bits to reconstruct the original message bits.

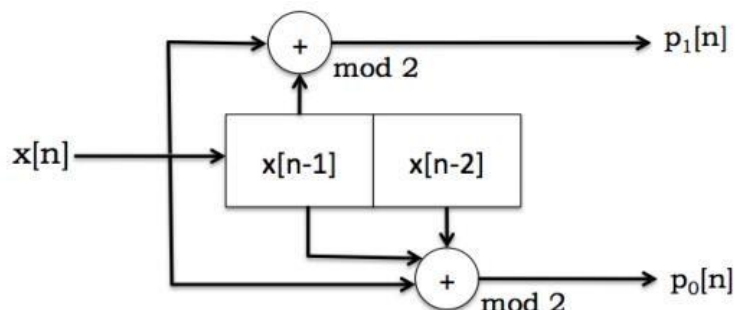


Figure 2-2 Block diagram view of convolutional coding with shift registers.

2.3.1 Block Diagram View

Figure 2-2 shows the same encoder as Figure 2-1 and its Equations in the form of a block diagram. The $x[n-i]$ values (here there are two) are referred to as the state of the encoder. The way to think of this block diagram is as a “black box” that takes message bits in and spits out parity bits. Input message bits, $x[n]$, arrive on the wire from the left. The box calculates the parity bits using the incoming bits and the state of the encoder (the $k-1$ previous bits; 2 in this example). After the r parity bits are produced, the state of the encoder shifts by 1, with $x[n]$ taking the place of $x[n-1]$, $x[n-1]$ taking the place of $x[n-2]$, and so on, with $x[n-K+1]$ being discarded. This block diagram is directly amenable to a hardware implementation using shift registers.

2.3.2 State Machine View

Another useful view of convolutional codes is as a state machine, which is shown in Figure 2-3 for the same example that we have used (Figure 2-1). The state machine for a convolutional code is identical for all codes with a given constraint length, K , and the number of states is always $2^K - 1$. Only the p_i labels change depending on the number of generator polynomials and the values of their coefficients. Each state is labeled with $x[n-1] x[n-2] \dots x[n-K+1]$. Each arc is labeled with $x[n]/p_0 p_1$. In this example, if the message is 101100, the transmitted bits are 11 11 01 00 01 10. This state machine view is an elegant way to explain what the transmitter does, and also what the receiver ought to do to decode the message, as we now explain. The transmitter begins in the initial state labeled “STARTING STATE” in Figure (2-3) and processes the message one bit at a time. For each message bit, it makes the state transition from the current state to the new one depending on the value of the input bit, and sends the parity bits that are on the corresponding arc. The receiver, of course, does not have direct knowledge of the transmitter’s state transitions. It only sees the received sequence of parity bits, with possible corruptions. Its task is to determine the best possible sequence of transmitter states that could have produced the parity bit sequence. This task is called decoding, which we will be introduced next, and then study in more detail later.

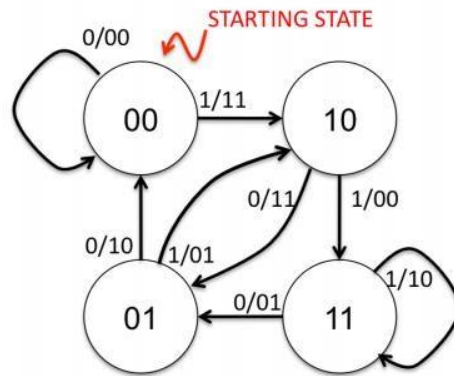


Figure 2-3 State machine view of convolutional coding.

2.4 Trellis Structure

The trellis is a structure derived from the state machine that will allow us to develop an efficient way to decode convolutional codes. The state machine view shows what happens at each instant when the sender has a message bit to process, but doesn't show how the system evolves in time. The trellis is a structure that makes the time evolution explicit.

An example is shown in Figure 2-4. Each column of the trellis has the set of states; each state in a column is connected to two states in the next column, the same two states in the state diagram. The top link from each state in a column of the trellis shows what gets transmitted on a "0", while the bottom shows what gets transmitted on a "1". The picture shows the links between states that are traversed in the trellis given the message 101100. We can now think about what the decoder needs to do in terms of this trellis. It gets a sequence of parity bits, and needs to determine the best path through the trellis that is, the sequence of states in the trellis that can explain the observed, and possibly corrupted, sequence of received parity bits.

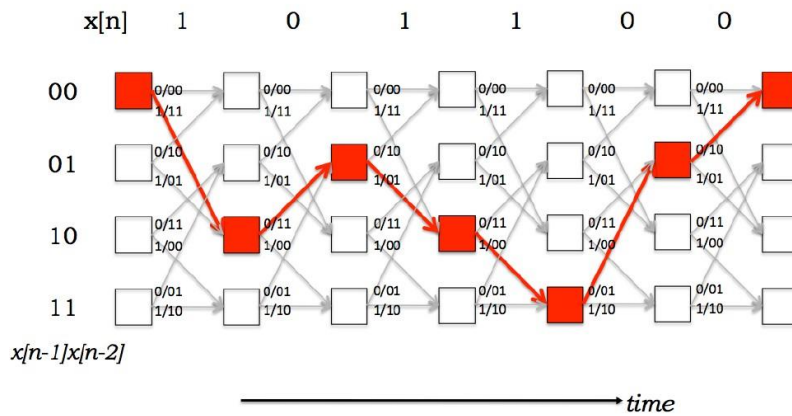


Figure 2-4 Trellis Diagram.

2.5 Channel Models

In any communications system, it is important to understand how a signal is affected by the transmission channel it encounters. We focus on two common models for communication channels that are used when evaluating the performance of convolutional codes [8].

2.5.1 Binary Symmetric Channel

The binary symmetric channel (BSC) is a channel model that involves only the transmission of bits, defining a hard line to determine between a “0” or “1”. For each unit of time, a bit is transmitted with probability of error p and probability of success $1-p$. The value p is known as the crossover probability, because it represents the probability that a bit “crosses over” from “0” to “1” or “1” to “0”, which can be seen in Figure 2-5 A BSC transmission can be modeled as:

$$r = s + n$$

Where r contains received bits, s contains transmitted bits, and n contains possible bit errors. The sequences r , s , and n have the same length and are indexed in discrete-time by i . If there is a bit error at time i , n_i will be 1, otherwise it is 0. The sequence n is an independent and identically distributed Bernoulli random process. The $+$ operator, also known as xor, applies any bit error at n_i by toggling the value of s_i . In other words, corresponds to modulo-2 addition.

When using a BSC model, the comparison between a transmitted sequence and a received sequence is most often done using Hamming distance. The Hamming distance between the sequences s and r is defined as the number of positions in which the corresponding elements are different. When a sequence is received over a BSC, it is compared with a group of possible transmitted sequences to determine which one it most closely resembles. Typically, the winner is the one with the shortest Hamming distance to the received sequence.

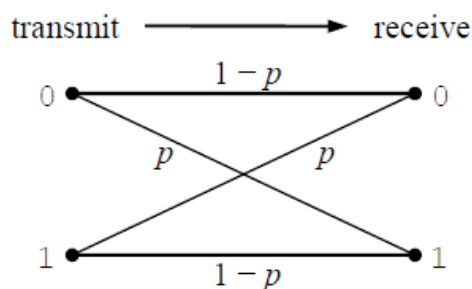


Figure 2-5 Binary Symmetric Channel Model

2.5.2 Additive White Gaussian Noise Channel

The additive white Gaussian noise (AWGN) channel is one of the most common mathematical models for a communication channel. As the name suggests, it assumes that a communication link is primarily affected by Gaussian noise. The AWGN model can be applied to many physical channels, which makes it very useful when evaluating the performance of a system.

In order for a data sequence to be physically transmitted, it must encounter some form of modulation. One simple modulation scheme is binary phase-shift keying (BPSK), where bits are mapped to antipodal values ($+A$ or $-A$), with “0” \rightarrow ($-A$) and “1” \rightarrow ($+A$). Once a bit sequence has been modulated, it is sent over the channel, where it encounters additive Gaussian noise. A mathematical representation of this is:

$$r = s + n$$

Where r contains noisy received values, s contains transmitted antipodal bit values, and n contains noise values. This is similar to the BSC model in the above section, however

each sequence is real-valued and the + operator performs addition of reals. Each value in n is an independent Gaussian random variable with mean $\mu_n = 0$ and variance $\sigma_n^2 = N_0/2$. E_s is the energy in a symbol (for BPSK, $E_s = A^2$) and $N_0/2$ is the two-sided power spectral density (PSD) of the noise. It is typical to normalize the symbol energy to $E_s = 1$, therefore making $A = 1$.

The performance of a digital communication system is often quantified with the bit error rate (BER) versus the signal-to-noise ratio (SNR). The SNR is typically defined as $E_b = N_0$, where E_b is the energy in an information bit. In an uncoded BPSK system, each transmitted symbol corresponds to one information bit, or $E_s = E_b$. In a coded BPSK system, multiple transmitted symbols may correspond to a single information bit, making the relationship dependent on the code rate R , or $E_s = RE_b$. Substituting this into the equation for the variance of the noise yields $\sigma_n^2 = 1/2RE_b$, assuming $E_s = 1$.

Using the standard deviation σ_n to scale a standard normal (mean 0, variance 1) random variable allows n to be generated with a desired value of $E_b = N_0$. When using the AWGN Channel model, the optimal comparison between a transmitted sequence and a received sequence is done using Euclidean distance. The squared Euclidean distance between the sequences s and r is defined as:

$$d_E^2(s, r) = \sum_{i=0}^{n-1} (s[i] - r[i])^2$$

When a sequence is received over an AWGN channel, it is compared with the set of possible transmitted sequences to determine which one it most closely resembles.

The winner is the one that is closest in squared Euclidean distance to the received sequence.

2.6 Decoding Convolutional Codes

Most of this part is taken from an online course by MIT University [9].

At the receiver, we have a sequence of voltage samples corresponding to the parity bits that the transmitter has sent. For simplicity, and without loss of generality, we will assume that the receiver picks a suitable sample for the bit, or averages the set

of samples corresponding to a bit, digitizes that value to a “0” or “1” by comparing to the threshold voltage (the demapping step), and propagates that bit decision to the decoder.

Thus, we have a *received bit sequence*, which for a convolutionally-coded stream corresponds to the stream of parity bits. If we decode this received bit sequence with no other information from the receiver’s sampling and demapper, then the decoding process is termed **hard-decision decoding** (“hard decoding”). If, instead (or in addition), the decoder is given the stream of voltage samples and uses that “analog” information (in digitized form, using an analog-to-digital conversion) in decoding the data, we term the process **soft-decision decoding** (“soft decoding”).

The Viterbi decoder can be used in either case. Intuitively, because hard-decision decoding makes an early decision regarding whether a bit 0 or 1 is, it throws away information in the digitizing process. It might make a wrong decision, especially for voltages near the threshold, introducing a greater number of bit errors in the received bit sequence. Although it still produces the most likely transmitted sequence given the received bit sequence, by introducing additional errors in the early digitization, the overall reduction in the probability of bit error will be smaller than with soft decision decoding. But it is conceptually easier to understand hard decoding, so we will start with that, before going on to soft decoding.

As mentioned before, the trellis provides a good framework for understanding the decoding procedure for convolutional codes (Figure 2-6). Suppose we have the entire trellis in front of us for a code, and now receive a sequence of digitized bits (or voltage samples). If there are no errors, then there will be some path through the states of the trellis that would exactly match the received sequence. That path (specifically, the concatenation of the parity bits “spit out” on the traversed edges) corresponds to the transmitted parity bits. From there, getting to the original encoded message is easy because the top arc emanating from each node in the trellis corresponds to a “0” bit and the bottom arrow corresponds to a “1” bit.

When there are bit errors, what can we do? As explained earlier, finding the most likely transmitted message sequence is appealing because it minimizes the probability of a bit error in the decoding. If we can come up with a way to capture the errors

introduced by going from one state to the next, then we can accumulate those errors along a path and come up with an estimate of the total number of errors along the path. Then, the path with the smallest such accumulation of errors is the path we want, and the transmitted message sequence can be easily determined by the concatenation of states explained above.

To solve this problem, we need a way to capture any errors that occur in going through the states of the trellis, and a way to navigate the trellis without actually materializing the entire trellis (i.e., without enumerating all possible paths through it and then finding the one with smallest accumulated error). The Viterbi decoder solves these problems.

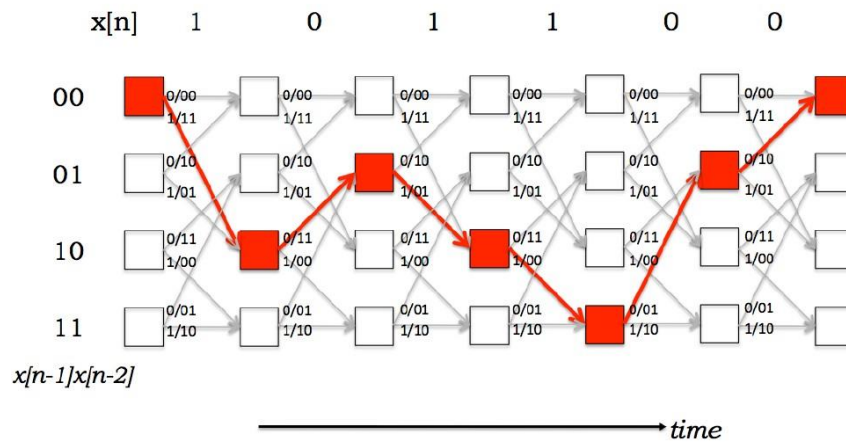


Figure 2-6 Trellis Structure.

2.6.1 The Viterbi Decoder

The decoding algorithm uses two metrics: the **branch metric** (BM) and the **path metric** (PM). The branch metric is a measure of the “distance” between what was transmitted and what was received, and is defined for each arc in the trellis. In hard decision decoding, where we are given a sequence of digitized parity bits, the branch metric is the Hamming distance between the expected parity bits and the received ones. An example is shown in Figure 2-7, where the received bits are 00. For each state transition, the number on the arc shows the branch metric for that transition. Two of the branch metrics are 0, corresponding to the only states and transitions where the corresponding Hamming distance is 0. The other non-zero branch metrics correspond to cases when there are bit errors.

The path metric is a value associated with a state in the trellis (i.e., a value associated with each node). For hard decision decoding, it corresponds to the Hamming distance with respect to the received parity bit sequence over the most likely path from the initial state to the current state in the trellis. By “most likely”, we mean the path with smallest Hamming distance between the initial state and the current state, measured over all possible paths between the two states. The path with the smallest Hamming distance minimizes the total number of bit errors, and is most likely when the BER is low.

The key insight in the Viterbi algorithm is that the receiver can compute the path metric for a (state, time) pair incrementally using the path metrics of previously computed states and the branch metrics.

2.6.1.1 Computing the Path Metric

Suppose the receiver has computed the path metric $PM[s, i]$ for each states at time step i (recall that there are 2^{K-1} states, where K is the constraint length of the convolutional code). In hard decision decoding, the value of $PM[s, i]$ is the total number of bit errors detected when comparing the received parity bits to the most likely transmitted message, considering all messages that could have been sent by the transmitter until time step i (starting from state “00”, which we will take to be the starting state always, by convention).

Among all the possible states at time step i , the most likely state is the one with the smallest path metric. If there is more than one such state, they are all equally good possibilities.

Now, how do we determine the path metric at time step $i + 1$, $PM[s, i + 1]$, for each state s ? To answer this question, first observe that if the transmitter is at state s at time step $i + 1$, then it must have been in only one of two possible states at time step

- i. These two predecessor states, labeled α and β , are always the same for a given state. In fact, they depend only on the constraint length of the code and not on the parity functions. Figure 2-7 shows the predecessor states for each state (the other end of each arrow). For instance, for state 00, $\alpha = 00$ and $\beta = 01$; for state 01, $\alpha = 10$ and $\beta = 11$.

Any message sequence that leaves the transmitter in state s at time $i + 1$ must have

left the transmitter in state α or state β at time i . For example, in Figure 4.2, to arrive in state '01' at time $i + 1$, one of the following two properties must hold:

- The transmitter was in state "10" at time i and the i th message bit was a "0". If that is the case, then the transmitter sent '11' as the parity bits and there were two bit errors, because we received the bits "00". Then, the path metric of the new state, $PM["01", i + 1]$ is equal to $PM["10", i] + 2$, because the new state is "01" and the corresponding path metric is larger by 2 because there are 2 errors.
- The other (mutually exclusive) possibility is that the transmitter was in state "11" at time i and the i th message bit was a "0". If that is the case, then the transmitter sent "01" as the parity bits and there was one-bit error, because we received "00". The path metric of the new state, $PM["01", i + 1]$ is equal to $PM["11", i] + 1$. Formalizing the above intuition, we can see that:

$$PM[s, i + 1] = \min (PM [\alpha, i] + BM [\alpha \rightarrow s], PM [\beta, i] + BM [\beta \rightarrow s])$$

In the decoding algorithm, it is important to remember which arc corresponds to the minimum, because we need to traverse this path from the final state to the initial one keeping track of the arcs we used, and then finally reverse the order of the bits to produce the most likely message.

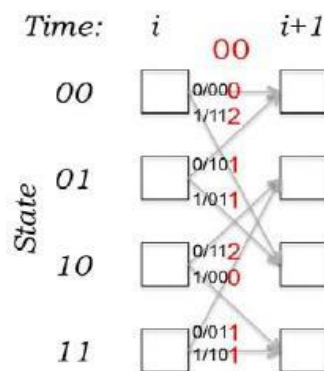


Figure 2-7 The branch metric for hard decision decoding.

2.6.1.2 Finding the Most Likely Path

We can now describe how the decoder finds the maximum-likelihood path. Initially, state “00” has a cost of 0 and the other $2^{K-1} - 1$ states have a cost of ∞ . The main loop of the algorithm consists of two main steps: first, calculating the branch metric for the next set of parity bits, and second, computing the path metric for the next column. The path metric computation may be thought of as an add-compare-select procedure:

- Add the branch metric to the path metric for the old state.
- Compare the sums for paths arriving at the new state (there are only two such paths to compare at each new state because there are only two incoming arcs from the previous column).
- Select the path with the smallest value, breaking ties arbitrarily. This path corresponds to the one with fewest errors.

Figure 2-8 and 2-9 shows the decoding algorithm in action from one time step to the next. This example shows a received bit sequence of 11 10 11 00 01 10 and how the receiver processes it. The fourth picture from the top shows all four states with the same path metric. At this stage, any of these four states and the paths leading up to them are most likely transmitted bit sequences (they all have a Hamming distance of 2). The bottom-most picture shows the same situation with only the survivor paths shown. A survivor path is one that has a chance of being the maximum-likelihood path; there are many other paths that can be pruned away because there is no way in which they can be most likely. The reason why the Viterbi decoder is practical is that the number of survivor paths is much, much smaller than the total number of paths in the trellis. Another important point about the Viterbi decoder is that future knowledge will help it break any ties, and in fact may even cause paths that were considered “most likely” at a certain time step to change. Figure 2-9 continues the example in Figure 2-8, proceeding until all the received parity bits are decoded to produce the most likely transmitted message, which has two bit errors.

2.7 Soft-Decision Decoding

Hard decision decoding digitizes the received voltage signals by comparing it to a threshold, before passing it to the decoder. As a result, we lose information: if the voltage was 0.500001, the confidence in the digitization is surely much lower than if the voltage was 0.999999. Both are treated as “1”, and the decoder now treats them the same way, even though it is overwhelmingly more likely that 0.999999 is a “1” compared to the other value. Soft-decision decoding (also sometimes known as “soft input Viterbi decoding”) builds on this observation. It does not digitize the incoming samples prior to decoding. Rather, it uses a continuous function of the analog sample as the input to the decoder. For example, if the expected parity bit is 0 and the received voltage is 0.3 V, we might use 0.3 (or 0.32, or some such function) as the value of the “bit” instead of digitizing it. For technical reasons that will become apparent later, an attractive soft decision metric is the square of the difference between the received voltage and the expected one. If the convolutional code produces p parity bits, and the p corresponding analog samples are $v = v_1, v_2, \dots, v_p$, one can construct a soft decision branch metric as follows:

$$BM_{\text{soft}} = \sum_{i=1}^p (u_i - v_i)^2 \quad (\text{I})$$

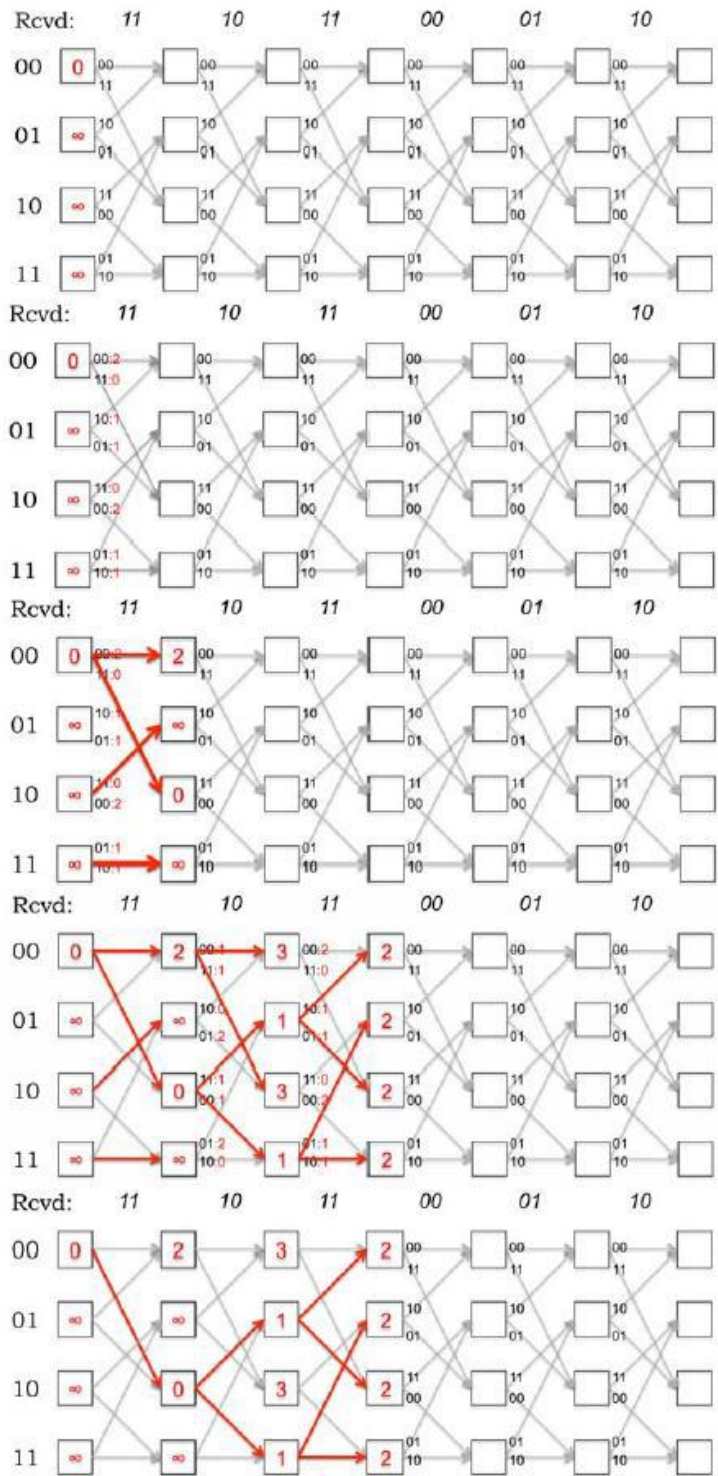


Figure 2-8 The Viterbi Decoder in Action

This picture shows four time steps, the bottom-most picture is the same as the one just before it, but with only the survivor paths shown.

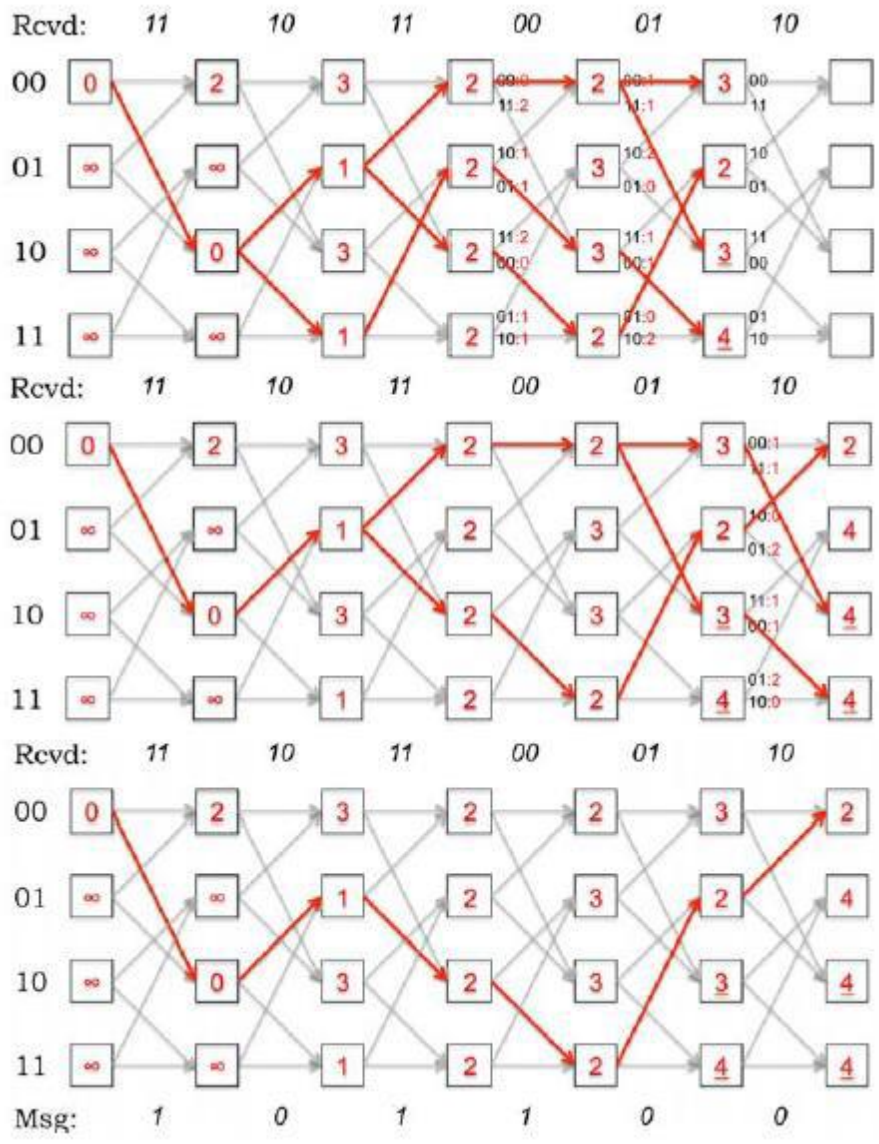


Figure 2-9 The Viterbi decoder in Action continued

The decoded message is shown. To produce this message, start from the final state with smallest path metric and work backwards, and then reverse the bits. At each state during the forward pass, it is important to remember the arc that got us to this state, so that the backward pass can be done properly.

Where $u = u_1, u_2, \dots, u_p$ are the expected p parity bits (each a 0 or 1). Figure 2-10 shows the soft decision branch metric for $p = 2$ when u is 00.

With soft decision decoding, the decoding algorithm is identical to the one previously described for hard decision decoding, except that the branch metric is no longer an integer Hamming distance but a positive real number (if the voltages are all between 0 and 1, then the branch metric is 0 and 1 as well).

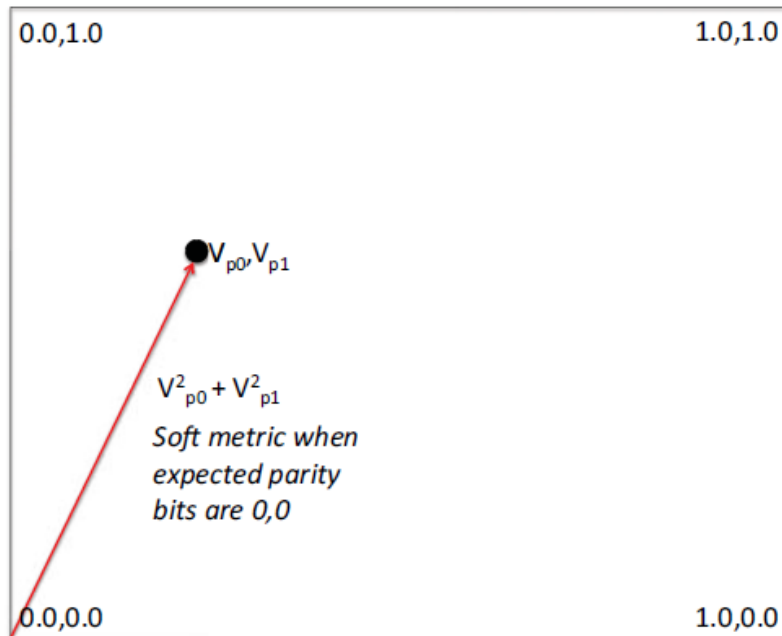


Figure 2-10 Branch Metric For Soft-Decision

It turns out that this soft decision metric is closely related to the probability of the decoding being correct when the channel experiences additive Gaussian noise. First, let's look at the simple case of 1 parity bit (the more general case is a straightforward extension). Suppose the receiver gets the i th parity bit as v_i volts. (In hard decision decoding, it would decode as 0 or 1 depending on whether v_i was smaller or larger than 0.5) What is the probability that v_i would have been received given that bit u_i (either 0 or 1) was sent? With zero-mean additive Gaussian noise, the PDF of this event is given by

$$f(v_i|u_i) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{d_i^2}{2\sigma^2}} \quad (\text{II})$$

Where $d_i = \sqrt{2}$ if $u_i = 0$ and $d_i = (\sqrt{2} - 1)\sqrt{2}$ if $u_i = 1$.

The log likelihood of this PDF is proportional to $-d_i^2$. Moreover, along a path, the PDF of the sequence $V = v_1, v_2, \dots, v_p$ being received given that a codeword $U = u_1, u_2, \dots, u_p$ was sent, is given by the product of a number of terms each resembling Eq. (II). The logarithm of this PDF for the path is equal to the sum of the individual log likelihoods, and is proportional to $-\sum_i d_i^2$. But that's precisely the negative of the branch metric we defined before, which the Viterbi decoder minimizes along the different possible paths! Minimizing this path metric is identical to maximizing the log likelihood along the different paths, implying that the soft decision decoder produces the most likely path that is consistent with the received voltage sequence.

This direct relationship with the logarithm of the probability is the reason why we chose the sum of squares as the branch metric in Eq. (I). A different noise distribution (other than Gaussian) may entail a different soft decoding branch metric to obtain an analogous connection to the PDF of a correct decoding.

Chapter 3: Turbo Codes

This chapter describes the turbo encoder/decoder and their structure in details [10],[11].

3.1 Turbo Encoder

The fundamental turbo code encoder is built using two identical recursive systematic convolutional (RSC) codes with parallel concatenation [12].

An RSC encoder is typically $r = 1/2$ and is termed a component encoder. The two component encoders are separated by an interleaver. Only one of the systematic outputs from the two component encoders is used, because the systematic output from the other component encoder is just a permuted version of the chosen systematic output.

Figure 3-1 shows the fundamental turbo code encoder with $r = 1/3$. The first RSC encoder outputs the systematic c_1 , and recursive convolutional c_2 sequences while the second RSC encoder discards its systematic sequence and only outputs the recursive convolutional c_3 sequence.

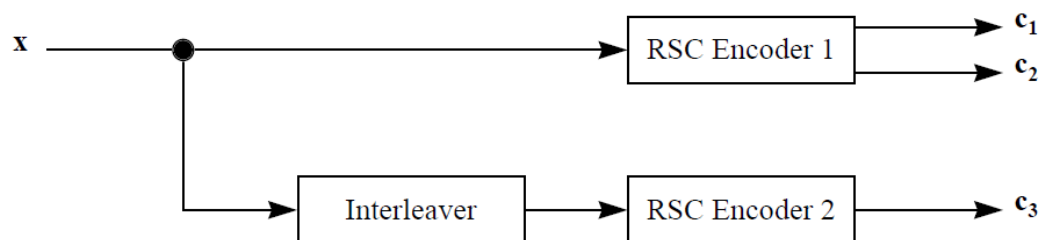


Figure 3-1: Fundamental turbo code encoder.

3.1.1 Recursive Systematic Convolutional (RSC) Encoder

The recursive systematic convolutional (RSC) encoder is obtained from the non recursive non-systematic (conventional) convolutional encoder by feeding back one

of its encoded outputs to its input. Figure 3-2 shows a conventional convolutional encoder.

The conventional convolutional encoder is represented by the generator sequences $g_1 = [111]$ and $g_2 = [101]$ and can be equivalently represented in a more compact form as $G = [g_1, g_2]$. The RSC encoder of this conventional convolutional encoder is represented as $G = [1, g_2 / g_1]$ where the first output (represented by g_1) is fed back to the input. In the above representation, 1 denotes the systematic output, g_2 denotes the feedforward output, and g_1 is the feedback to the input of the RSC encoder. Figure 3-3 shows the resulting RSC encoder.

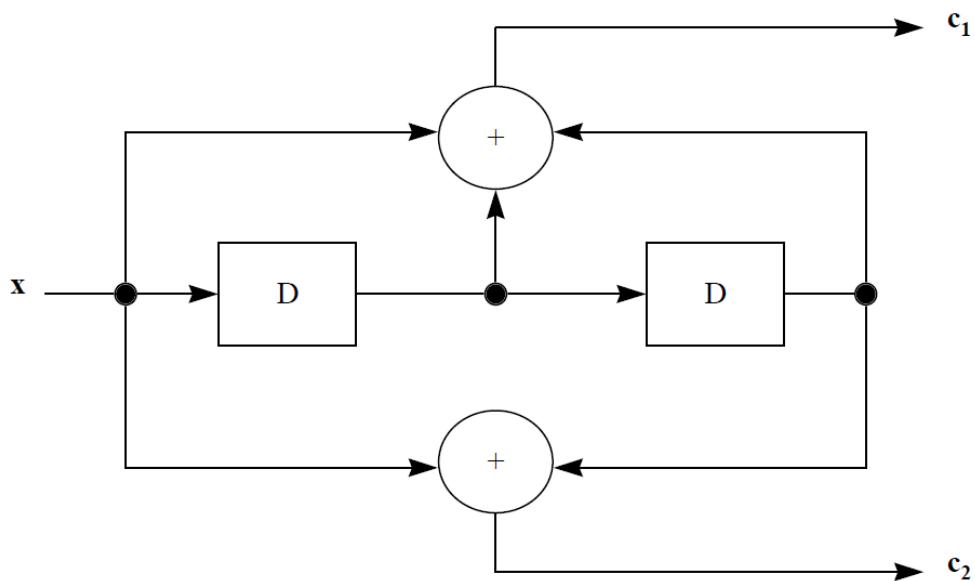


Figure 3-2: Conventional convolutional encoder with $r=1/2$ and $K=3$.

It was suggested in [13] that good codes can be obtained by setting the feedback of the RSC encoder to a primitive polynomial, because the primitive polynomial generates maximum-length sequences which adds randomness to the turbo code.

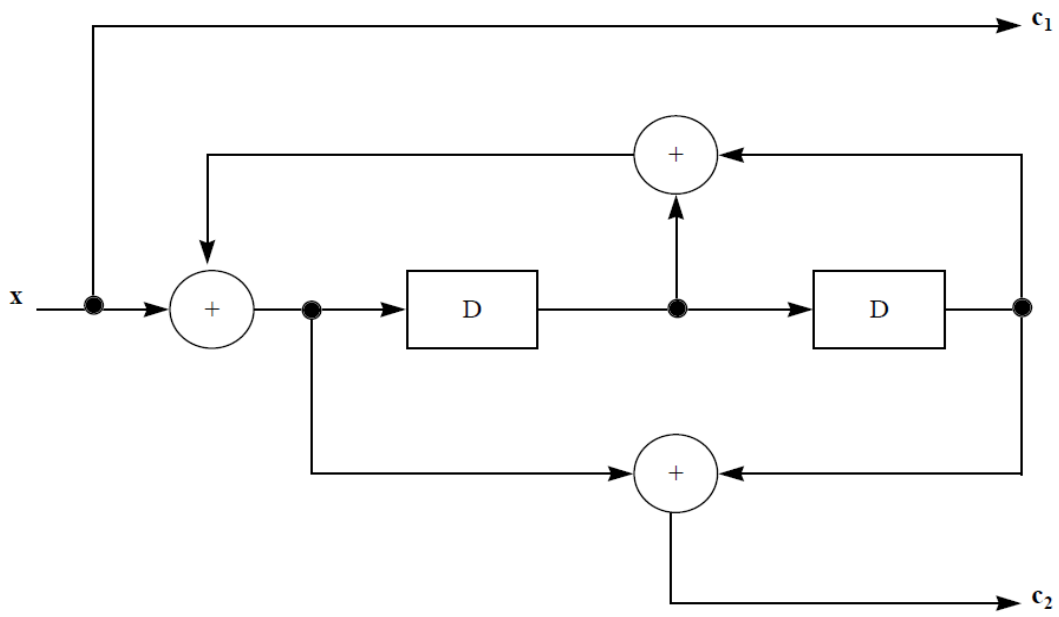


Figure 3-3: The RSC encoder obtained from Figure 3.2 with $r=1/2$ and $K=3$.

3.1.2 Trellis Termination

For the conventional convolutional encoder, the trellis is terminated by inserting $m = K-1$ additional zero bits after the input sequence. These additional bits drive the conventional convolutional encoder to the all-zero state (trellis termination). However, this strategy is not possible for the RSC encoder due to the feedback. The additional termination bits for the RSC encoder depend on the state of the encoder and are very difficult to predict. Furthermore, even if the termination bits for one of the component encoders are found, the other component encoder may not be driven to the all zero state with the same m termination bits due to the presence of the interleaver between the component encoders. Figure 3-4 shows a simple strategy that has been developed, which overcomes this problem.

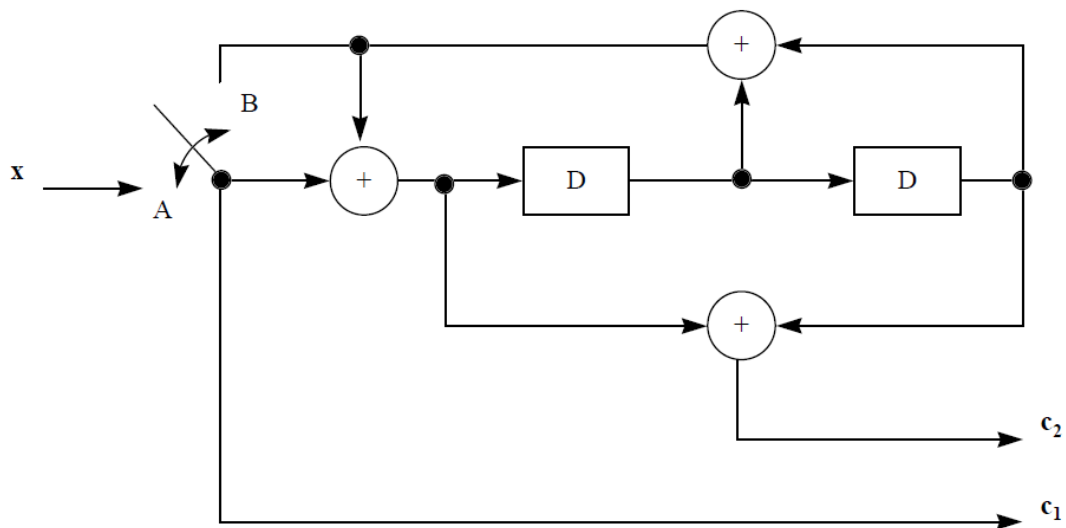


Figure 3-4: Trellis termination strategy for RSC encoder

For encoding the input sequence, the switch is turned on to position A and for terminating the trellis, the switch is turned on to position B.

3.1.3 Concatenation of Codes

A concatenated code is composed of two separate codes that are combined to form a larger code. There are two types of concatenation, namely serial and parallel concatenations. Figure 3-5 shows the serial concatenation scheme for transmission.

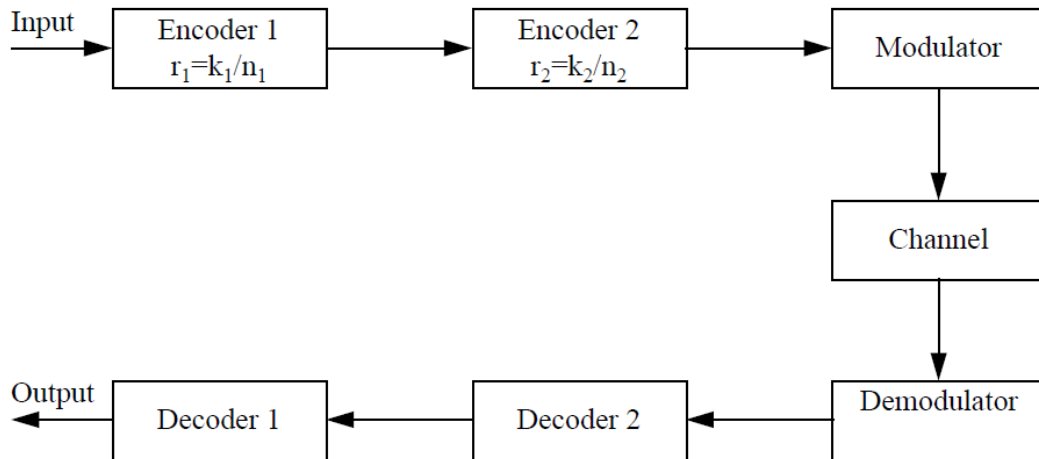


Figure 3-5: Serial concatenated code.

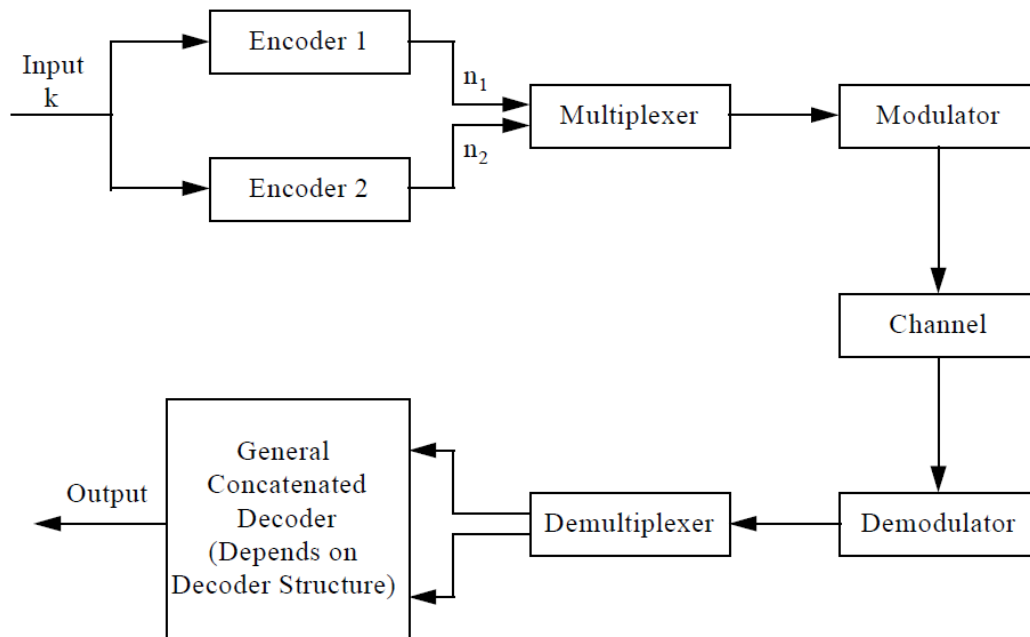


Figure 3-6: Parallel concatenated code.

The total code rate for serial concatenation is:

$$r_{\text{tot}} = \frac{K_1 K_2}{n_1 n_2}$$

Which is equal to the product of the two code rates.

Figure 3-6 shows the parallel concatenation scheme for transmission.

The total code rate for parallel concatenation is:

$$r_{\text{tot}} = \frac{K}{n_1 n_2}$$

For both serial and parallel concatenation schemes, an interleaver is often used between the encoders to improve burst error correction capacity or to increase the randomness of the code. Turbo codes use the parallel concatenated encoding scheme. However, the turbo code decoder is based on the serial concatenated decoding scheme. The serial concatenated decoders are used because they perform better than the parallel concatenated decoding scheme due to the fact that the serial concatenation scheme has the ability to share information between the concatenated decoders whereas the decoders for the parallel concatenation scheme are primarily decoding independently. Later it will be shown how the serial concatenated decoding scheme is implemented for a turbo code.

3.1.4 Interleaver Design

For turbo codes, an interleaver is used between the two component encoders. The interleaver is used to provide randomness to the input sequences. Also, it is used to increase the weights of the codewords as shown in Figure 3-7.

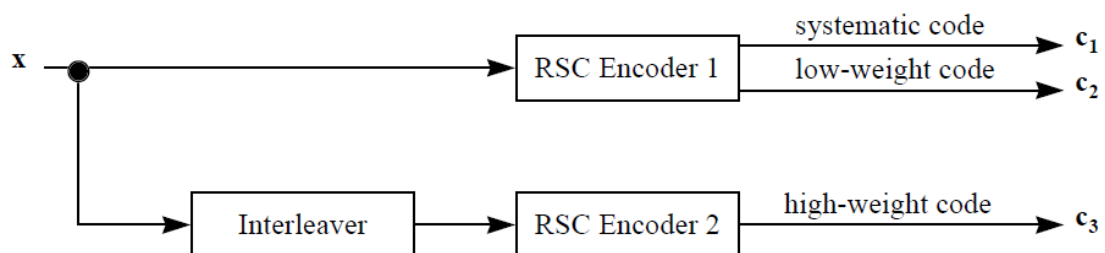


Figure 3-7: The interleaver increases the code weight for Encoder 2 compared to Encoder 1.

From Figure 3-7, the input sequence \mathbf{x} produces a low-weight recursive convolutional code sequence \mathbf{c}_2 for RSC Encoder 1. To avoid having RSC Encoder 2 produce another low-weight recursive output sequence, the interleaver permutes the input sequence \mathbf{x} to obtain a different sequence that hopefully produces a high-weight recursive convolutional code sequence \mathbf{c}_3 . Thus, the turbo code's code weight is moderate, combined from Encoder 1's low-weight code and Encoder 2's high-weight code. Figure 3-8 shows an illustrative example.

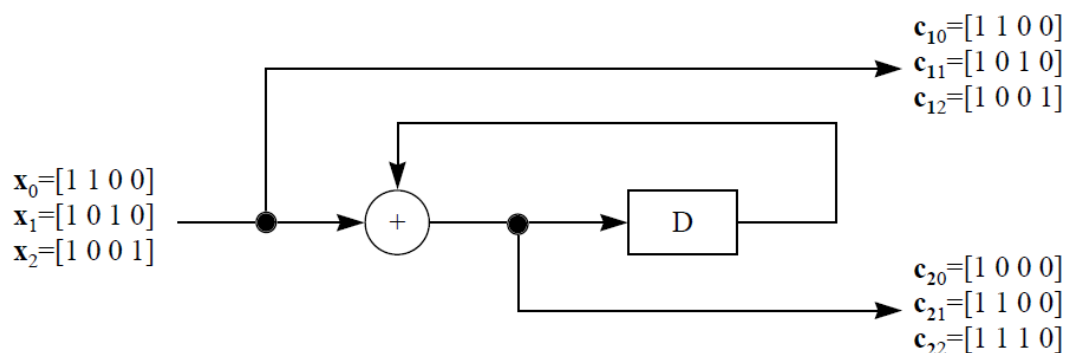


Figure 3-8: An illustrative example of an interleaver's capability.

Table 3.1: Input and Output Sequences for Encoder in Figure 3-8

	Input Sequence x_i	Output Sequence c_{1i}	Output Sequence c_{2i}	Codeword Weight i
$i = 0$	1 1 0 0	1 1 0 0	1 0 0 0	3
$i = 1$	1 0 1 0	1 0 1 0	1 1 0 0	4
$i = 2$	1 0 0 1	1 0 0 1	1 1 1 0	5

As it can be seen from Table 3.1, the codeword weight can be increased by utilizing an interleaver.

The interleaver affects the performance of turbo codes because it directly affects the distance properties of the code. By avoiding low-weight codewords, the BER of a turbo code can improve significantly. Thus, much research has been done on interleaver design.

3.1.5 LTE Turbo Encoder in the 3GPP Standard

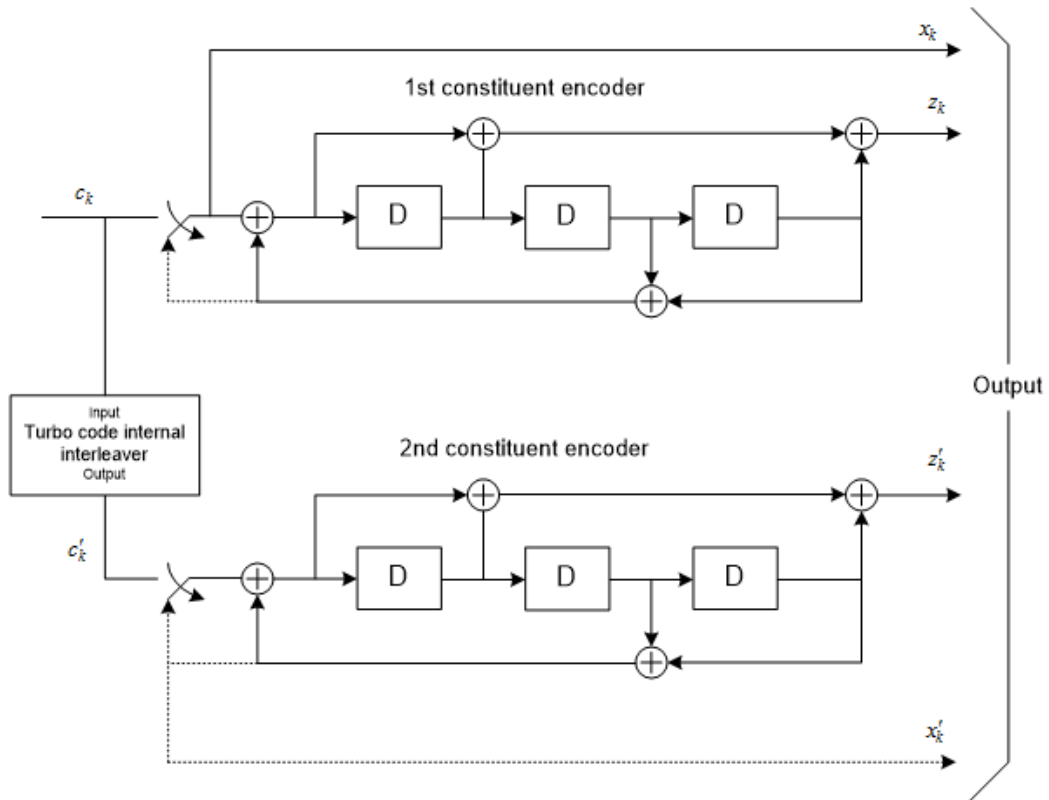


Figure 3-9: Structure of LTE Turbo Encoder.

The LTE Turbo Encoder scheme is Parallel Concatenated Convolution Code (PCCC). It comprises of two constraint length $K = 4$ (8 state) RSC encoders concatenated in parallel. The overall code rate is approximately $r = 1/3$. Figure 3-9 shows the LTE turbo encoder [14].

The two convolutional encoders used in the Turbo code are identical with generator polynomials:

$$g_0(D) = 1 + D^2 + D^3$$

$$g_1(D) = 1 + D + D^3$$

Where g_0 and g_1 are the feedback and feed forward generator polynomials respectively.

The transfer function of each constituent convolutional encoder is: $G(D) = \frac{g_0(D)}{g_1(D)}$

The data bits are transmitted together with the parity bits generated by two constituent convolutional encoders. Prior to encoding, both the convolutional encoders are set to all zero state, i.e., each shift register is filled with zeros. The turbo encoder consists of an internal interleaver which interleaves the input data bits $c_1, c_2 \dots c_K$ to $c'_1, c'_2 \dots c'_K$ which are then input to the second constituent encoder. Thus, the data is encoded by the first encoder in the natural order and by the second encoder after being interleaved. The systematic output of the second encoder is not used and thus the output of the turbo coder is serialized combination of the systematic bits X_k , parity bits from the first (upper) encoder Z_k and parity bits from the second encoder Z'_k for $k = 1, 2, \dots K$. So the transmitted sequence will be:

$$X_1, Z_1, Z'_1, X_2, Z_2, Z'_2 \dots X_K, Z_K, Z'_K$$

The size of the input data word may range from as few as 40 to as many as 6144 bits. If the interleaver size is equal to the input data size K the data is scrambled according to the interleaving algorithm, otherwise dummy bits are added before scrambling. After all the data bits K have been encoded, trellis termination is performed by passing tail bits from the constituent encoders bringing them to all zeros state. To achieve this, the switches in Figure 3-9 are moved in the down position. The input in this case is shown by dashed lines (input=feedback bit). Because of the interleaver, the states of both the constituent encoders will usually be different, so the tail bits will also be different and need to be dealt separately.

As constraint length $K = 4$ constituent convolutional encoders are used, so the transmitted bit stream includes not only the tail bits $\{X_{k+1}, X_{k+2}, X_{k+3}\}$ corresponding to the upper encoder but also tail bits corresponding to the lower encoder $\{X'_{k+1}, X'_{k+2}, X'_{k+3}\}$. In addition to these six tail bits, six corresponding parity bits $\{Z_{k+1}, Z_{k+2}, Z_{k+3}\}$ and $\{Z'_{k+1}, Z'_{k+2}, Z'_{k+3}\}$ for the upper and lower encoder respectively are also transmitted. First, the switch in the upper (first) encoder is brought to lower (flushing) position and then the switch in the lower (second) encoder. The tail bits are then transmitted at the end of the encoded data frame. The tail bits sequence are:

$$X_{K+1}, Z_{K+1}, X_{K+2}, Z_{K+2}, X_{K+3}, Z_{K+3}, X'_{K+1}, Z'_{K+1}, X'_{K+2}, Z'_{K+2}, X'_{K+3}, Z'_{K+3}$$

The total length of the encoded bit sequence now becomes $3K + 12$, $3K$ being the coded data bits and 12 being the tail bits. The code rate of the encoder is thus $r = c / (3K + 12)$. However, for large size of input K , the fractional loss in code rate due to tail bits is negligible and thus, the code rate is approximated at $1/3$.

3.1.6 Interleaver

The bits input to the turbo code internal interleaver are denoted by c_0, c_1, \dots, c_K , where K is the number of input bits. The bits output from the turbo code internal interleaver are denoted by c'_0, c'_1, \dots, c'_K .

The relationship between the input and output bits is as follows:

$$c'_i = c'_{\pi(i)}, \quad i = 0, 1, \dots, (K - 1)$$

where the relationship between the output index i and the input index $\pi(i)$ satisfies the following quadratic form:

$$(i) = (f_1 \cdot i + f_2 \cdot i^2) \text{ mod } K$$

The parameters f_1 and f_2 depend on the block size K and are summarized in

Appendix A. [14]

3.2 Decoder

This section describes the basic turbo code decoder. The turbo code decoder is based on a modified Viterbi algorithm that incorporates reliability values to improve decoding performance. First, this section introduces the concept of reliability for Viterbi decoding. Then, the metric that will be used in the modified Viterbi algorithm for turbo code decoding is described. Finally, the decoding algorithm and implementation structure for a turbo code are presented. [11]

3.2.1 Turbo Decoder

The iterative decoding of concatenated codes has been termed Turbo decoding after the name of turbo engines. It then gave its name to a whole class of codes, the parallel concatenated convolutional codes, Turbo codes. This iterative decoding process is shown in Figure 3-10.

During the operation, the probability of decoding in favor of the correct decision is improved by exchanging information between the two decoders. A single iteration begins by the first decoder (DEC1) calculating a soft output and then passing it to the second decoder (DEC2). The new decoder computes its soft output and the iteration is completed by passing the output of DEC2 to DEC1 to repeat the process over and over. The soft output is a real number called the a posteriori probability (APP) that measures the probability of a correct decision for each bit in the information sequence.

We need a "soft-in/soft-out" decoder for decoding the constituent codes. Such decoder uses a priori values for all information bits, if available, as well as the channel outputs. It delivers soft outputs for all information bits. There are two categories of soft decision decoders. The first is based on the Maximum A posteriori Probability (MAP) decoding algorithm [15] while the second is based on a Maximum Likelihood (ML) decoding algorithm such as the Soft Output Viterbi Algorithm (SOVA) [16]. The latter is a modified Viterbi algorithm that yields soft

outputs.

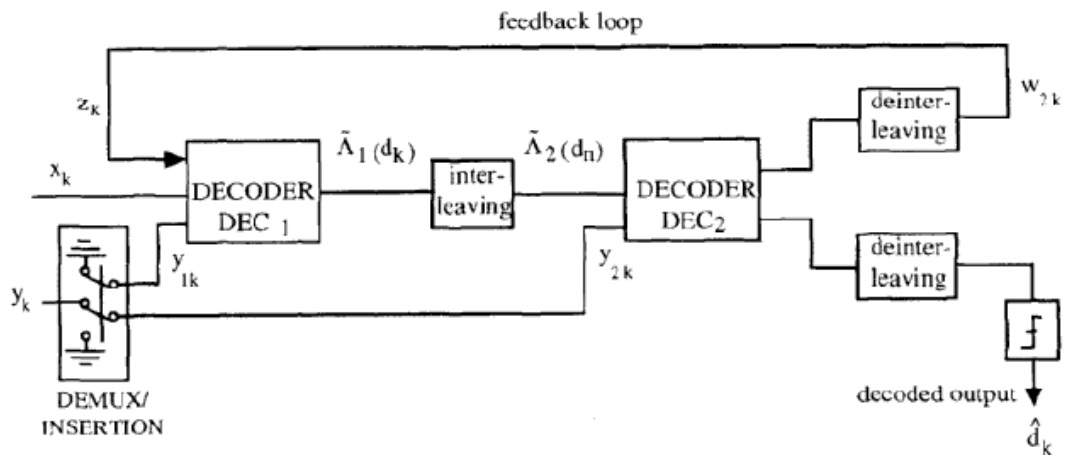


Figure 3-10: Turbo Decoder

The gain of using the Log-MAP based decoder compared to SOVA based decoders is only around 0.1 dB for an E_b/N_0 around 2.0 dB [17]. Also MAP has a complexity of $O(n^2)$ for its comparisons, and $O(2n^2)$ for its summations, while SOVA has a complexity of $O(0.5 n^2)$ for its comparisons and $O(0.5n^2)$ for its summations, where n is the number of bits for decoding [18]. The small MAP decoder gain does not make up for increased complexity cost and is therefore not as interesting for the industry as SOVA. This is the reason for investigating SOVA thoroughly instead of the MAP algorithm.

3.2.2 Principle of the General Soft-Output Viterbi Decoder

The Viterbi algorithm produces the ML output sequence for convolutional codes. This algorithm provides optimal sequence estimation for one stage convolutional codes. For concatenated (multistage) convolutional codes, there are two main drawbacks to conventional Viterbi decoders.

First, the first Viterbi decoder produces bursts of bit errors which degrades the performance of the second Viterbi decoders. Second, the first Viterbi decoder produces hard decision outputs which prohibits the second Viterbi decoders from deriving the benefits of soft decisions [19]. Both of these drawbacks can be reduced and the performance of the overall concatenated decoder can be significantly improved if the Viterbi decoders are able to produce reliability (soft-output) values [20].

The reliability values are passed on to subsequent Viterbi decoders as a priori information to improve decoding performance. This modified Viterbi decoder is referred to as the soft-output Viterbi algorithm (SOVA) decoder. Figure 3-11 shows a concatenated SOVA decoder.

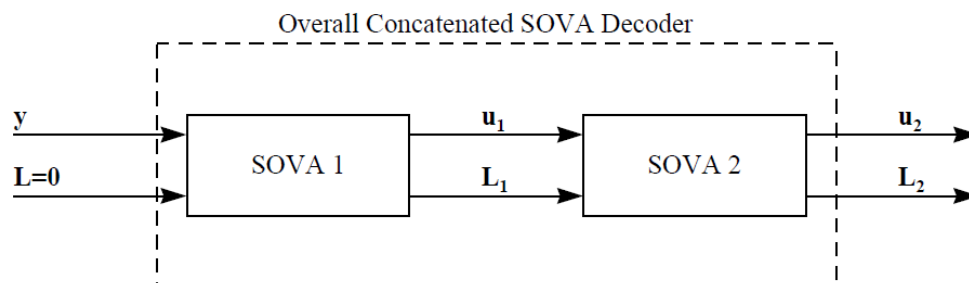


Figure 3-11: A concatenated SOVA decoder where y represents the received channel values, u represents the hard decision output values, and L represents the associated reliability values.

3.2.3 Likelihood Functions

The mathematical foundations of hypothesis testing rest on Bayes' theorem. For communications engineering, where applications involving an AWGN channel are of great interest, the most useful form of Bayes' theorem expresses the *a posteriori probability* (APP) of a decision in terms of a continuous-valued random variable x in the following form:

$$P(d = i | x) = \frac{p(x|d = i)P(d = i)}{p(x)} \quad i = 1, \dots \quad (3.1)$$

$$p(x) = \sum_{i=1}^M P(x|d = i)P(d = i) \quad (3.2)$$

Where $P(d = i|x)$ is the APP, and $d = i$ represents data d belonging to the i^{th} signal class from a set of M classes. Further, $p(x|d = i)$ represents the probability density function (pdf) of a received continuous-valued data-plus-noise signal x , conditioned on the signal class $d = i$. Also, $P(d = i)$, called the a priori probability, is the probability of occurrence of the i^{th} signal class.

Typically x is an “observable” random variable or a test statistic that is obtained at the output of a demodulator or some other signal processor. Therefore, $p(x)$ is the pdf of the received signal x , yielding the test statistic over the entire space of signal classes.

Let the binary logical elements 1 and 0 be represented electronically by voltages +1 and -1, respectively. The variable d is used to represent the transmitted data bit, whether it appears as a voltage or as a logical element. Sometimes one format is more convenient than the other; the reader should be able to recognize the difference from the context.

The decision rule in terms of APPs is as follows is given by:

$$\begin{array}{c}
 H1 \\
 P(d = +1|x) > P(d = -1|x) \\
 < \\
 H2
 \end{array} \tag{3.3}$$

Equation (3.3) states that you should choose the hypothesis $H1$, ($d = +1$), if the APP $P(d = +1|x)$, is greater than the APP $P(d = -1|x)$. Otherwise, you should choose hypothesis $H2$, ($d = -1$). Using the Bayes' theorem of Equation (3.1), the APPs in Equation (3.3) can be replaced by their equivalent expressions, yielding the following:

$$\begin{array}{c}
 H1 \\
 P(x|d = +1)P(d = +1) > P(x|d = -1)P(d = -1) \\
 < \\
 H2
 \end{array} \tag{3.4}$$

Where the pdf $p(x)$ appearing on both sides of the inequality in Equation (3.1) has been canceled.

Equation (3.4) is generally expressed in terms of a ratio, yielding the so-called likelihood ratio test, as follows:

$$\begin{array}{c}
 H1 \\
 P(x|d = +1)P(d = +1) > P(x|d = -1)P(d = -1) \\
 < \\
 H2
 \end{array} \tag{3.5}$$

3.2.3.1 Log-Likelihood Ratio

By taking the logarithm of the likelihood ratio developed in Equations (3.3) through (3.5), we obtain a useful metric called the *log-likelihood ratio (LLR)*. It is a real number representing a soft decision out of a detector, designated by as follows:

$$L(d | x) = \log \left[\frac{P(d = +1 | x)}{P(d = -1 | x)} \right] = \log \left[\frac{P(x | d = +1)P(d = +1)}{P(x | d = -1)P(d = -1)} \right] \quad (3.6)$$

$$L(d | x) = \log \left[\frac{P(x | d = +1)}{P(x | d = -1)} \right] + \log \left[\frac{P(d = +1)}{P(d = -1)} \right] \quad (3.7)$$

$$L(d | x) = L(x | d) + L(d) \quad (3.8)$$

Where $L(x|d)$ is the LLR of the test statistic x obtained by measurements of the channel output x under the alternate conditions that $d = +1$ or $d = -1$ may have been transmitted, and $L(d)$ is the a priori LLR of the data bit d .

To simplify the notation, Equation (3.8) is rewritten as follows:

$$L'(d') = L_c(x) + L(d) \quad (3.9)$$

Where the notation $L_c(x)$ emphasizes that this LLR term is the result of a channel measurement made at the receiver. Equations (3.1) through (3.9) were developed with only a data detector in mind. Next, the introduction of a decoder will typically yield decision-making benefits.

For a systematic code, it can be shown that the LLR (soft output) $L(d')$ out of the decoder is equal to Equation (3.10):

$$L(d') = L'(d') + L_e(d') \quad (3.10)$$

Where $L'(d')$ is the LLR of a data bit out of the demodulator (input to the decoder), and $L_e(d')$, called the *extrinsic* LLR, represents extra knowledge gleaned from the decoding process. The output sequence of a systematic decoder is made up of values representing data bits and parity bits. From Equations (3.9) and (3.10), the output LLR $L(d')$ of the decoder is now written as follows:

$$L(d') = L_c(x) + L(d) + L_e(d') \quad (3.11)$$

Equation (3.11) shows that the output LLR of a systematic decoder can be represented as having three LLR elements a channel measurement, a priori knowledge of the data, and an extrinsic LLR stemming solely from the decoder. To yield the final $L(\hat{d})$, each of the individual LLRs can be added as shown in Equation (11), because the three terms are statistically independent. This soft decoder output $L(\hat{d})$ is a real number that provides a hard decision as well as the reliability of that decision. The sign of $L(\hat{d})$ denotes the hard decision; that is, for positive values of $L(\hat{d})$ decide that $d = +1$, and for negative values decide that $d = -1$. The magnitude of $L(\hat{d})$ denotes the reliability of that decision. Often, the value of $L_e(\hat{d})$ due to the decoding has the same sign as $L_c(x) + L(d)$, and therefore acts to improve the reliability of $L(\hat{d})$.

The channel model is assumed to be flat fading with Gaussian noise. By using the Gaussian pdf $f(z)$,

$$f(z) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(z-m)^2}{2\sigma^2}} \quad (3.12)$$

Where m is the mean and the σ^2 is the variance, it can be shown that

$$\ln \frac{p(y|x=+1)}{p(y|x=-1)} = \ln \frac{e^{-\frac{E_b}{N_o}(y-a)^2}}{e^{-\frac{E_b}{N_o}(y+a)^2}} = \ln \frac{e^{-\frac{E_b}{N_o}2ay}}{e^{\frac{E_b}{N_o}2ay}} = 4 \frac{E_b}{N_o} ay \quad (3.13)$$

Where E_b/N_o is the signal to noise ratio per bit (directly related to the noise variance) and a is the fading amplitude. For nonfading Gaussian channel, $a=1$.

Generally speaking the LLR value can be formulated as $L(d|x) = L_c Y_k + L(u_k)$

3.2.4 Reliability of the General SOVA Decoder

The reliability of the SOVA decoder is calculated from the trellis diagram as shown in Figure 3-12.

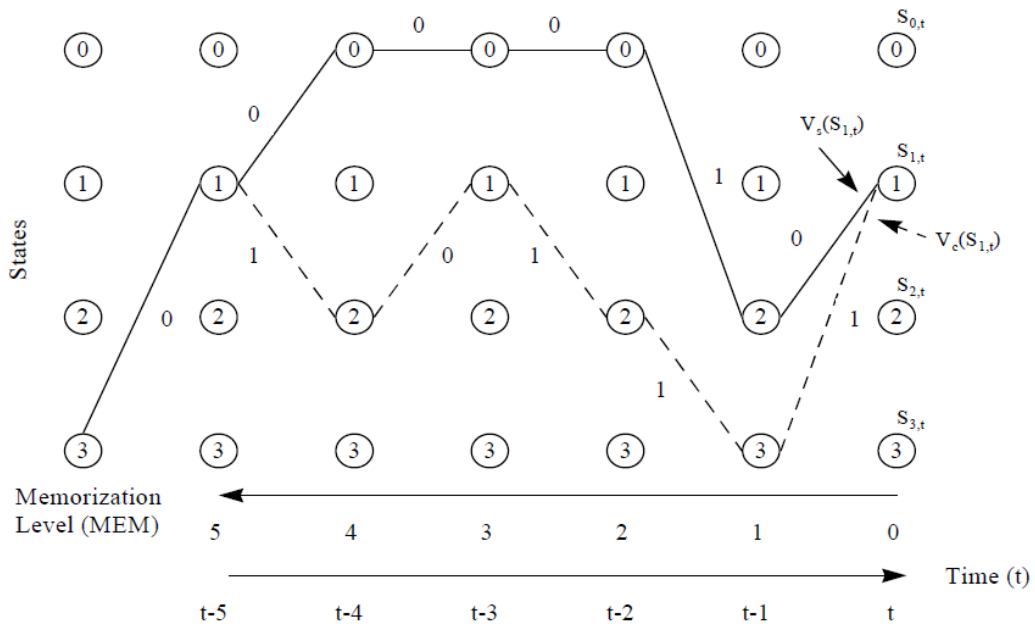


Figure 3-12: Example of survivor and competing paths for reliability estimation at time t

In Figure 3-12, a 4-state trellis diagram is shown. The solid line indicates the survivor path (assumed here to be part of the final ML path) and the dashed line indicates the competing (concurrent) path at time t for state 1. For the sake of brevity, survivor and competing paths for other nodes are not shown. The label $S_{1,t}$ represents state 1 and time t . Also, the labels $\{0,1\}$ shown on each path indicate the estimated binary decision for the paths. The survivor path for this node is assigned an accumulated metric $V_s(S_{1,t})$ and the competing path for this node is assigned an accumulated metric $V_c(S_{1,t})$. The fundamental information for assigning a reliability value $L(t)$ to node $S_{1,t}$'s survivor path is the absolute difference between the two accumulated metrics, $L(t) = |V_s(S_{1,t}) - V_c(S_{1,t})|$. The greater this difference, the more reliable is the survivor path. For this reliability calculation, it is assumed that the survivor accumulated metric is always "better" than the competing accumulated metric. Furthermore, to reduce complexity, the reliability values only need to be calculated for the ML survivor path (assume it is known for now) and are unnecessary for the other survivor paths since they will be discarded later.

To illustrate the concept of reliability, two examples are given below. In these examples, the Viterbi algorithm selects the survivor path as the path with the smaller accumulated metric. In the first example, assume that at node $S_{1,t}$ the accumulated survivor metric $V_s(S_{1,t})=50$ and that the accumulated competing metric $V_c(S_{1,t})=100$.

The reliability value associated with the selection of this survivor path is $L(t)=|50-100|=50$.

In the second example, assume that the accumulated survivor metric does not change, $V_s(S_{1,t})=50$, and that the accumulated competing metric $V_c(S_{1,t})=75$. The resulting reliability value is $L(t)=|50-75|=25$. Although in both of these examples the survivor path has the same accumulated metric, the reliability value associated with the survivor path is different. The reliability value in the first example provides more confidence (twice as much confidence) in the selection of the survivor path than the value in the second example.

Figure 3-13 illustrates a problem with the use of the absolute difference between accumulated survivor and competing metrics as a measure of the reliability of the decision.

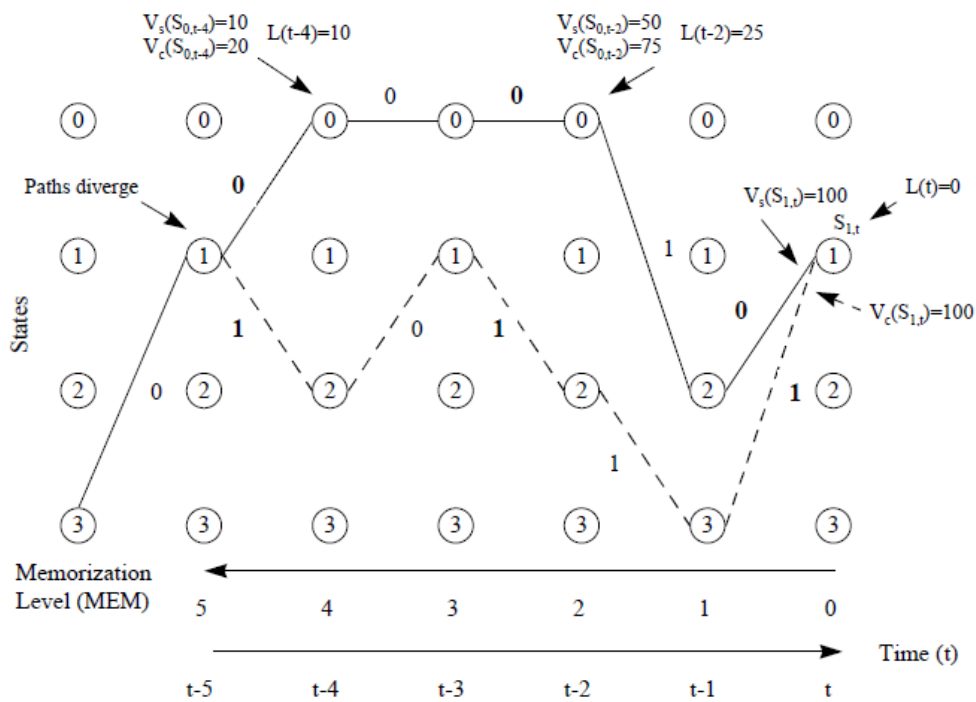


Figure 3-13: Example that shows the weakness of reliability assignment using metric values directly.

In Figure 3-13, the survivor and competing paths at $S_{1,t}$ have diverged at time t-5. The survivor and competing paths produce opposite estimated binary decisions at times t, t-2, and t-4 as shown in bold labels. For the purpose of illustration, let us suppose that the survivor and competing accumulated metrics at $S_{1,t}$ are equal, $V_s(S_{1,t}) = V_c(S_{1,t}) = 100$.

This means that both the survivor and competing paths have the same probability of being the ML path. Furthermore, let us assume that the survivor accumulated metric is “better” than the competing accumulated metric at time $t-2$ and $t-4$ as shown in Figure 4.3. To reduce the figure complexity, these competing paths for times $t-2$ and $t-4$ are not shown.

From this argument, it can be seen that the reliability value assigned to the survivor path at time t is $L(t)=0$, which means that there is no reliability associated with the selection of the survivor path. At times $t-2$ and $t-4$, the reliability values assigned to the survivor path were greater than zero ($L(t-2)=25$ and $L(t-4)=10$) as a result of the “better” accumulated metrics from the survivor path. However, at time t , the competing path could also have been the survivor path because they have the same metric. Thus, there could have been opposite estimated binary decisions at times t , $t-2$, and $t-4$ without reducing the associated reliability values along the survivor path.

To improve the reliability values of the survivor path, a trace back operation to update the reliability values has been suggested. This updating procedure is integrated into the Viterbi algorithm as follows:

For node $S_{k,t}$ in the trellis diagram (corresponding to state k at time t),

1. Store $L(t) = | V_s(S_{k,t}) - V_c(S_{k,t}) |$. (This is also denoted as Δ in other papers.)

If there is more than one competing path, then multiple reliability values must be calculated and the smallest reliability value is then set to $L(t)$.

2. Initialize the reliability value of $S_{k,t}$ to $+\infty$ (most reliable).

3. Compare the survivor and competing paths at $S_{k,t}$ and store the memorization levels (MEMs) where the estimated binary decisions of the two paths differ.

4. Update the reliability values at these MEMs with the following procedure:

a. Find the lowest $MEM > 0$, denoted as MEM_{low} , whose reliability value has not been updated.

b. Update MEM_{low} 's reliability value $L(t - MEM_{low})$ by assigning the lowest reliability value between $MEM = 0$ and $MEM = MEM_{low}$.

Continuing from the example, the opposite bit estimations between the survivor and competing bit paths for $S_{1,t}$ are located and stored as $MEM=\{0, 2, 4\}$. With this MEM information, the reliability updating process is accomplished as shown in Figure 3-14 and Figure 3-15. Figure 3-14, the first reliability update is shown. The lowest $MEM>0$, whose reliability value has not been updated, is determined to be $MEM_{low}=2$. The lowest reliability value between $MEM=0$ and $MEM=MEM_{low}=2$ is found to be $L(t)=0$. Thus, the associated reliability value is updated from $L(t-2)=25$ to $L(t-2)=L(t)=0$. The next lowest $MEM>0$, whose reliability value has not been updated, is determined to be $MEM_{low}=4$.

The lowest reliability value between $MEM=0$ and $MEM=MEM_{low}=4$ is found to be $L(t)=L(t-2)=0$. Thus, the associated reliability value is updated from $L(t-4)=10$ to $L(t-4)=L(t)=L(t-2)=0$. Figure 3-15 shows the second reliability update.

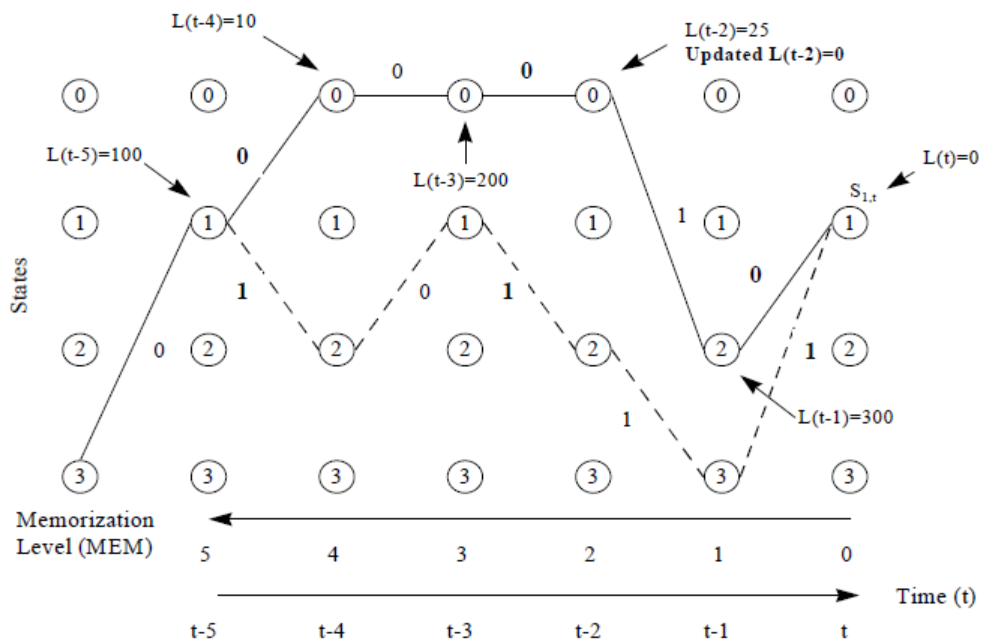


Figure 3-14: Updating process for time t-4 ($MEM_{low}=4$).

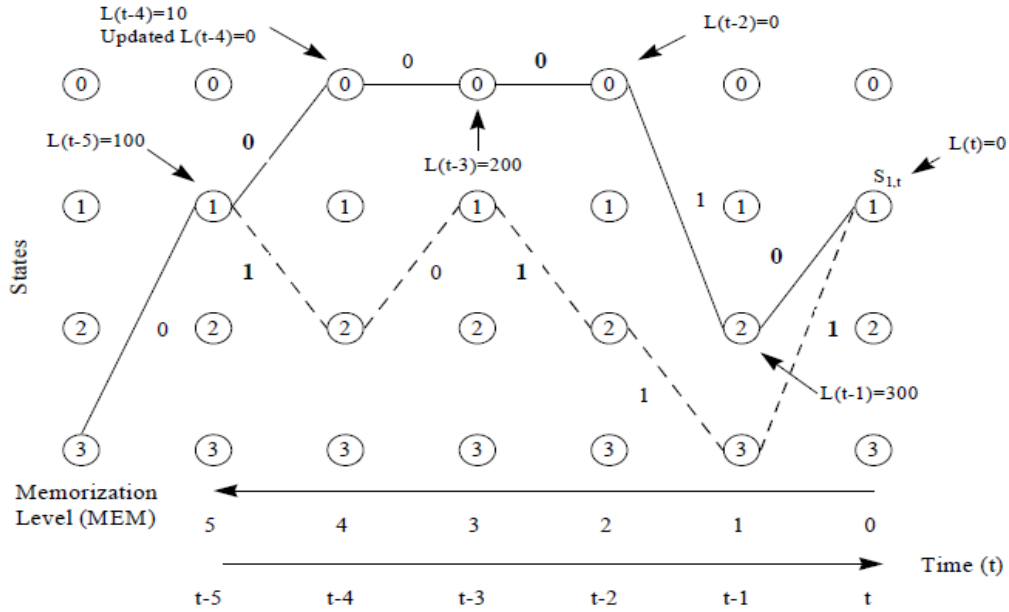


Figure 3-15: Updating process for time $t-2$ ($MEM_{low}=2$).

3.2.5 SOVA Component Decoder for a Turbo Code

The SOVA component decoder estimates the information sequence using one of the two encoded streams produced by the turbo code encoder. Figure 3-16 shows the inputs and outputs of the SOVA component decoder.



Figure 3-16: SOVA component decoder

The SOVA component decoder processes the (log-likelihood ratio) inputs $L(\mathbf{u})$ and $L_c \mathbf{y}$, where $L(\mathbf{u})$ is the a-priori sequence of the information sequence \mathbf{u} and $L_c \mathbf{y}$ is the weighted received sequence. The sequence \mathbf{y} is received from the channel. However, the sequence $L(\mathbf{u})$ is produced and obtained from the preceding SOVA component decoder.

If there is no preceding SOVA component decoder then there are no a-priori values. Thus, the $L(\mathbf{u})$ sequence is initialized to the all-zero sequence. A similar concept is

also shown at the beginning of the chapter in Figure 3-1. The SOVA component decoder produces \mathbf{u}' and $\mathbf{L}(\mathbf{u}')$ as outputs where \mathbf{u}' is the estimated information sequence and $\mathbf{L}(\mathbf{u}')$ is the associated log-likelihood ratio (“soft” or L-value) sequence.

The SOVA component decoder operates similarly to the Viterbi decoder except the ML sequence is found by using a modified metric. This modified metric, which incorporates the a-priori value, is derived in Appendix B.

For systematic codes, this can be modified to become

$$M_t^{(m)} = M_{t-1}^{(m)} + u_t^{(m)} L_c y_{t,1} + \sum_{j=2}^N x_{t,j}^{(m)} L_{ct,j} y_{t,j} + u_t^{(m)} L(u_t) \quad (3.14)$$

For each state in the trellis diagram where m denotes allowable binary trellis branch/transition to a state (m= 1, 2).

$M_t^{(m)}$ is the accumulated metric for time t on branch m.

$u_t^{(m)}$ is the systematic bit (1st bit of N bits) for time t on branch m.

$x_{t,j}^{(m)}$ is the j-th bit of N bits for time t on branch m ($2 \leq j \leq N$).

$y_{t,j}^{(m)}$ is the received value from the channel corresponding to $x_{t,j}^{(m)}$.

$L_c = 4 \frac{E_b}{N_0}$ is the channel reliability value.

$L(u_t)$ is the a-priori reliability value for time t. This value is from the preceding decoder. If there is no preceding decoder, then this value is set to zero.

Figure 3-17 shows a trellis diagram with two states S_a and S_b and a transition period between time t-1 and time t. The solid line indicates that the transition will produce an information bit $u_t=+1$ and the dash line indicates that the transition will produce an information bit $u_t=-1$. The source reliability $L(u_t)$, which may be either a positive or a negative value, is from the preceding SOVA component decoder. The “add on” value is incorporated into the SOVA metric to provide a more reliable decision on the estimated information bit. For example, if $L(u_t)$ is a “large” positive number, then it would be relatively more difficult to change the estimated bit decision from +1 to -1

between decoding stages (based on assigning $\max\{M_t^{(m)}\}$ to the survivor path). However, if $L(u_t)$ is a “small” positive number, then it would be relatively easier to change the estimated bit decision from +1 to -1 between decoding stages. Thus, $L(u_t)$ is like a buffer which tries to prevent the decoder from choosing the opposite bit decision to the preceding decoder.

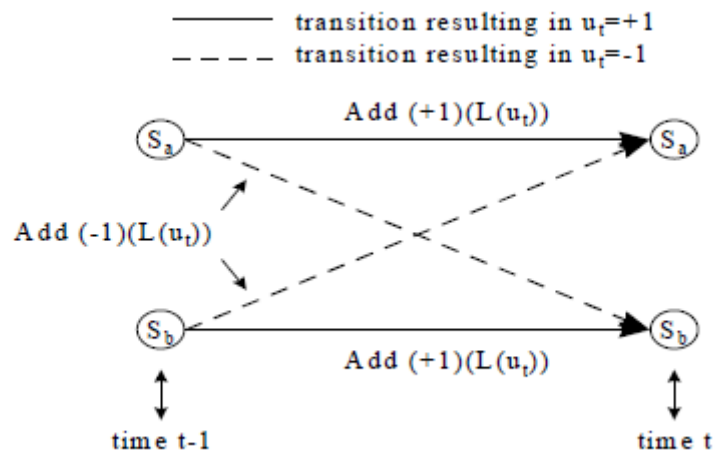


Figure 3-17: Source reliability for SOVA metric computation

At time t , the reliability value (magnitude of the log-likelihood ratio) assigned to a node in the trellis is determined from $\Delta_t^0 = |M_t^{(1)} - M_t^{(2)}|$ Where Δ_t^{MEM} denotes the reliability value at memorization level MEM relative to time t .

This notation is similar to the notation $L(t-MEM)$ as used before and is shown in Figure 4.10 for discussion.

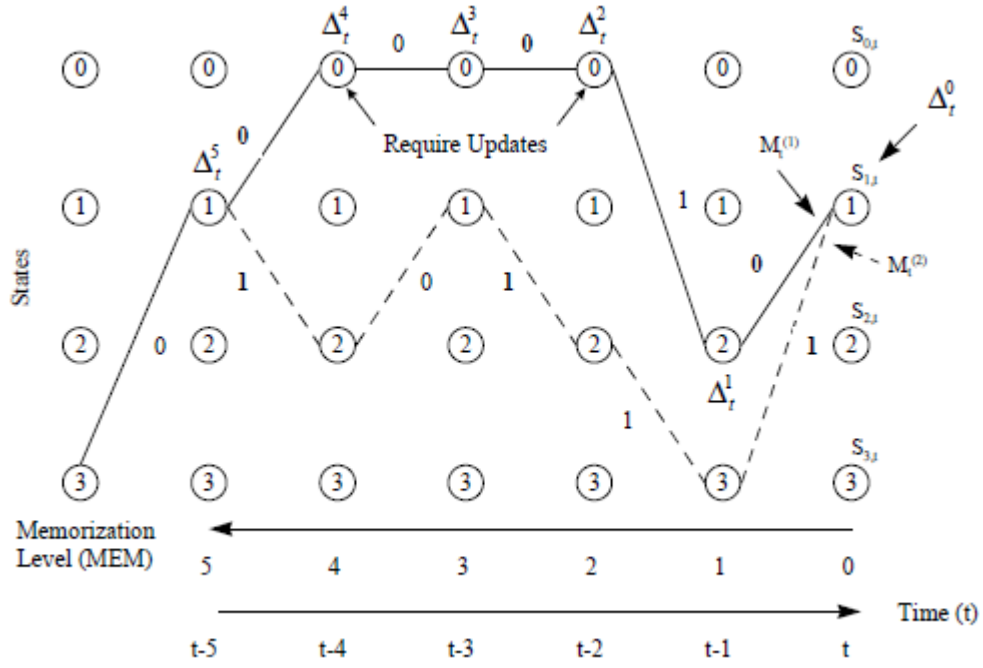


Figure 3-18: Example of SOVA survivor and competing paths for reliability estimation.

The probability of path m at time t and the SOVA metric are stated in [21] to be related as

$$P(\text{path}(m)) = P(S_t^{(m)}) = e^{-\frac{M_t^{(m)}}{2}} \quad (3.15)$$

At time t , let us suppose that the survivor metric of a node is denoted as M_t and the competing metric is denoted as M_t . Thus, the probability of selecting the correct survivor path is

$$\begin{aligned} P(\text{correct}) &= \frac{P(\text{path}(1))}{P(\text{path}(1)) + P(\text{path}(2))} \\ &= \frac{e^{-\frac{M_t^{(1)}}{2}}}{e^{-\frac{M_t^{(1)}}{2}} + e^{-\frac{M_t^{(2)}}{2}}} = \frac{e^{\Delta_t^0}}{1 + e^{\Delta_t^0}} \end{aligned} \quad (3.16)$$

The reliability of this path decision is calculated as

$$\log \frac{P(\text{correct})}{1-P(\text{correct})} = \log \frac{e^{\Delta_t^0}}{1+e^{\Delta_t^0}} \cdot \frac{1+e^{\Delta_t^0}}{e^{\Delta_t^0}} \cdot \frac{1}{1+e^{\Delta_t^0}} \quad (3.17)$$

The reliability values along the survivor path for a particular node at time t are denoted as Δ_t^{MEM} , where $MEM = 0, \dots, t$. For this node at time t , if the bit on the survivor path at $MEM=k$ (or equivalently at time $t-MEM$) is the same as the associated bit on the competing path, then there would be no bit error if the competing path was chosen. Thus, the reliability value at this bit position remains unchanged. However, if the bits differ on the survivor and competing path at $MEM=k$, then there is a bit error. The reliability value at this bit error position must then be updated using the same updating procedure as described at the beginning of the chapter. As shown in Figure 3-18, reliability updates are required for $MEM=2$ and $MEM=4$.

The reliability updates are performed to improve the “soft” or L-values. It is shown in [22] that the “soft” or L-value of a bit decision is

$$L(\hat{u}_{t-MEM}) \approx \hat{u}_{t-MEM} \cdot \min_{l=0, \dots, MEM} \{\Delta_t^k\} \quad (3.18)$$

The soft output Viterbi algorithm (along with its reliability updating procedure) can be implemented as follows:

1. (a) Initialize time $t = 0$.
 (b) Initialize $M_0^{(m)} = 0$ only for the zero state in the trellis diagram and all other state to $-\infty$.

2. (a) Set time $t = t + 1$.

(b) Compute the metric $M_t^{(m)} = M_{t-1}^{(m)} + u_t^{(m)} L_c y_{t,1} + \sum_{j=2}^N x_{t,j}^{(m)} L_{ct,j} y_{t,j} + u_t^{(m)} L(u_t)$

for each state in the trellis diagram where

m denotes allowable binary trellis branch/transition to a state ($m = 1, 2$).

$M_t^{(m)}$ is the accumulated metric for time t on branch m .

$u_t^{(m)}$ is the systematic bit (1st bit of N bits) for time t on branch m .

$x_{t,j}^{(m)}$ is the j -th bit of N bits for time t on branch m ($2 \leq j \leq N$).

$y_{t,j}^{(m)}$ is the received value from the channel corresponding to $x_{t,j}^{(m)}$.

$L_c = 4 \frac{E_b}{N_0}$ is the channel reliability value.

$L(u_t)$ is the a-priori reliability value for time t . This value is from the preceding decoder. If there is no preceding decoder, then this value is set to zero.

3. Find $\max M_t^{(m)}$ for each state. For simplicity, let $M_t^{(1)}$ denote the survivor path metric and $M_t^{(2)}$ denote the competing path metric.

4. Store $M_t^{(1)}$ and its associated survivor bit and state paths.

5. Compute $\Delta_t^0 = |M_t^{(1)} - M_t^{(2)}|$.

6. Compare the survivor and competing paths at each state for time t and store the

MEMs where the estimated binary decisions of the two paths differ.

7. Update $\Delta_t^{MEM} \approx \min_{K=0, \dots, U} \{ \Delta_t^K \}$ for all MEMs from smallest to largest MEM.

8. Go back to Step (2) until the end of the received sequence.

9. Output the estimated bit sequence \mathbf{u}' and its associated “soft” or L-value sequence $\mathbf{L}(\mathbf{u}') = \mathbf{u}' \bullet \Delta$, where \bullet operator defines element by element multiplication operation and Δ is the final updated reliability sequence. $\mathbf{L}(\mathbf{u}')$ is then processed (to be discussed later) and passed on as the a-priori sequence $\mathbf{L}(\mathbf{u})$ for the succeeding decoder.

3.2.6 SOVA Iterative Turbo Code Decoder

The SOVA component decoder produces the “soft” or L-value $L(u't)$ for the estimated bit $(u't)$ (for time t). The “soft” or L-value $L(u't)$ can be decomposed into three distinct terms as stated in [22].

$$L(u't) = L(u't) + L_{cyt,1} + L_e(u't)$$

$L(u't)$ is the a-priori value and is produced by the preceding SOVA component decoder. $L_{cyt,1}$ is the weighted received systematic channel value. $L_e(u't)$ is the extrinsic value produced by the present SOVA component decoder.

The information that is passed between SOVA component decoders is the extrinsic value

$$L_e(u't) = L(u't) - L(u't) - L_{c,yt,1}$$

The a-priori value $L(u't)$ is subtracted out from the “soft” or L-value $L(u't)$ to prevent passing information back to the decoder from which it was produced. Also, the weighted received systematic channel value $L_{c,yt,1}$ is subtracted out to remove “common” information in the SOVA component decoders. Figure 3-19 shows that the turbo code decoder is a closed loop serial concatenation of SOVA component decoders. In this closed loop decoding scheme, each of the SOVA component decoders estimates the information sequence using a different weighted parity check stream. The turbo code decoder further implements iterative decoding to provide more dependable reliability/a-priori estimations from the two different weighted parity check streams, hoping to achieve better decoding performance.

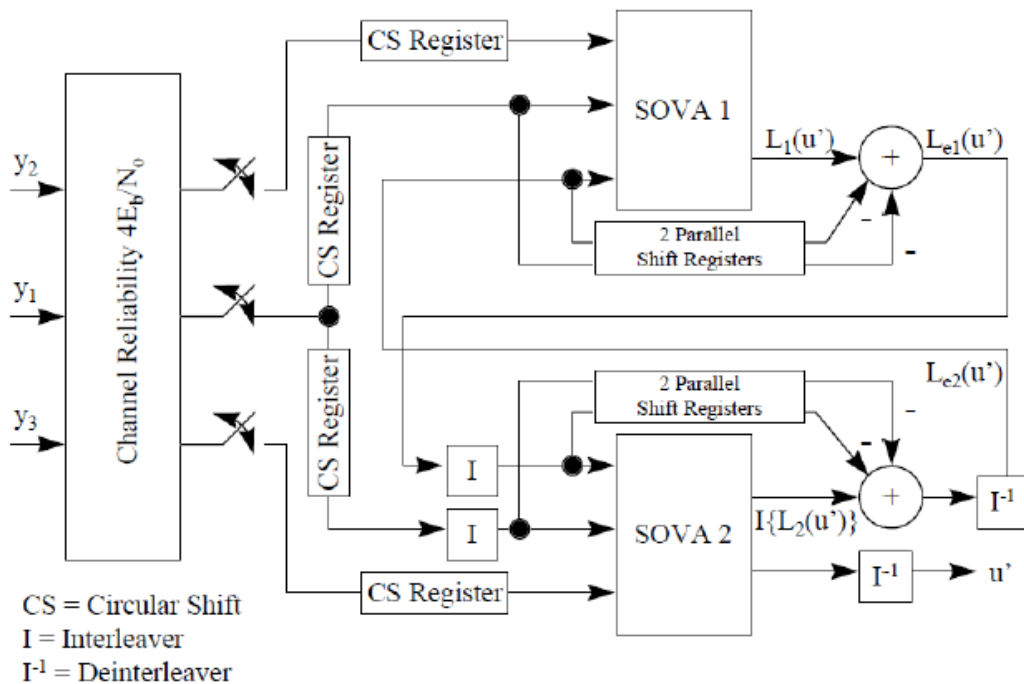


Figure 3-19: SOVA iterative turbo code decoder.

The iterative turbo code decoding algorithm for the n-th iteration is as follows:

1. The SOVA1 decoder inputs sequences $4E_b/N_0$ y_1 (systematic), $4E_b/N_0$ y_2 (parity check), and $L_{e2}(u')$ and outputs sequence $L_1(u')$. For the first iteration, sequence $L_{e2}(u')=0$ because there is no initial a-priori value (no extrinsic values from SOVA2).

2. The extrinsic information from SOVA1 is obtained by

$$L_{e1}(u') = L_1(u') - L_{e2}(u') - L_{cy1} \quad \text{Where } L_c = 4 \frac{E_b}{N_0}.$$

3. The sequences $4 \frac{E_b}{N_0} y_1$ and $L_{e1}(u')$ are interleaved and denoted as $I\{4 \frac{E_b}{N_0} y_1\}$ and $I\{L_{e1}(u')\}$.

4. The SOVA2 decoder inputs sequences $I\{4E_b/N_0 y_1\}$ (systematic), and $I\{4E_b/N_0 y_3\}$ (parity check that was already interleaved by the turbo code encoder), and $I\{L_{e1}(u')\}$ (a-priori information) and outputs sequences $I\{L_2(u')\}$ and $I\{u'\}$.

5. The extrinsic information from SOVA2 is obtained by

$$I\{L_{e2}(u')\} = I\{L_2(u')\} - I\{L_{e1}(u')\} - I\{L_{cy1}\}.$$

6. The sequences $I\{L_{e2}(u')\}$ and $I\{u'\}$ are deinterleaved and denoted as $L_{e2}(u')$ and u' . $L_{e2}(u')$ is fed back to SOVA1 as a-priori information for the next iteration and u' is the estimated bits output for the n-th iteration.

Chapter 4: Matlab Results and Analysis

4.1 Matlab Implementation

The input data frame size is between 40 and 6144 bits as is the size of the LTE interleaver according to the 3gpp standard. The turbo encoder consists of two main blocks, i.e., the recursive convolutional encoder and the interleaver. The encoded data frame is modulated using BPSK Modulation and sent over the channel. The channel model used in the simulation is AWGN channel. After adding noise to the data, the LLR is calculated and decoded using the turbo decoder using a specified number of iteration. The decoded bits are compared with the original bits to obtain the number of errors, hence calculate the BER.

As a rule of thumb, the number of samples required to obtain the ber with high accuracy is given by $10x - 100x \text{ BER}^{-1}$ samples. For example, if the estimated $\text{BER} = 10^{-6}$, 10^8 samples are used for a relative variance of 0.01 (99% confidence).

The decoder implementation is complex and computationally extensive. It includes processing using a number of loops. A limit is set on the maximum number of bits to be encoded and maximum allowable error for early termination of the code. The decoder decodes iteratively checking the number of errors after every iteration. If the number of errors is zero for an iteration, the code will not execute the next iteration to decrease processing load.

There are a large number of simulation options to consider when measuring the performance of a turbo decoder (for example, number of iterations, channel model, frame size, etc.). For this study many permutations were considered.

For performance analysis, it should be clear that there are many different configurations of turbo encoders. In this study, the LTE Turbo Encoder Scheme is used, which is a Parallel Concatenated Convolutional Code (PCCC) with two 8-state constituent encoders and one turbo code internal interleaver. The coding rate of turbo encoder is 1/3. The structure of turbo encoder is shown in details in chapter 3.

4.2 BER Performance over AWGN Channel

Since LTE Turbo decoder supports a wide range of frame sizes, three distinct frame sizes were used in the simulation to show the effect of frame sizes on the performance. The chosen sizes are 64, 1024, and 6144.

The turbo code program was simulated for frame size $K = 64$ over an AWGN channel. The SNR range was used from 0 to 4 dB. The number of decoder iterations was chosen to be 10. The BER for the iterations is shown in Figure 4-1. The BER values at the end of each iteration are given in Table 4-1.

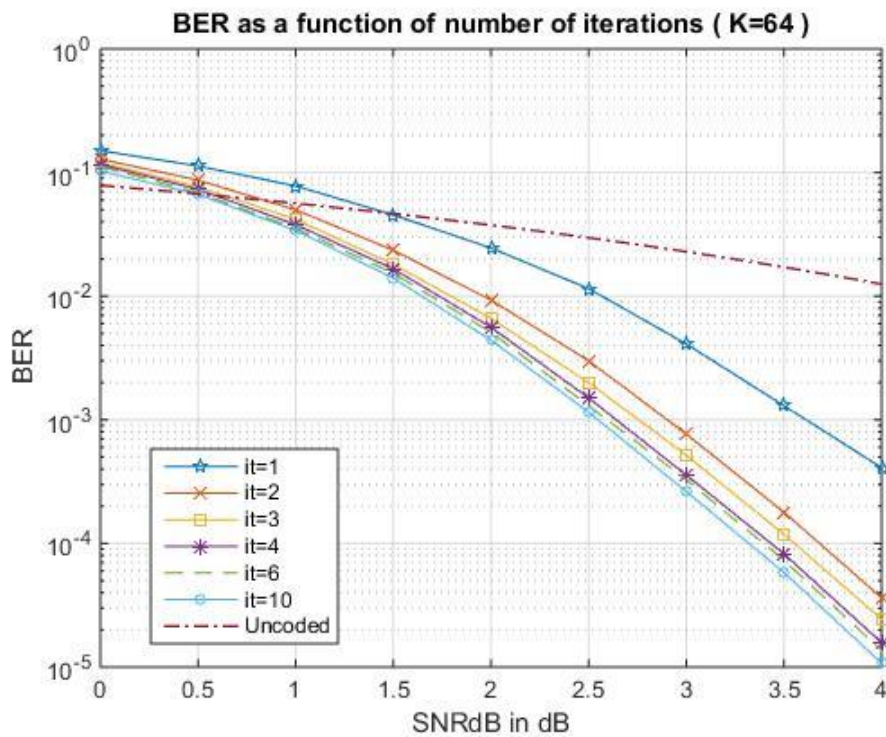


Figure 4-1 BER for frame K=64 over AWGN Channel

Table 4.1: BER values for frame K=64 over AWGN Channel

	SNR = 0	SNR = 1	SNR = 2	SNR = 3	SNR = 4
Iter =1	0.1502	0.0778	0.0245	0.0041	4.094e-04
Iter =2	0.1283	0.0499	0.0093	7.6750e-04	3.625e-05
Iter =3	0.1200	0.0417	0.0066	5.1667e-04	2.4375e-05
Iter =4	0.1163	0.0378	0.0056	3.5677e-04	1.5719e-05
Iter =6	0.1119	0.0353	0.0051	3.2708e-04	1.3624e-05
Iter = 10	0.1028	0.0340	0.0044	2.6266e-04	1.0743e-05

Similarly, the code was simulated for frame size $K = 1024$ over AWGN channel. The SNR range was used from 0 to 4 dB. The number of decoder iterations was chosen to be 5. Figure 4-2 depicts the performance improvement when the frame size is increased. The BER values at the end of each iteration are given in Table 4-3.

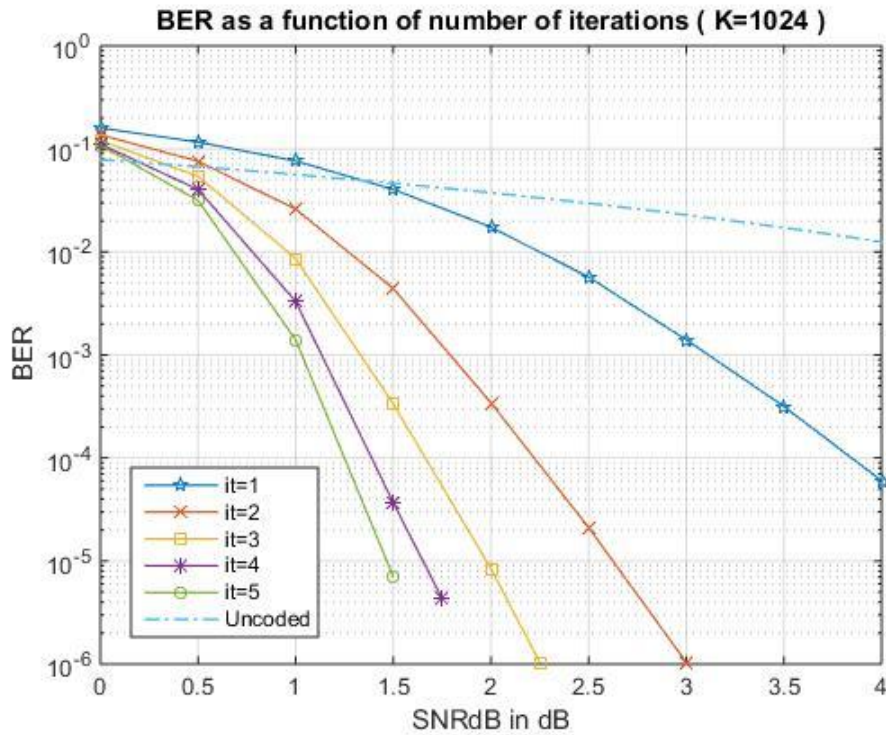


Figure 4-2 BER for frame $K=1024$ over AWGN Channel

Table 4.2: BER values for frame $K=1024$ over AWGN Channel

	SNR = 0	SNR = 0.5	SNR = 1	SNR = 1.5	SNR = 2
Iter =1	0.0762	0.1165	0.0762	0.0404	0.0174
Iter =2	0.1368	0.0759	0.0260	0.0044	3.4258e-04
Iter =3	0.1207	0.0541	0.0086	3.3320e-04	8.4352e-06
Iter =4	0.1113	0.0403	0.0033	3.6719e-05	-
Iter =5	0.1083	0.0316	0.0014	7.0312e-06	-

Finally, Fig. 4-3 describes the performance for frame size $K = 6144$ over AWGN channel. The SNR range was used from 0 to 4 dB. The number of decoder iterations was chosen to be 5. Some important BER values at the end of each iteration are given in Table 4-3.

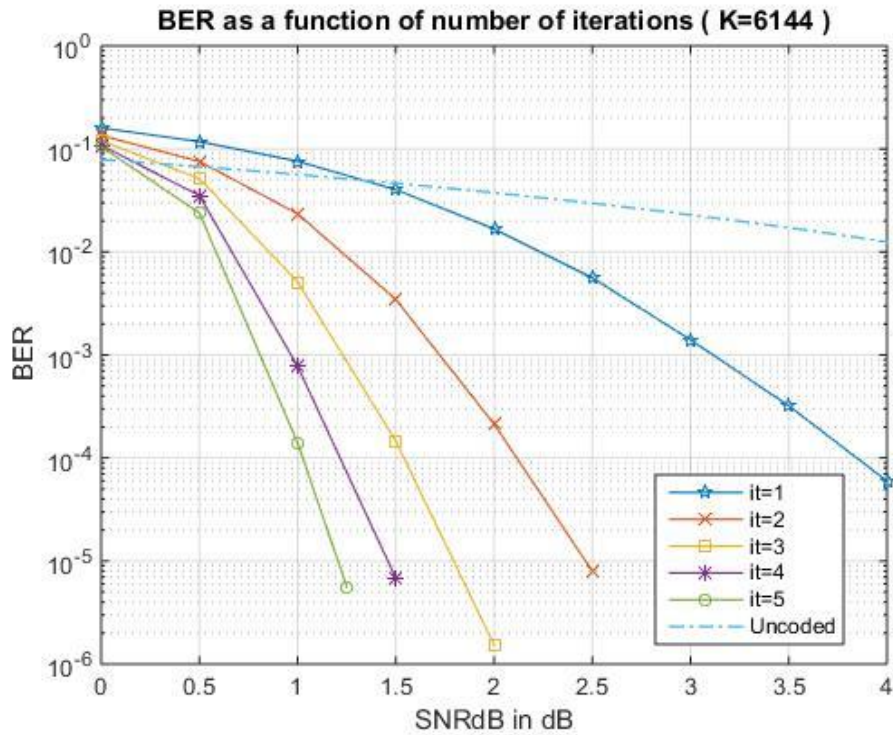


Figure 4-3: BER for frame $K=6144$ over AWGN Channel

Table 4.3: BER values for frame $K=6144$ over AWGN Channel

	SNR = 0	SNR = 0.5	SNR = 1	SNR =1.5	SNR = 2
Iter =1	0.1598	0.1177	0.0759	0.0402	0.0167
Iter =2	0.1352	0.0754	0.0234	0.0034	0.0002
Iter =3	0.1207	0.0514	0.0051	1.4583e-004	1.4583e-004
Iter =4	0.1084	0.0352	7.7441e-004	1.1068e-005	-
Iter =5	0.1053	0.0238	1.3997e-004	-	-

4.3 Interpretation of Results

It can be seen that as the number of iteration increases, the BER performance improves. However, the rate of improvement decreases. This is depicted by the overlapping curves after 5th iterations as shown in Fig.4-1. The BER does not show significant improvement after 5th iteration. Thus, the number of iterations should be kept such as to avoid extra computations.

Turbo code performance can be improved by increasing the frame size K. The code can achieve higher BER with the increase of frame size. This is because the interleaver permutes the data and the decoder is better able to decode the data.

However, it can be seen that by increasing the frame size K, the code can achieve the same BER at much lower SNR.

It should be noted that larger frame sizes mean more latency as the encoding and decoding is done per frame. Thus, the performance improvement is achieved at the cost of increased latency.

The performance comparison of turbo code can be done by plotting the BER for different frame sizes K as in Figure 4-4. The figure shows an example that by increasing the frame size K, the BER performance of the code improves. As a result, lower BER can be achieved by keeping the SNR constant.

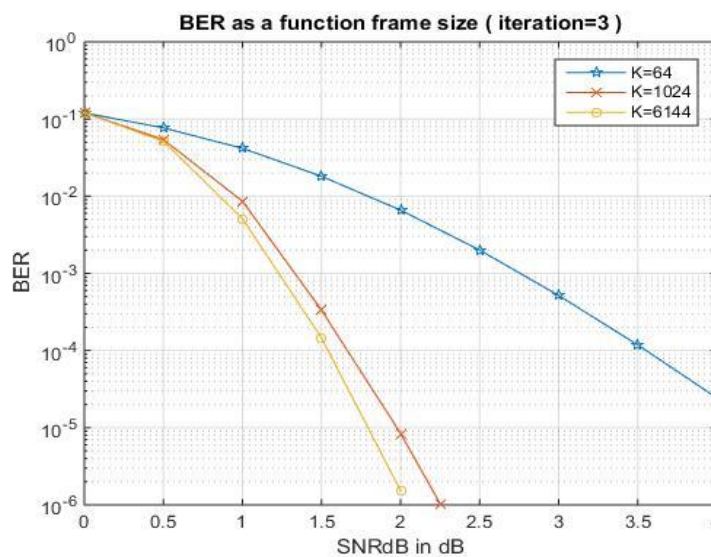


Figure 4-4: BER Comparison between different frame sizes over AWGN using 3 iterations

Chapter 5: Hardware Architecture for SOVA

This chapter presents a hardware architecture that can be applied to implementation of SOVA decoders. Optimizations are done to reduce the hardware complexity of the SOVA decoder. Some hardware issues are discussed through the chapter, and trade-offs between the hardware costs and performance are presented. [23],[24],[25]

5.1 SOVA Component

The Hardware Architecture for SOVA consists of three stages, namely the trellis, merge and decode stages, as illustrated in Figure 5-5.

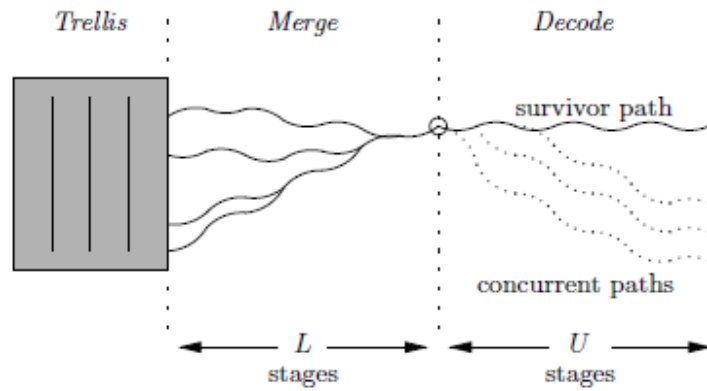


Figure 5-5: Hardware stages in SOVA

5.1.1 Trellis Stage

The first stage is the trellis stage that computes the state metrics of each of the 2^v states, according to Equation 5.1. That can be separated into the recursion and the branch metric terms.

$$M_K(S_K) = M_{K-1}(S_{K-1}) + \underbrace{\frac{1}{2} L_c \cdot y_{k,1} \cdot u_k + \frac{1}{2} \sum_{v=2}^n x_{k,v}^{(m)} \cdot L_c \cdot y_{k,v}}_{\text{branch metric}} + \frac{1}{2} u_K \cdot L(u_K) \quad (5.1)$$

Assuming that the expected parity bit $x_{k,2} = +1$, or equivalently that the parity bit has a value of '0', and let the branch metrics for the cases when the expected systematic bits of $u_k=0$ and $u_k=1$ be denoted by $\lambda_{0,0}$ and $\lambda_{1,0}$ respectively.

These branch metrics are then calculated as follows

$$\begin{aligned}\lambda_{0,0} &= \frac{1}{2}L_k \cdot (+1) + \frac{1}{2}L_c \cdot y_{k,1} \cdot (+1) + \frac{1}{2}L_c \cdot y_{k,2} \cdot (+1) \\ &= \left(\frac{1}{2}L_k + \frac{1}{2}L_c \cdot y_{k,1} \right) + \frac{1}{2}L_c \cdot y_{k,2}\end{aligned}\quad (5.2)$$

$$\begin{aligned}\lambda_{1,0} &= \frac{1}{2}L_k \cdot (-1) + \frac{1}{2}L_c \cdot y_{k,1} \cdot (-1) + \frac{1}{2}L_c \cdot y_{k,2} \cdot (+1) \\ &= -\left(\frac{1}{2}L_k + \frac{1}{2}L_c \cdot y_{k,1} \right) + \frac{1}{2}L_c \cdot y_{k,2}\end{aligned}\quad (5.3)$$

Where $y_{k,1}$ and $y_{k,2}$ are the received channel systematic and parity values respectively, and L_k is input extrinsic information. Likewise, by assuming once again that the expected parity bit $x_{k,2} = -1$, the branch metrics for the expected systematic bits of $u_k = 0$ and $u_k=1$ denoted by $\lambda_{0,1}$ and $\lambda_{1,1}$ respectively can be calculated.

$$\begin{aligned}\lambda_{0,1} &= \frac{1}{2}L_k \cdot (+1) + \frac{1}{2}L_c \cdot y_{k,1} \cdot (+1) + \frac{1}{2}L_c \cdot y_{k,2} \cdot (-1) \\ &= \left(\frac{1}{2}L_k + \frac{1}{2}L_c \cdot y_{k,1} \right) - \frac{1}{2}L_c \cdot y_{k,2} = -\lambda_{0,0}\end{aligned}\quad (5.4)$$

$$\begin{aligned}\lambda_{1,1} &= \frac{1}{2}L_k \cdot (-1) + \frac{1}{2}L_c \cdot y_{k,1} \cdot (-1) + \frac{1}{2}L_c \cdot y_{k,2} \cdot (+1) \\ &= -\left(\frac{1}{2}L_k + \frac{1}{2}L_c \cdot y_{k,1} \right) + \frac{1}{2}L_c \cdot y_{k,2} = -\lambda_{1,0}\end{aligned}\quad (5.5)$$

It can be observed that $\lambda_{0,1}$ and $\lambda_{1,1}$ are simply the negative values of $\lambda_{1,0}$ and $\lambda_{0,0}$ respectively. This implies that there is only a need to generate two of the four branch

metrics, since the other two can be easily obtained in the ACS when needed. A branch metric calculation unit BMC as shown in Figure 5-1 is used to generate $\lambda_{0,0}$ and $\lambda_{0,1}$.

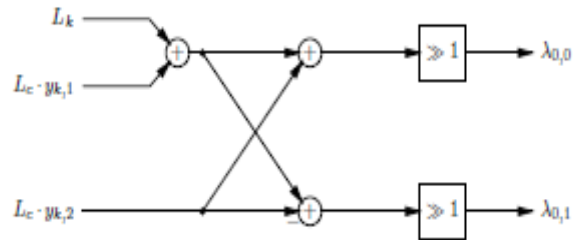


Figure 5-1: BMC module in the trellis unit

Other than the BMC unit, there are 2^v ACS in the trellis unit that will perform recursion by adding the branch metric to the previous survivor path metric, compare the two resultant metrics and finally select a survivor path metric to be saved for use at the next stage. Each ACS will perform calculations for one state, and the connections between the ACS are dependent on the generator's polynomials. In the case of the LTE turbo code, there are a total of 8 ACS modules connected together as shown in Figure 5-2.

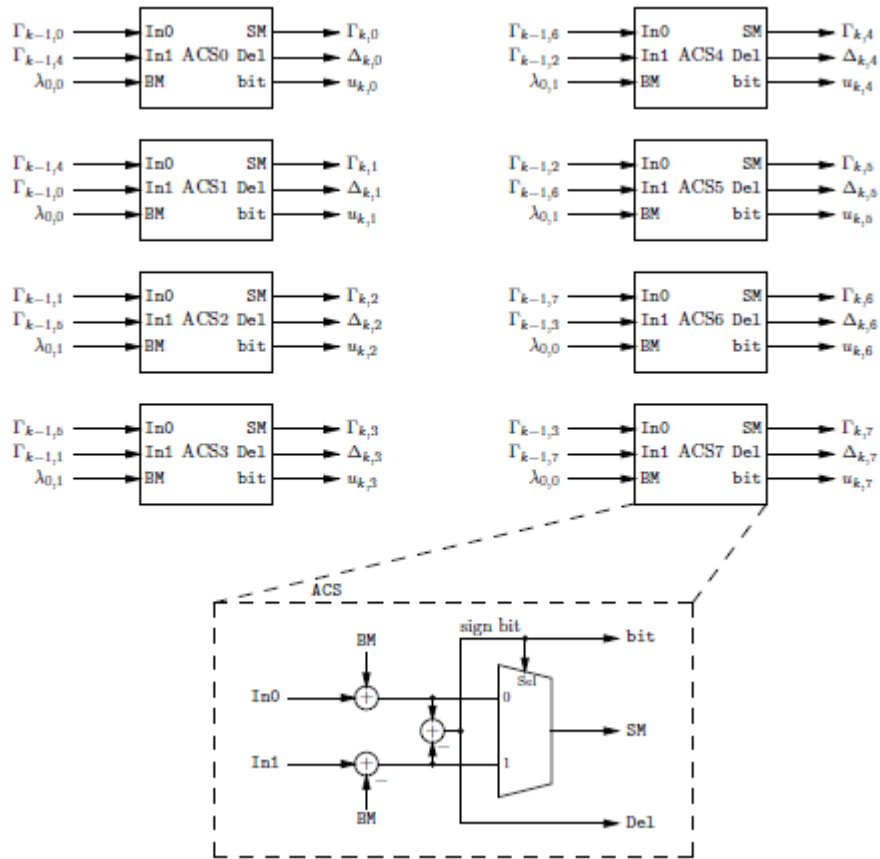


Figure 5-2: Trellis unit for LTE consisting of 8 ACS

A close up view of the ACS can be seen in Figure 5-2. By taking advantage of the fact that the branch metric and its inverse are negative values of one another, the ACS uses a subtractor at one of its input branches to obtain the required inverse branch metric. Thus by using one addition and one subtraction, the path metrics for both cases when the input decision bit is ‘1’ and ‘0’ can be obtained. The two path metrics are then compared to select the survivor (larger) metric which will be output from ACS and stored for the next stage of the trellis. The hard decision bit corresponding to the selection of the survivor path, together with the difference between the two metric values are also output to be used in later stages for merging and SOVA updates.

5.2 Trace back and Updating Depths

In a practical decoder, it is not possible to perform VA over the entire block of channel data due to the excessive latency and storage requirements. Instead, sliding windows of merging depth L and update depth U are used to limit the trace back and decoding depths. One possible modification is to vary L and U parameters for the SOVA algorithm.

Increasing L will increase the likelihood for a merged path while using a larger U increases the number of updates for the reliability value. However, it is clear that increasing these two values will increase the memory requirements and latency. The simulation results for determining the optimal values for L and U are presented in Section 5.4 Hardware considerations such as performance, latency and hardware requirements.

5.2.1 Merge Stage

The merge stage of depth L performs Viterbi decoding on the hard decision bits determined at the trellis stage. The depth L has to be sufficiently large for all 2^v paths to merge after L stages, and is usually expressed in terms the constraint length of the encoder K . In hardware, the merge stage can either be implemented via the Register Exchange (RE) or traceback method. The block diagram of a RE unit suitable for use in LTE is shown in Figure 5-3.

The RE method utilizes registers to store all the $N \cdot L$ decision bits within the trellis. Each row in the RE unit contains the decision bits of the entire path of length L corresponding to the state of the first register of the row. The hard decision bits from the trellis unit are used as the select signals for the MUXes to control the state exchanges. The connections between the columns of registers are identical for all columns and dependent on the generator polynomial. Assuming that the depth of the RE is sufficient for merging, the output of all N rows of the RE unit would give the same decision bit (i.e. decision bit of the survivor path), which would be selected as the estimated received bit u_k .

For the case of traceback, the decision bits are stored in a memory instead of registers, and a decision bit d stages away from a given state is to be determined by traversing d steps backwards in a trellis. The main advantage of traceback is that it can be implemented efficiently in dense memory, but the drawback is that there is increased latency as compared to RE. Both methods are commonly used in hardware designs of Viterbi decoders, and the chosen method is usually dependent on the trade-off between latency and hardware utilization. In this thesis, only the RE method is considered, due to the short latency required for the LTE decoder.

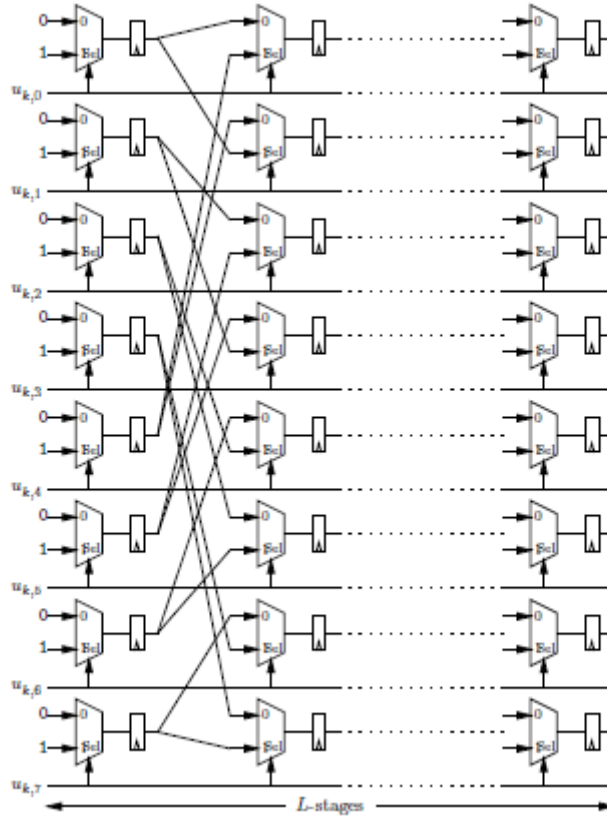


Figure 5-3: Block diagram of register exchange unit

5.2.2 Decode Stage

The decode stage of depth U performs Viterbi decoding and reliability updates on the metric difference values obtained at the output of the trellis stage. Reliability updates are performed on the U reliability values on the survivor path $L_{K-L-1}^S, L_{K-L-2}^S, \dots, L_{K-L-U}^S$ with each reliability value being updated for up to U times. The output of the decode stage, multiplied by the hard decision bit, i.e. $\hat{u}_{k-L-U} \cdot L_{K-L-U}^S$ is the intrinsic output of the SOVA decoder.

5.3 Block diagram of the hardware architecture for a SOVA decoder

Based on the SOVA components in the previous section in this section the block diagram of the hardware architecture for a HR-SOVA decoder is as shown in Figure 5-4.

The trellis unit (TRU) performs the branch metric computation followed by the Add-Compare-Select (ACS) operations as described in Section 5.2.1. The Survivor Memory Unit (SMU) performs as the merge stage to determine the ML path at L

stages away. The Path Comparison Unit (PCU) has a RE unit with similar structure as that in the SMU and it performs the decode stage with depth U . Viterbi decoding is performed using hardware decision bits that are stored in the First In First Out (FIFO) memory (FIFO U) as shown in the figure. There are two FIFOs required in the architecture; the first of which (FIFO U) is used to store the hard decision bits u_k decoded by the trellis unit and the second (FIFO Δ) is used to store the metric difference Δ_k computed by the trellis unit. For each stage, u_k and Δ_k for all 2^V states will be stored. The SMU and PCU are made up of columns of RE units. Each row of RE registers stores the hard decision sequence of the respective state.

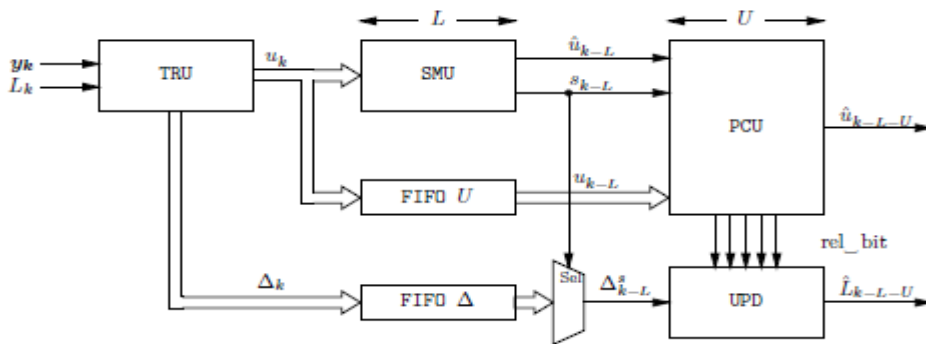


Figure 5-4: System architecture of SOVA decoder

The reliability updates are performed by the UPD module. The UPD module consists of U units of UPE elements that update and store the reliability values L_{K-L-j}^S at each stage of the decoding, based on the survivor and concurrent hard decision bit sequences. As reliability updates require survivor and concurrent path decision bits (u_{K-L-j}^C and u_{K-L-j}^S) for comparison before deciding if an update is needed, the PCU has additional logic to provide these relevance bits.

In order to obtain the survivor and concurrent paths, the SMU will first determine the ML state s_k by selecting the largest state metric Γ_i from the trellis unit as shown in Figure 5-5.

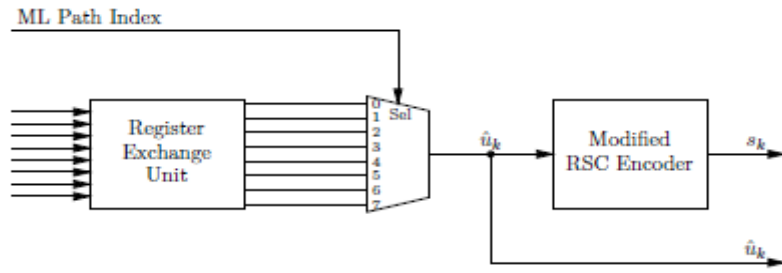


Figure 5-5: Block diagram of SMU module

Very often in practical designs, finding the largest state metric is not practical, and since it can be assumed that after L stages, the trellis has merged, any one of the N outputs of the register exchange unit can be used instead of the state chosen by the ML path index. The corresponding decision bit \hat{u}_{k-L} is output from the RE unit, and the associated state at L stages away (end of SMU) s_{k-L} can be determined by using an encoder. The encoder used to determine s_{k-L} is illustrated in Figure 5-6.

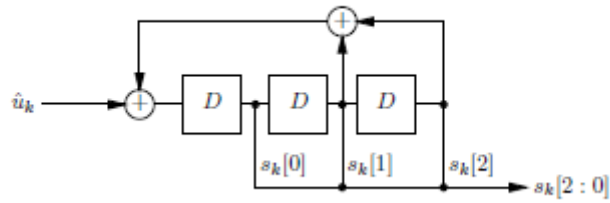


Figure 5-6: Encoder to determine state s_{k-L}

With s_{k-L} and \hat{u}_{k-L} determined, these two inputs are then used to select the desired rows from the RE in the PCU that correspond to the survivor and concurrent path sequences of decision bits. The selection is performed by letting \hat{v}_{k-L} represent the complementary decision bit of \hat{u}_{k-L} .

That is,

$$\hat{u}_{k-L} = 0 \Rightarrow \hat{v}_{k-L} = 1$$

$$\hat{u}_{k-L} = 1 \Rightarrow \hat{v}_{k-L} = 0$$

The previous transition state of s_{k-L} will be

$$s_{k-L-1}^s(\hat{u}_{k-L}) \xleftarrow{\hat{u}_{k-L}} s_{k-L}$$

$$s_{k-L-1}^c(\hat{v}_{k-L}) \xleftarrow{\hat{v}_{k-L}} s_{k-L}$$

and thus survivor and concurrent states S_{K-L-1}^S and a row select signal $SC = S_{K-L-1}^C$ can be obtained. A block diagram of the PCU is shown in Figure 5-7. By means of 8-to-1 MUXes, the SC signal selects the row in the RE network that corresponds to the set of concurrent path bits ($u_{K-L-1}^C, u_{K-L-2}^C, \dots, u_{K-L-U}^C$). The relevance bits are then determined by computing the XOR result of the decision bits of the survivor and the concurrent sequences. Thus, a relevance bit of '0' means that the survivor and concurrent bits at the stage are the same, i.e. $u_{K-L-j}^S = u_{K-L-j}^C$, and conversely a relevance bit of '1' implies that the bits are different ($u_{K-L-j}^S \neq u_{K-L-j}^C$).

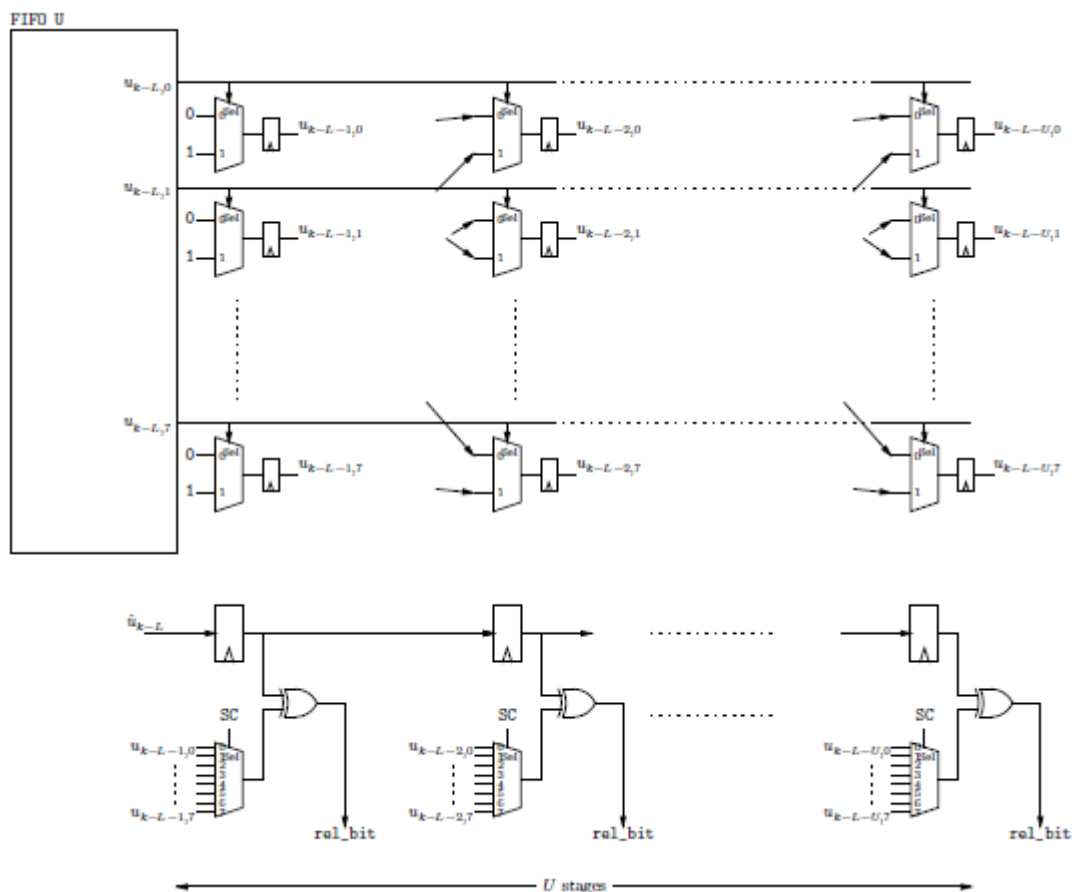


Figure 5-7: PCU for SOVA

In SOVA, the update rule is only applied when the survivor and concurrent bits are different. Therefore the relevance bits generated in the PCU are used in the UPD to indicate if reliability updates are required for each of the U stages. The UPD module consists of U units of UPE, with each UPE element responsible for checking a relevance bit from the PCU and to decide if an update is required. If an update is

required, the reliability value stored in the previous UPE stage is then compared against Δ_{k-L}^s that is selected from the FIFO delta using s_{k-L} . The block diagram of the UPD is as shown in Figure 5-8

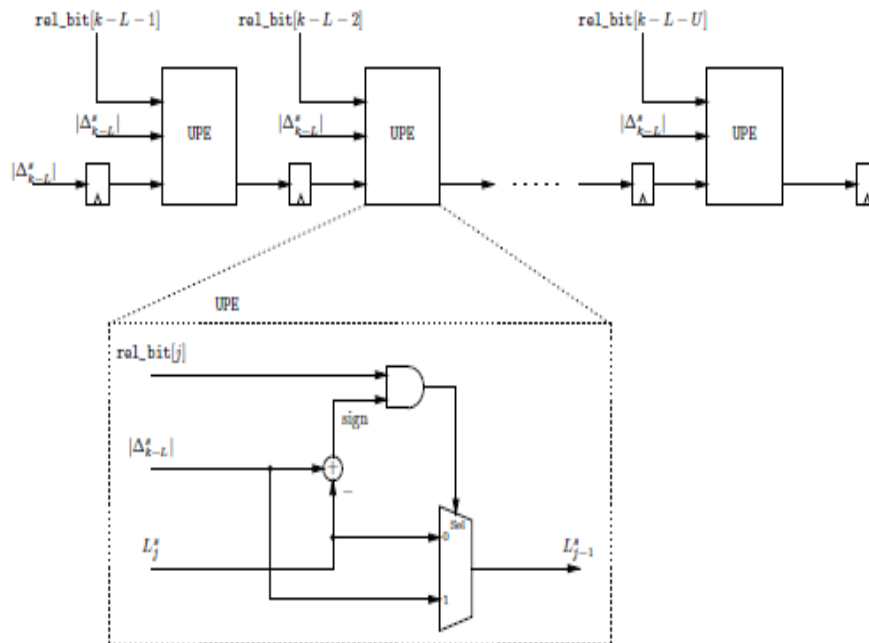


Figure 5-8: Block diagram of UPD module

5.4 Sliding Window

Sliding window implementations of decoding algorithms are used to reduce the memory requirements in turbo decoders. The performance of turbo codes depends heavily on frame length and deteriorates rapidly as the frame length decreases as shown in chapter 4. A long frame length, however, means a long decoding trellis for which the memory requirements as well as decoder complexity are excessive from an implementation viewpoint. Sliding window or finite-length window decoding can significantly reduce the memory requirements and the complexity of the decoder.

In trellis based decoding, the number of trellis stages required to make reliable decisions determines the length of the decoding window and is referred to as the decision depth D . Forward recursion in forward SOVA starts by building the first D stages of the trellis. This is followed by SOVA traceback at each stage of the trellis in

the current window. After the SOVA traceback the decoded bit at the first stage of the trellis is released and the decoding window slides forward by one trellis stage. The decoded bit at the second trellis stage is released in this window followed by another slide of the window and so on as shown in Figure 5-9 [25].

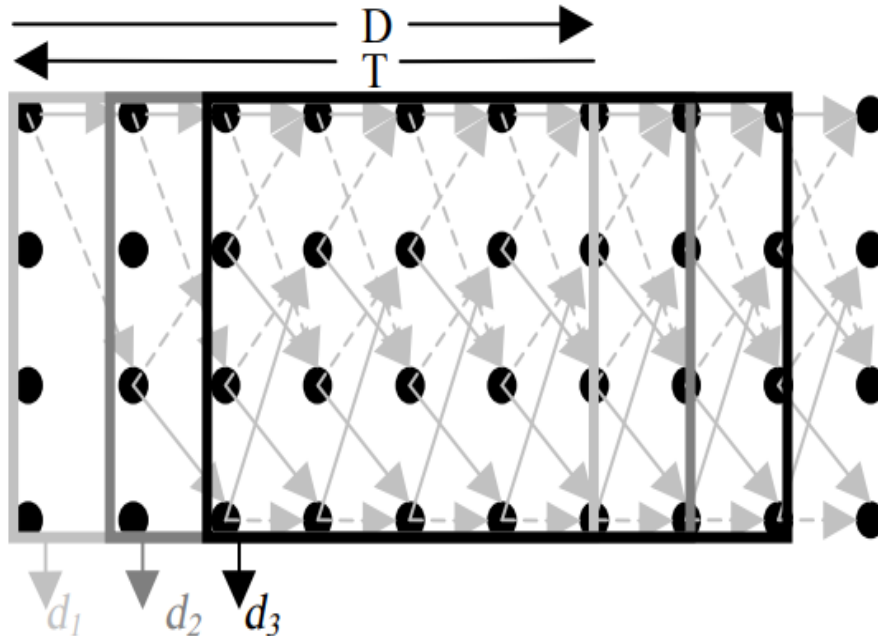


Figure 5-9: One bit releases sliding window decoding

Backward SOVA operates in the similar fashion, the only difference is that it starts from the last stage in the trellis and moves in the opposite direction, thus releasing the bits in reverse order.

The main objectives of using the sliding window technique are to reduce the decoding delay and storage requirements at the expense of a slight loss in performance. The overall code block at the receiver is divided into smaller sub-blocks (windows) of length W . The decoder output can already be calculated for the first block of information bits (trellis sections) within this window, thus reducing decoding delay significantly for large block lengths. When the first window is decoded, the decoder window is then shifted by one bit to the right in order to decode the next window.

It is worth to be mentioned that the traceback (merge (L)) stage and the update (decode (U)) stage can use different window sizes. In order to choose the depth of each stage, simulations were done to investigate the performance of turbo decoder for different window sizes.

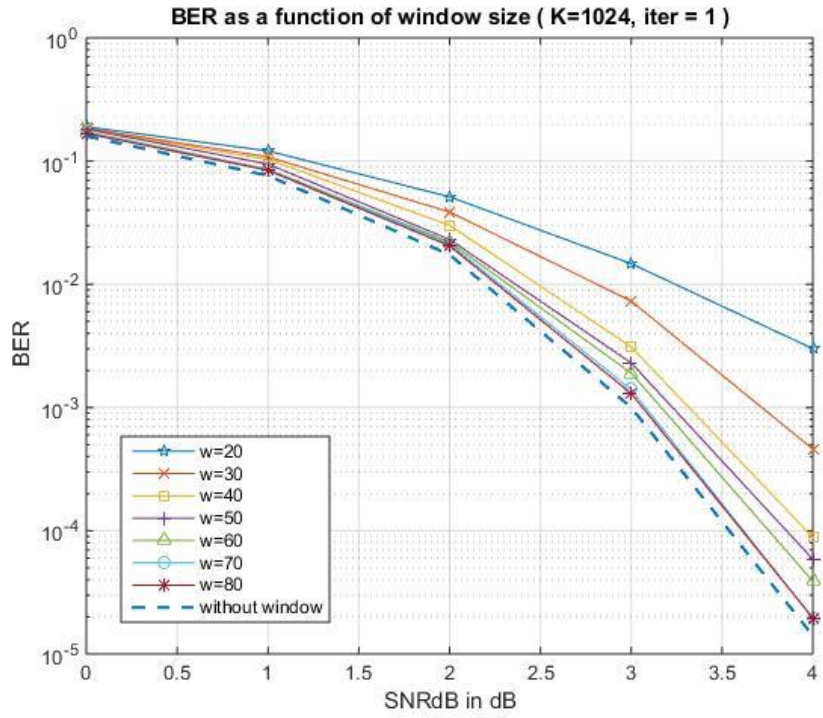


Figure 5-10 BER for frame size K=1024 over AWGN Channel using 1 iteration

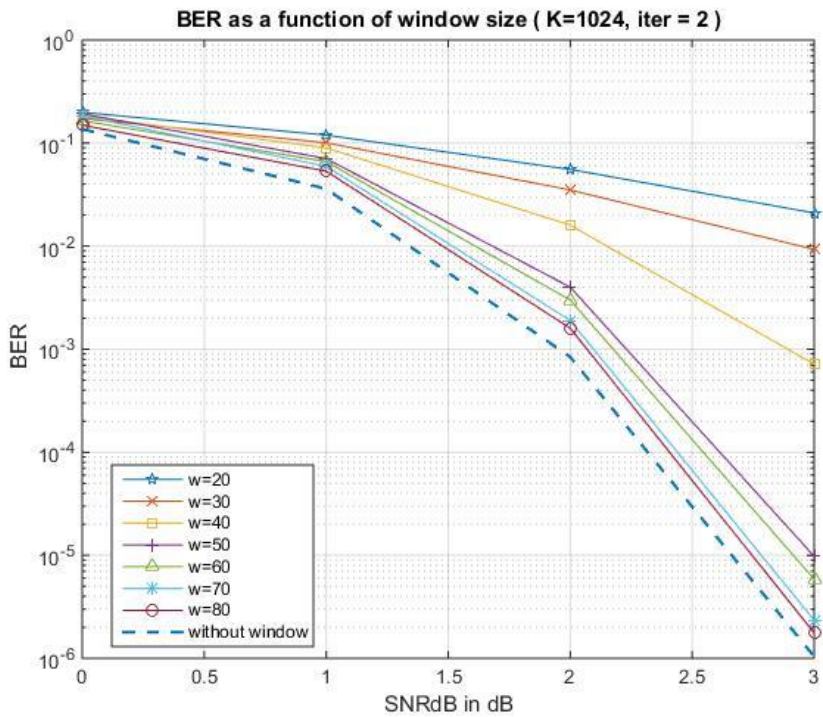


Figure 5-11 BER for frame size K=1024 over AWGN channel using 2 iterations

Figures 5-10, and 5-11 show the BER performance when the traceback window size changes. The frame length used is $K=1024$. The window sizes range is from $W=20$ to $W=80$. As shown in these figures, the performance converges to the original one (without using sliding window technique). The increment in performance is getting smaller as the window size increases. So $L=70$ was chosen since there's no significant improvement in performance for larger windows.

Similarly, simulations were done to design the optimum window size for update stage. $U=30$ was chosen. Hence the total latency (the number of clock cycles taken by our SOVA decoder before releasing the first bit) is 100 clock cycles

5.5 Quantization

Much of the work on turbo decoding assumes that the decoder has access to infinitely soft (unquantized) channel data. In practice, however, a quantizer is used at the receiver and the turbo decoder must operate on finite precision, quantized data. Floating-point units would make the hardware more complex, so quantization leads to simpler hardware design. In addition, the number of bits used to represent the quantized data is considered an important factor to the process of optimizing the HDL design since it represents the width of the memories and data buses used in our design.

This section discusses the process of converting from a floating-point simulation model of a turbo decoder to quantized decoder representative of one that could be implemented in hardware. One approach to quantization is presented here [26]. Performance measurements are done and the word size is designed according to the proper quantization.

5.5.1 Quantization Process

For simulation purposes the output of the encoder is generally taken to be a ± 1 value. After being transmitted through a noisy channel, these values are still distributed around ± 1 . When the input to a quantized system is received, it must be scaled up by some amount. Scaling must be performed to retain some of the important information that would be lost in a floating-point to integer conversion of the input value. For example, if the transmitted data is -1 and the received value is -0.4 , there is still significant probability information in the received value. Standard floating-

point to integer conversion of -0.4 would yield 0 . Since a 0 has equal probability of being a $+1$ or a -1 , all received probability information is lost. Conversion of -0.4 to -1 would indicate too high of a probability that a -1 was transmitted. By simply multiplying the input by some fixed value, we move the distribution to the new-scaled value as shown in figure 5-12. This allows us to better approximate the floating-point system in our quantized implementation, without losing as much of the probability information. The following figure illustrates the effect of scaling on the received values, in a Gaussian channel.

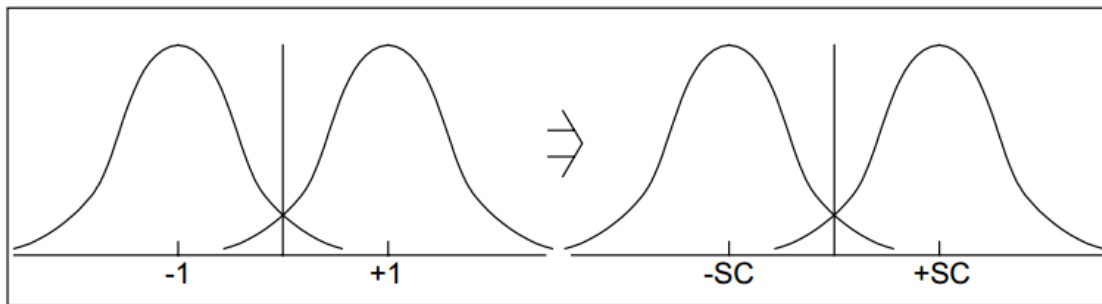


Figure 5-12 Scaling the received value in quantization process

To select the proper input-scaling factor, we must consider the quantization of the inputs. The input quantization and the input scale factor define where the received distribution is saturated. (Note: since we are using the log-likelihood ratio for the input to the decoder, the received distribution is also scaled by the signal-to-noise ratio. The following diagram illustrates how the saturation limits, defined by our choice of quantization, determine how much of the distribution will be clipped or saturated at the max/min values

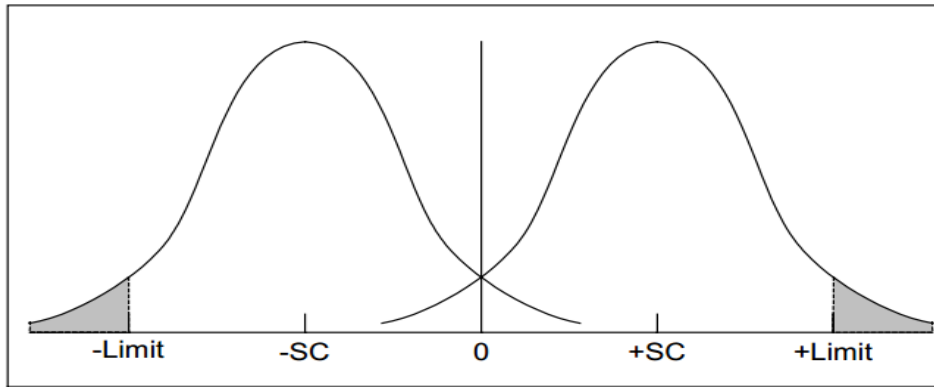


Figure 5-13 Scaling Factor and Limits for quantization process

We can adjust the input scale factor sc and our clipping limits cl (a clipping limit will be imposed by our limited bit width on the input signal, discussed later) as shown in figure 5-13, to provide an estimated amount of distribution clipping. The clipping estimate accounts for the amount of clipping associated with the adjacent limit. The amount of clipping due to the limit of the opposite sign is negligible. The following equations calculate the amount of clipping

from both quantization limits. The percent of values clipped by adjacent limit is given by:

$$Q\left(\frac{cl-sc}{sc}\right) * 100 = pcla. \text{ The percentage of values clipped by the opposite limit is}$$

$$Q\left(\frac{cl+sc}{sc}\right) * 100 = pclo.$$

For example if we are using a 6-bit quantization on the input, +31 to -32, and we want no more than 5% of the values to be clipped we would use a scaling factor of:

$$pcla = Q\left(\frac{31-sc}{sc}\right) * 100 = 5\% \text{ which yields a scale factor } sc = 11.7. \text{ The amount of}$$

$$\text{clipping resulting from the opposite sign limit is } pclo. = Q\left(\frac{31+11.7}{11.7}\right) * 100$$

The limit at which an integer variable is forced to saturate defines the effective quantization (or number of bits required).

By defining saturation limits for the inputs and all internally calculated values, we can approximate the performance of a hardware implementation's specific bit widths. After any calculation for a particular variable, saturation logic is added.

After multiplying the data with the proper input scaling and defining the saturation limits, the floating point data could be rounded to the nearest integer value. Hence, these integer values can be used directly to be decoded using our turbo decoder.

5.5.2 Choosing the proper word size

In order to choose the optimum number of bits to represent the quantized data, simulations were using different quantization bits. The performance of Turbo decoder was measured, hence the proper word size was selected such that it achieves BER very close to the BER resulted from using unquantized (floating point) data.

Fig 5-14 shows an example for the simulations done to design the word size. The BER curves are plotted for different word sizes, using a fixed frame size $K = 1024$ over an AWGN channel. The SNR range was used from 0 to 5 dB. The number of decoder iterations was chosen to be two iterations. It's shown that increasing the number of quantization bits improves the performance of the turbo decoder. The BER curves converges to the one represents the unquantized turbo decoder.

The BER values are given in Table 5.1. It can be noticed that 8 bits for quantization is sufficient, since BER doesn't fall significantly for larger word sizes.

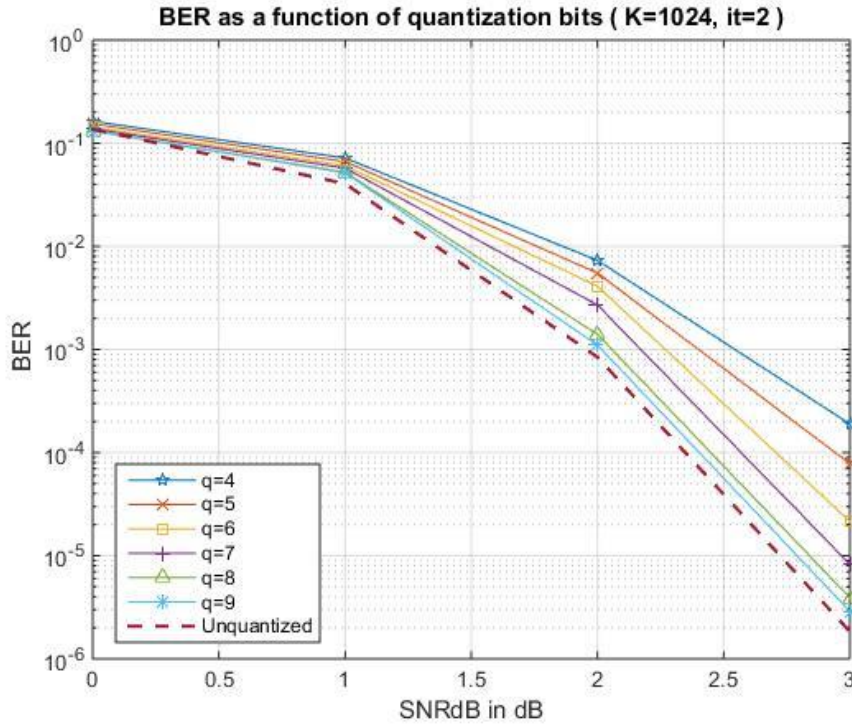


Figure 5-14 BER for frame size K=1024 over AWGN Channel for different word sizes

Table 5.1: BER values for different word sizes using frame size K=1024

	SNR = 0	SNR = 1	SNR = 2	SNR = 3
q = 4	0.1634	0.0718	0.0073	1.8945e-04
q = 5	0.1540	0.0663	0.0055	7.805 · e-05
q = 6	0.1432	0.0605	0.0041	2.1484e-05
q = 7	0.1375	0.0573	0.0027	8.2484e-06
q = 8	0.1314	0.0524	0.0014	3.9063e-06
q = 9	0.1307	0.0516	0.0011	2.8863e-06
Unquantized	0.1368	0.1368	0.1368	1.8391e-06

Chapter 6: Hardware Implementation and Results

This chapter gives performance results of the HDL implementation of the SOVA decoder design outlined in Chapter 3. Then the implementation of the complete turbo decoder is presented in details. The HDL version of the decoder was compared with a given reference decoder written in MATLAB, which was known to be accurate.

The Turbo decoder using proposed algorithm was implemented using Verilog hardware description language, which offers high abstraction level during the implementation. Designs are completed with the Integrated Software Environment (ISE), which is a software suite developed by Xilinx that allows designers to take their designs from design entry through FPGA device programming. The Verilog description was synthesized using XST (Xilinx Synthesis Tool) on Spartan-6 FPGA SP605 Evaluation Kit.

6.1 Design Flow in ISE

The ISE manages and processes a design through the following steps in the ISE design flow [28].

6.1.1 Design Entry

Design entry is the first step in the ISE design flow. During design entry, the design source files can be created based on the design objectives using a Hardware Description Language (HDL), such as VHDL, Verilog, or ABEL, or using a schematic. Multiple formats for the lower-level source files are also supported in design entry.

6.1.2 Synthesis

After design entry and optional simulation, Xilinx Synthesis Technology (XST), integrated in ISE, synthesizes VHDL, Verilog, or mixed language designs to create Xilinx specific netlist files. Then they are accepted as input to the implementation step.

6.1.3 Implementation

After synthesis, ISE design implementation converts the logical design into a physical file format that can be downloaded to the selected target device. The implementation process includes four major steps: *Translate*, which merges the incoming netlists and constraints into a Xilinx design file; *Map*, which fits the design into the available resources on the target device; *Place and Route*, which places and routes the design to the timing constraints; *Programming file generation*, which creates a bitstream file that can be downloaded to the device

6.1.4 Verification

A design can be verified at several points in the design flow. The integrated ISE simulator or ModelSim software can be used to verify the functionality and timing of a design or a portion of the design. These simulators interpret VHDL or Verilog code into circuit functionality and displays logical results of the described HDL to determine correct circuit operation. In-circuit verification can also be carried out with the Chipscope software, also provided by Xilinx, after programming the FPGA device.

6.1.5 Device Configuration

After generating a programming file, it is downloaded from a host computer to a Xilinx device on a development board. The Spartan-6 XC6SLX45T-FGG484-3C FPGA on SP605 Kit is used for in-circuit verification and BER testing. This device belongs to the Spartan-6 FPGA family. The designing and testing flow are shown in the figure below.

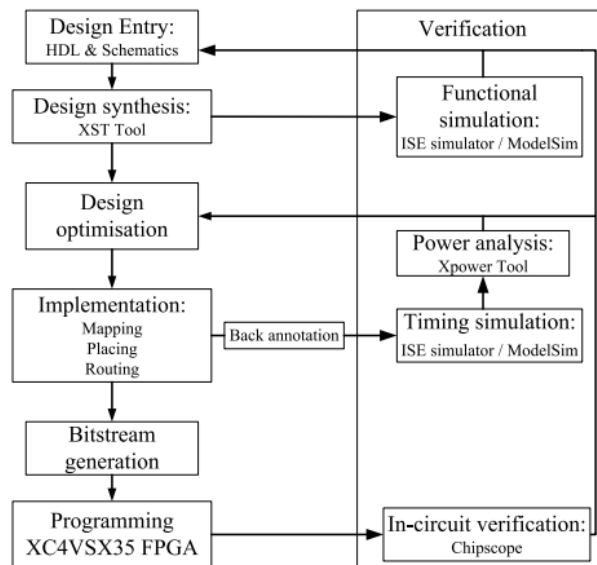


Figure 6-1: Design and verification process of the FPGA implementations

6.2 Testing framework for SOVA Block

Since SOVA is the main block in the turbo decoder, it was implemented using Verilog and tested alone to make sure. Testing SOVA was really required to point out the defects and errors that were made during the development phases and to ensure that it should not result into any failures because it can be very expensive to fix in the later stages of the turbo decoder development.

Figure 6.2 describes the interfaces of the SOVA decoder, and the used signal names and Table 6.1 defines them.

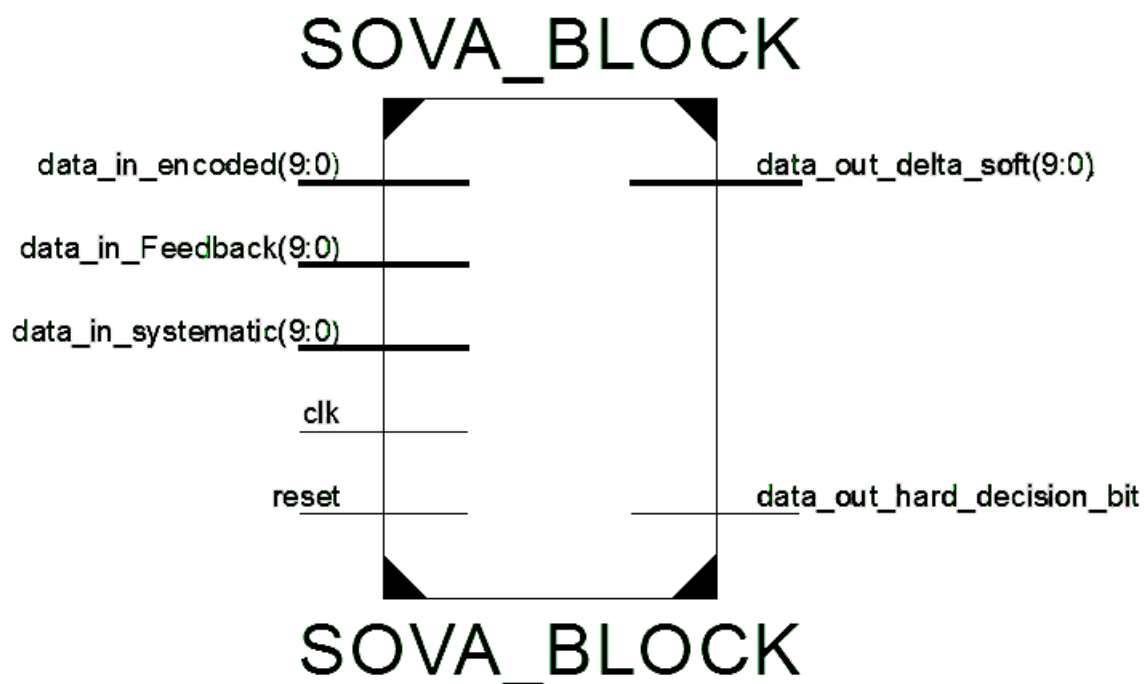


Figure 6-2: Interface of SOVA Block

Table 6.1: Description of SOVA Signals

Pin	Direction	Description
Data_in_encoded	Input	RSC encoded data: is the quantized version of the received convolutional sequence of the RSC encoder after it's sent over the channel
Data_in_feedback	Input	RSC 1 encoded data: is the quantized version of the extrinsic sequence produced by the previous SOVA component decoder.
Data_in_systematic	Input	Systematic data: is the quantized version of the

		received systematic sequence of the RSC encoder after it's sent over the channel
Clk	Input	Clock: All synchronous operations occur on the rising edge of the clock signal.
Reset	Input	Reset: is an active-high, asynchronous resets all the registers inside the SOVA
Data_out_delta_soft	Output	Soft Output: the soft information output (LLRs) that will be passed to the next SOVA
Data_out_hard	Output	Hard Output: the hard decision sequence of the decoded data

The SOVA decoder is designed to have 5 main blocks as shown in chapter X. Each block is described in structural RTL style in separate Verilog files and combined structurally in one file which described the exact connections and signal names as shown in the RTL schematic in fig 6.3.

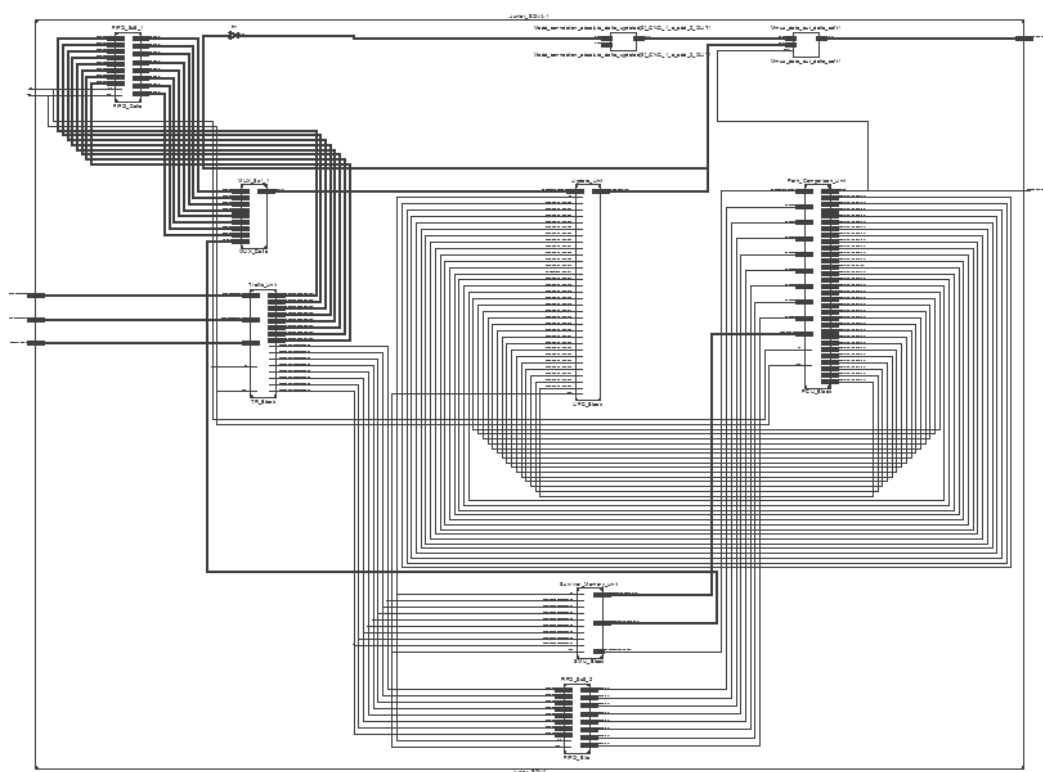


Figure 6-3: RTL Schematic of SOVA Module

6.3 Simulation Results of Behavioral RTL design

The results in figures 6.4-6.5, are from the case where the frame length is equal to 1024. The quantized systematic and encoded data are generated from MATLAB and stored in files. The input files are read by the SOVA test bench to be used in simulation

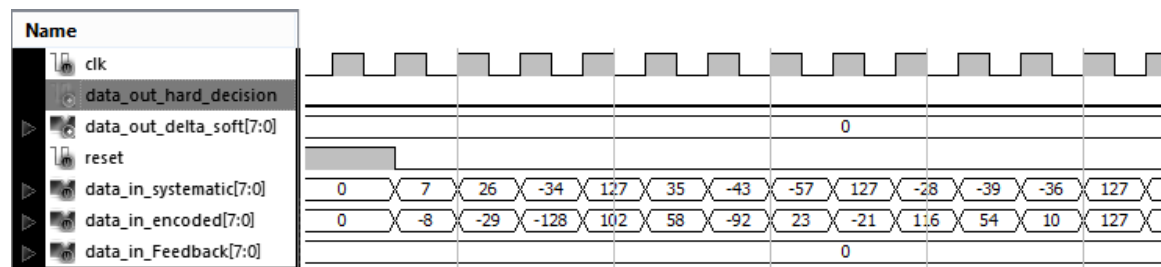


Figure 6-4: Loading SOVA Block with Input Data

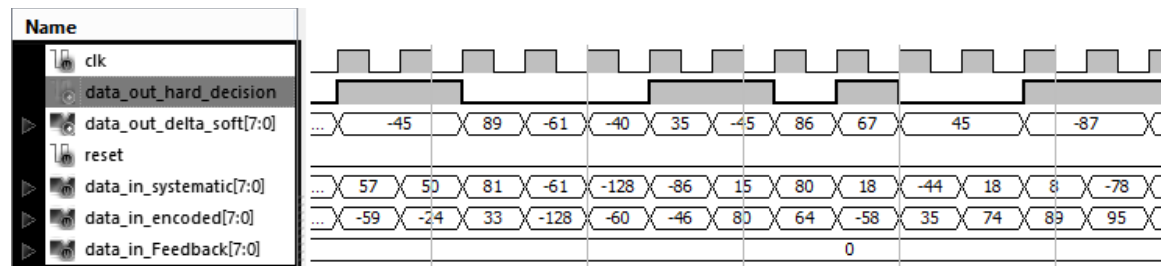


Figure 6-5: Output of SOVA Block

6.4 Comparison with Software Reference

The Verilog and MATLAB decoders were compared by simulating using frame length of 1024 over an AWGN channel. The SNR range was used from 0 to 5 dB. 8 bits for quantization are used, traceback window of length = 70, and update window if length = 30 are used.

MATLAB was used to generate noisy, encoded streams of data, and each decoder used a common traceback. A bit error rate (BER) comparison was done using only the information bit decisions, meaning the reliability outputs of each decoder were mapped to 0's and 1's before being compared for equality. Figure 6.6 shows the comparison in BER performance.

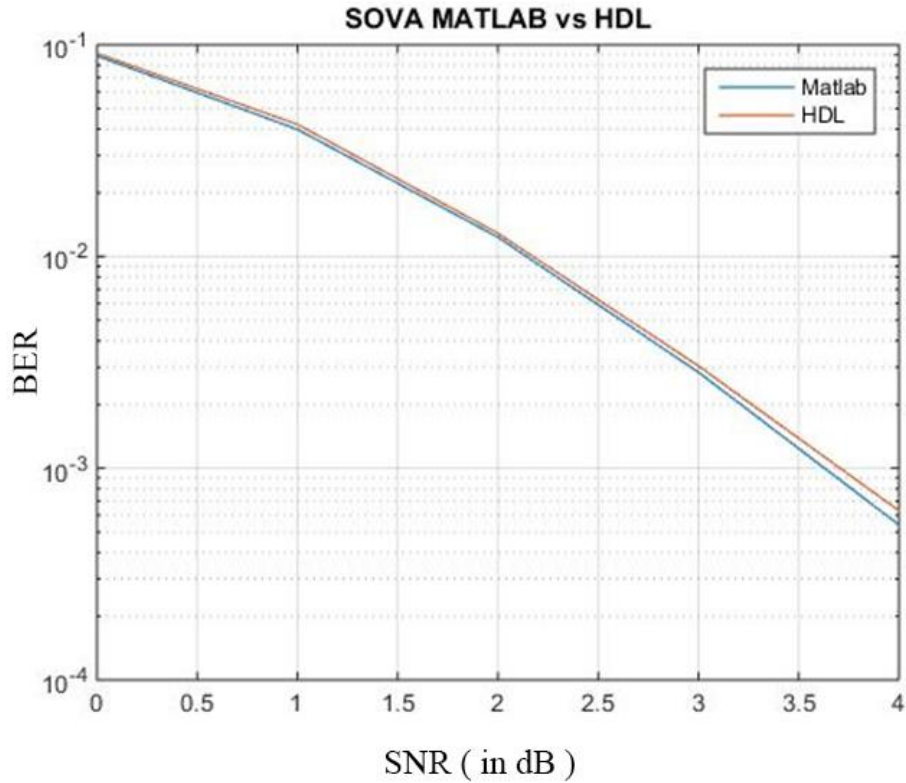


Figure 6-6: Performance of SOVA

6.5 Hardware Performance

The Verilog used to define the SOVA decoder was synthesized, mapped, and routed. The processing was done using the Xilinx ISE design tools. A user constraint was defined for the clock signal to have a period of 6ns with a 50% duty cycle. This forced ISE to work harder in its attempts to find the maximum clock frequency.

Tables 6.2 shows the hardware results obtained in the building of the designs. We see in this table that the overall footprint of the SOVA decoder is relatively small, with all builds using about 10% of the available LUTs.

Table 6.2: FPGA Resources of SOVA Module

Device Utilization Summary (estimated values)				[1]
Logic Utilization	Used	Available	Utilization	
Number of Slice Registers	1669	54576	3%	
Number of Slice LUTs	2919	27288	10%	
Number of fully used LUT-FF pairs	1502	3086	48%	
Number of bonded IOBs	43	296	14%	
Number of BUFG/BUFGCTRLs	1	16	6%	

6.6 Turbo Decoder implementation

6.6.1 Scheduling of Computation at block level

The encoded data is sent in a sequence of ... systematic, parity1, parity2, systematic, parity1, parity2, ... which gives a rate of 1/3 as specified in the 3GPP LTE standard. The received sequence has to be demultiplexed to SOVA1 and SOVA2. SOVA1 expects to work on systematic, and parity1 bits, while SOVA2 expects to work on parity2 and the interleaved version of systematic as shown in fig 6.7.

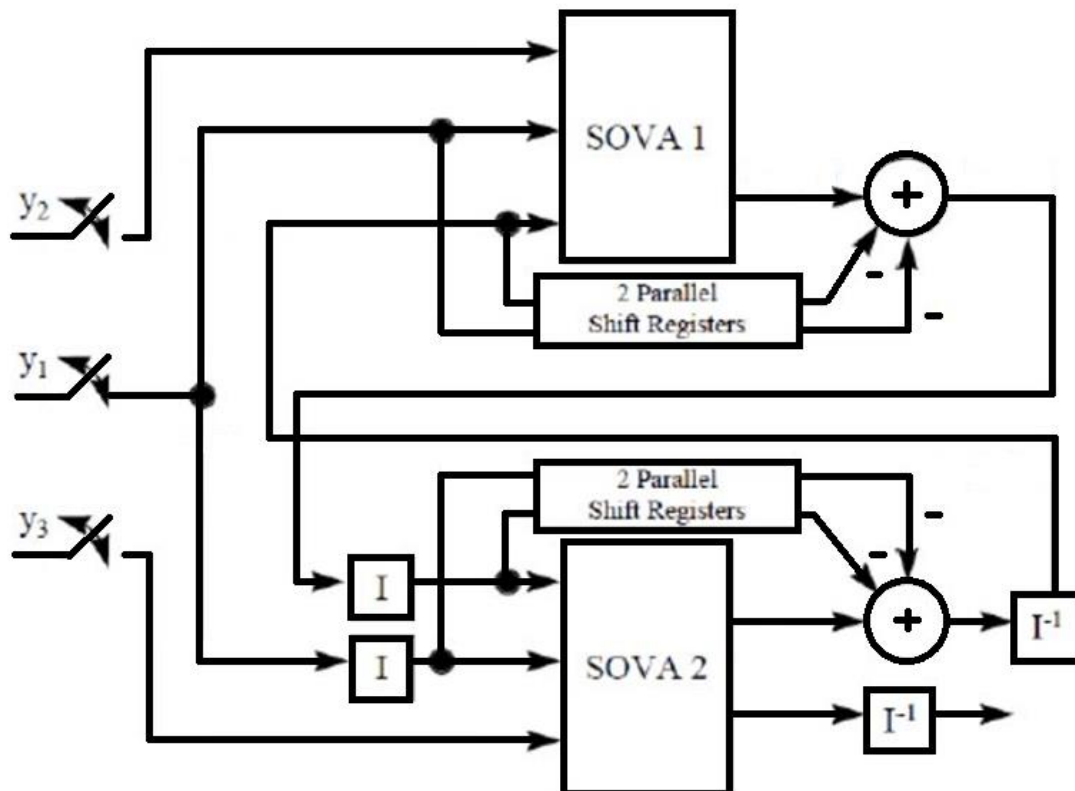


Figure 6-7: Turbo Decoder Architecture

Each block of data needs to be processed through the SOVA Blocks number of times equals to the number of iterations specified for the turbo decoder. Due to data dependency, the processing needs to be sequential, that is SOVA1 iteration 1 produces the results for SOVA2 iteration 1, SOVA2 iteration 1 produces the results for SOVA1 iteration 2, and so on.

Thus it's obvious that while SOVA1 is processing some data, SOVA2 is idle and waiting for SOVA1 to finish. The same happens for SOVA1 when SOVA2 is processing the data. Thus pipelining across the data blocks has to be implemented

with the cost of using more hardware resources and introducing more output delay. However, we preferred to simplify the design, so instead of using two blocks for SOVA decoder, only one block is used to save almost half of the hardware resources. The new connections are shown in fig 6.8.

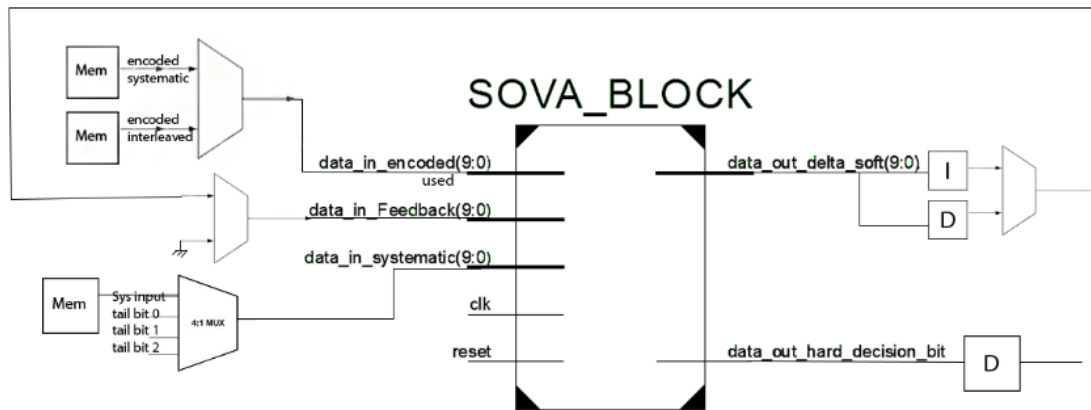


Figure 6-8: Implementation of Turbo Decoder

The top level of the design shown in Fig. 6.8 shows that the turbo decoder is designed to have 4 main blocks. Brief descriptions for each block is shown as follows:

1. SOVA block
2. Memories.
3. Interleavers, Deinterleaver.
4. Control Unit

The central control unit is used to generate the control pulses to synchronize the operation of the input buffer, SOVA, and memories. It also generates “select” signal to alternate between the interleaved and the original versions of data blocks back and forth at every half iteration. Its design is very simple. It has an internal counter used as a Timer to generate these control signals and these addresses at specific times.

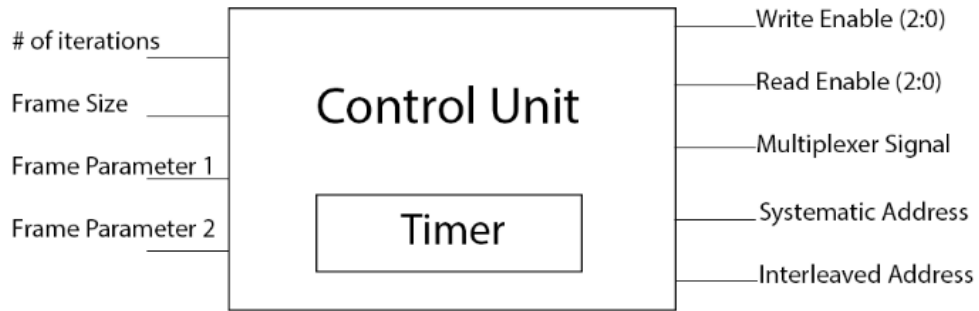


Figure 6-9: Control Unit of Turbo Decoder

The Interface between these modules and the control unit is implemented and also shown in the next RTL schematic

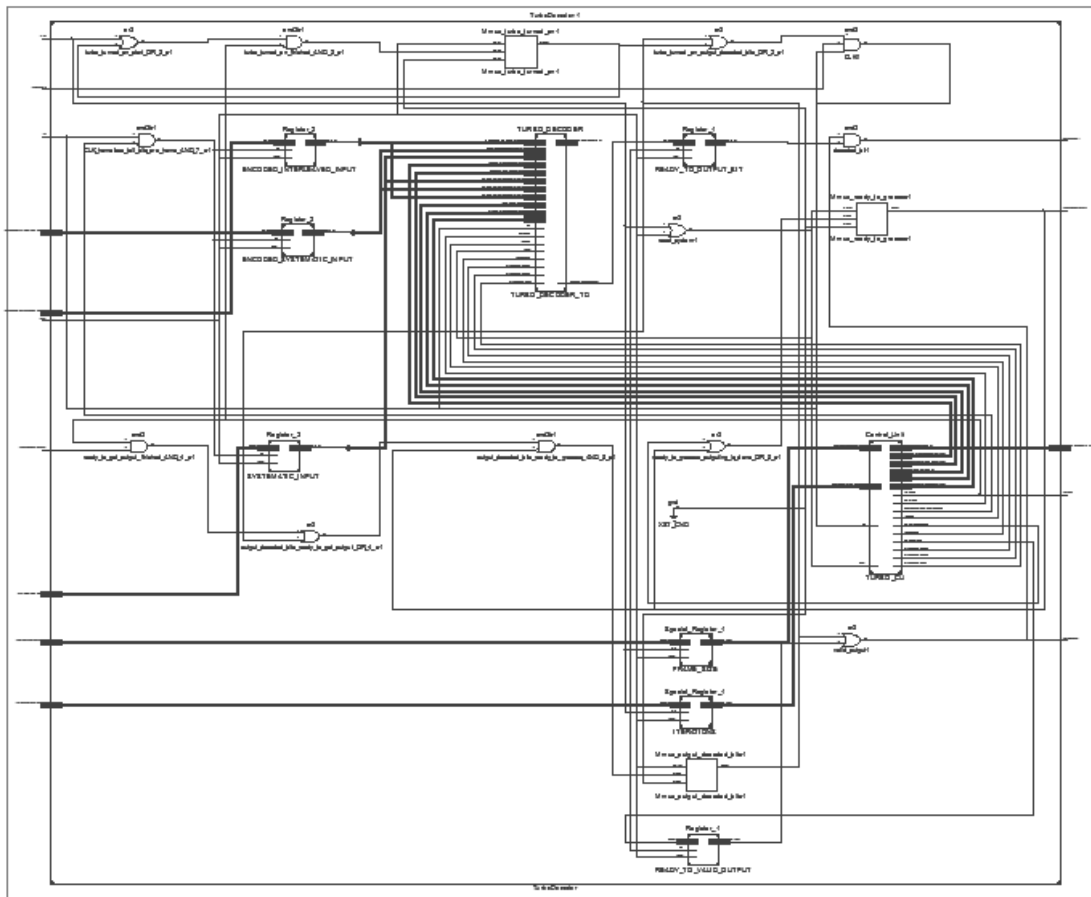


Figure 6-10: RTL Schematic of Turbo Decoder

6.6.2 Turbo Decoder as a black box

After implementing the proposed turbo decoder, the interface was chosen carefully such that that module could be used properly without resulting in any errors or failures. Figure 6.11 describes the interfaces of the turbo decoder, and the used signal names and Table 6.3 defines them.

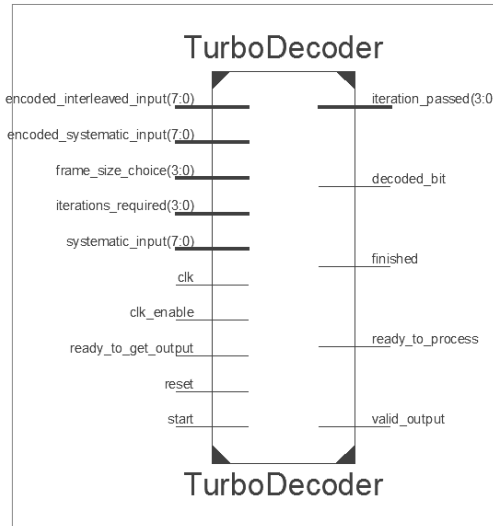


Figure 6-11: Interface of Turbo Decoder

Table 6.3: Description of Turbo Decoder Signals

Pin	Direction	Description
Systematic_Input	Input	Systematic data: is the quantized version of the received systematic sequence of the RSC encoder after it's sent over the channel
Encoded_Systematic_Input	Input	RSC encoded data: is the quantized version of the received convolutional sequence of the RSC encoder after it's sent over the channel
Encoded_Interleaved_Input	Input	RSC 1 encoded data: is the quantized version of the extrinsic sequence produced by the previous SOVA component decoder.
CLK	Input	Clock: All synchronous operations occur on the rising edge of the clock signal.
CLK_Enable	Input	Clock Enable: When deasserted (Low), rising clock edges are ignored and the core is held in its current state.
Reset	Input	Reset: A signal used to Reset the core to its initial state.

Start	Input	Start: A hand shaking signal used to indicate the start of input data reception
Frame_Size_Choice	Input	Frame Size Choice: Each supported Frame size has an ID number that should be given to the core with the high edge of Start Signal to inform the core with the required frame size.
Iterations_Required	Input	Number of Iterations Required: informs the core with the number of iterations to do on the data. The core supports from a single iteration up to 15 iterations which is a relatively big number.
Ready_To_Get_Output	Input	Ready to Receive Bits: A hand shaking signal used to inform the core that the user is ready to receive decoded bits.
Iteration_Passed	Output	Number of Iterations Passed: It is used to inform the user how many iterations has passed.
Finished	Output	Finished: A hand shaking signal that is set to high when the core finished processing.
Decoded_Bits	Output	Hard Output: the hard decision sequence of the decoded data
Valid_Output	Output	Valid_Output: A hand shaking signal which is set to high when data on Decoded_Bits is Valid to take.
Ready_To_Process	Output	Ready To Process A New Frame: A hand shaking signal that is Set to high when all decoded bits are output.

6.6.3 Additional Feature:

This interface has another feature that is added to make the core is absolutely configurable. The feature is that the core is outputting the decoded bits after each iteration –not just after the required number of iterations- and Setting the **Valid_Output** signal to indicate the outputting of these bits. This is very useful to minimize the time for the error correction stage. For example, let’s suppose that the

core is asked to do a 6 iterations on the frame and that after 3 iterations the Code Redundancy Check (CRC) – using this feature – has found that the output after the third iteration is error free so no need for waiting another 3 iterations to have this output. It can take it and send a signal to an earlier stage to reset the decoder and give it another frame to process on it.

6.6.4 Generic Parameters

Table 6.4 shows a list of parameters that can be adjusted before synthesis of the turbo decoder code.

Table 6.4: Description of Turbo Decoder module parameters

Parameter	Description
Data Bus Size	Width of the input data bus according to the number quantization bits.
Address Bus Size	Width of the address bus according to the size of the largest frame.

6.7 Simulation Results of Behavioral RTL design

6.7.1 Initialization and Data Input to Decoder

Figures 6.12 shows an important instance from simulation results which reveal how handshaking signals work to receive data. The Start signal needs to be set with the entering all configurations and input data of the frame.

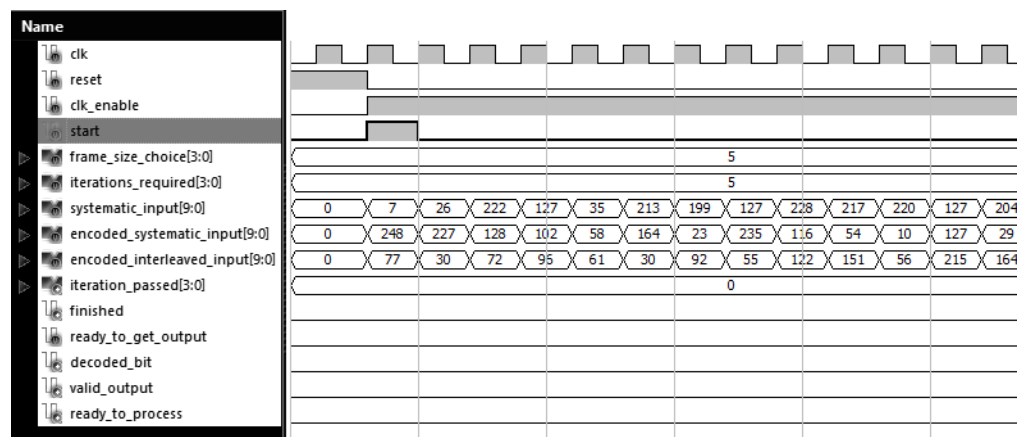


Figure 6-12: handshaking signals: Start

6.7.2 Decoder Output When Processing is Done

Figures 6.13-6.16 show 4 important instances from simulation results which reveal how hand shaking signals work to output decoded bits and also show an example of the additional feature added to the Turbo Decoder. The following figure shows the Finished Signal Set to 1 after finishing the required number of Iterations.



Figure 6-13: handshaking signals: Finished

Next Figure shows how the module using our core signaled that it is ready to get output and shows also how the core responded to it by outputting the decoded data and setting the Valid Signal.

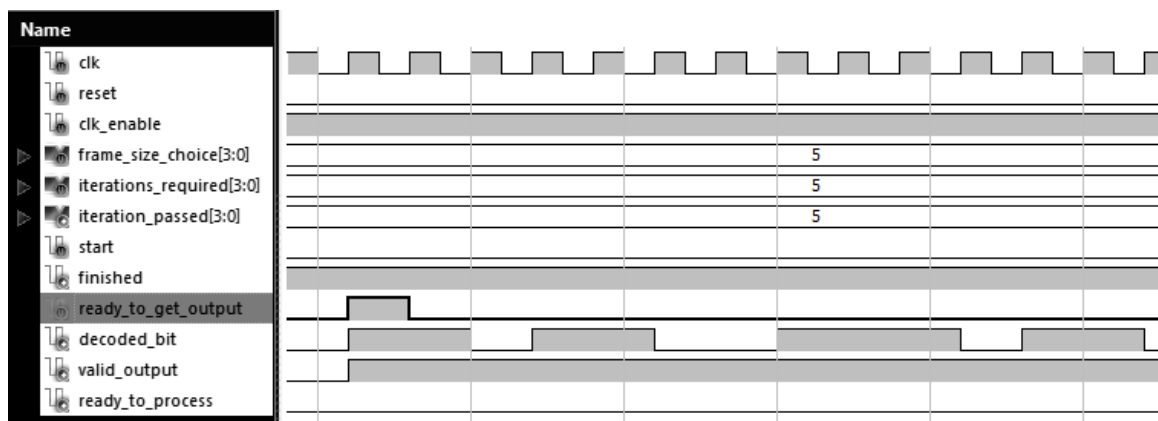


Figure 6-14: handshaking signals: Ready_To_Get_Output signal

The Figure 6.15 shows how the core acknowledges his availability to process a new frame after finishing outputting the decoded data after the required number of iterations.

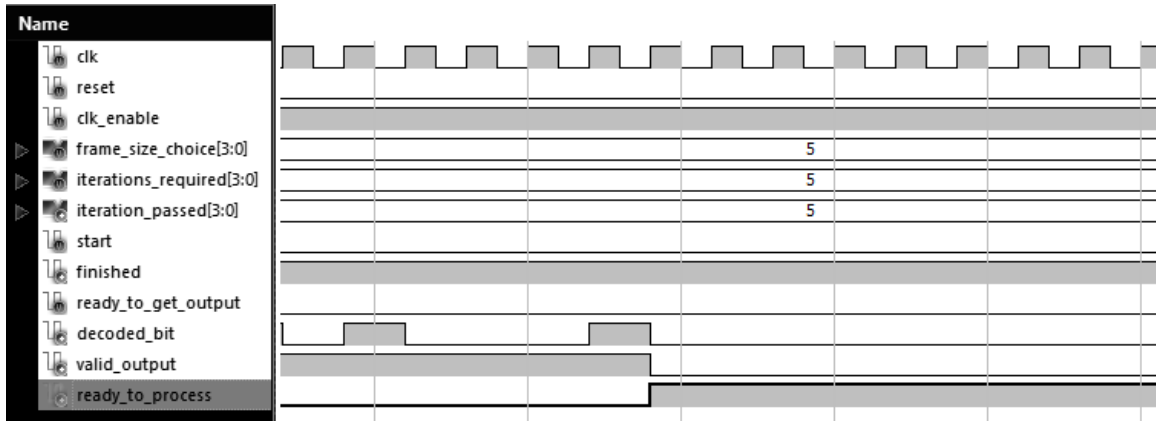


Figure 6-15: handshaking signals: Ready_To_Process signal

The Figure below shows how the signals work to provide the new feature stated before.

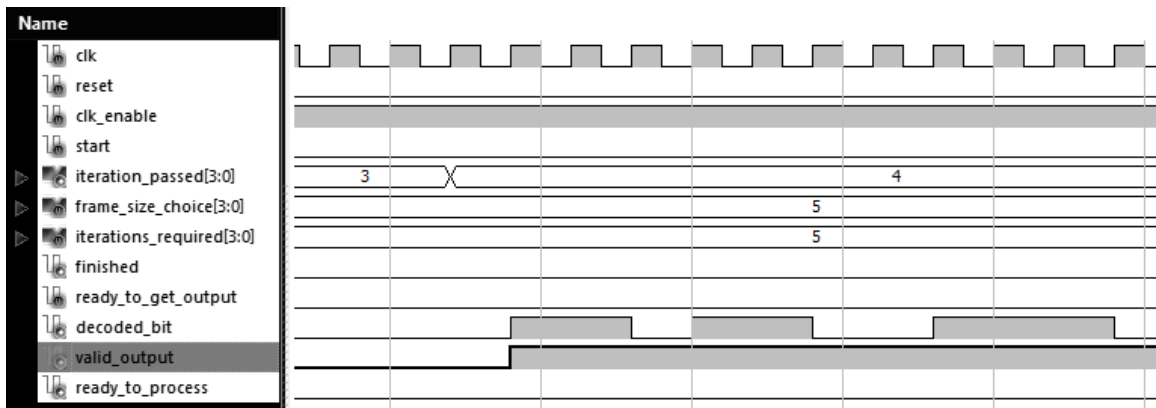


Figure 6-16: handshaking signals: Valid_Output signal

6.8 Performance and Resource Usage

The code has been extensively tested to optimize performance. A user constraint was defined for the clock signal to have a period of 27 ns with a 50% duty cycle. This forced ISE to work harder in its attempts to find the maximum clock frequency. Table 6.5 shows the resource requirements. These results have been obtained for the chosen parameters mentioned previously in chapter 4.

Table 6.5: FPGA Resources used by Turbo Decoder

Device Utilization Summary (estimated values)				[-]
Logic Utilization	Used	Available	Utilization	
Number of Slice Registers	1733	54576	3%	
Number of Slice LUTs	4300	27288	15%	
Number of fully used LUT-FF pairs	1346	4687	28%	
Number of bonded IOBs	45	296	15%	
Number of Block RAM/FIFO	26	116	22%	
Number of BUFG/BUFGCTRLs	3	16	18%	
Number of DSP48A1s	8	58	13%	

6.9 BER Performance

Figure 6.17 shows the difference between the BER performance in hardware and the MATLAB model. Marginal loss in performance is shown due to the finite length window and using quantization.

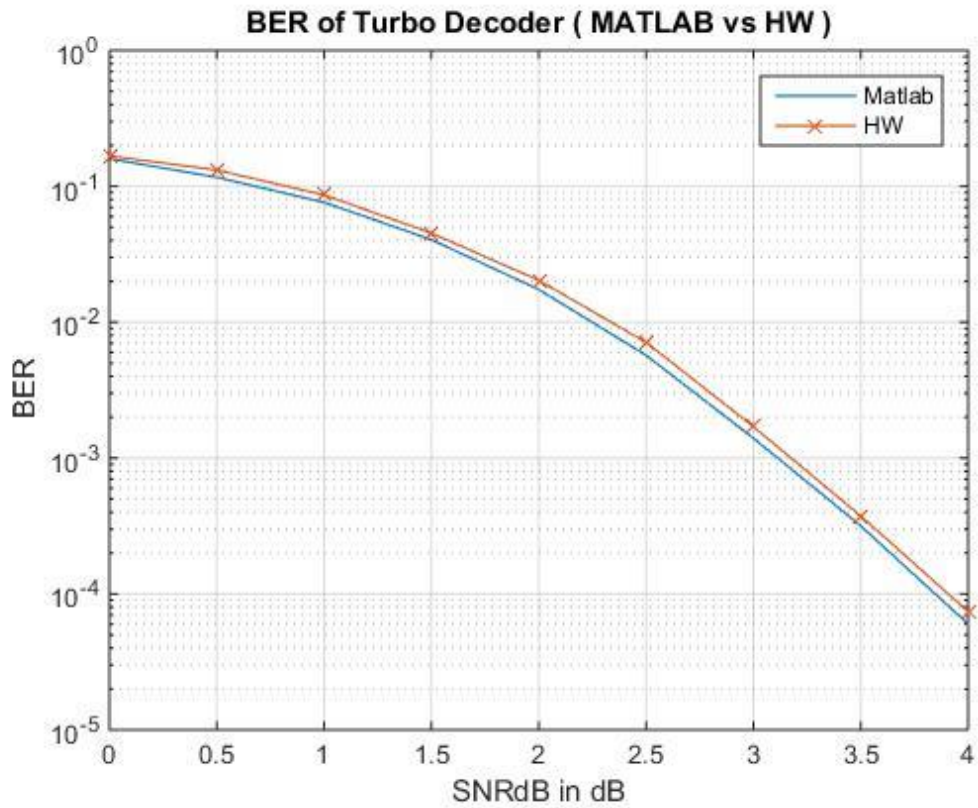


Figure 6-17: BER Comparison for frame size K=1024 using 1 iteration

Chapter 7: PCI Express Interconnect

This chapter presents an overview of the PCI Express architecture and key concepts. PCI Express is a high performance, general purpose I/O interconnect defined for a

wide variety of computing and communication platforms. Key PCI attributes, such as its usage model, load-store architecture, and software interfaces, are maintained, whereas its parallel bus implementation is replaced by a highly scalable, fully serial interface. PCI Express takes advantage of recent advances in point-to-point interconnects, Switch-based technology, and packetized protocol to deliver new levels of performance and features. Power Management, Quality Of Service (QoS), Hot-Plug/Hot-Swap support, Data Integrity, and Error Handling are among some of the advanced features supported by PCI Express. [29]

7.1 PCIe Link

A Link represents a dual-simplex communications channel between two components. The fundamental PCI Express Link consists of two, low-voltage, differentially driven signal pairs: a Transmit pair and a Receive pair as shown in 7-1.

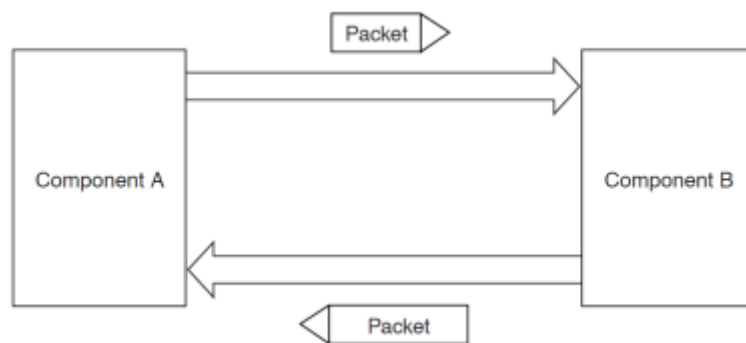


Figure 7-1 PCI Express Link

Some of the primary link's attributes are the following. The PCI Express link consists of dual unidirectional differential Links, implemented as a Transmit pair and a Receive pair. Once initialized, each link must only operate at one of the supported signaling levels. In the first generation of PCI Express technology, there was only one signaling rate supported, which provided an effective **2.5 Gigabits/second/Lane/direction** of raw bandwidth. The second generation provides an effective **5.0 Gigabits/second/Lane/direction** of raw bandwidth. The data rate is expected to increase in the future as the technology advances.

A link must support at least one lane and each Lane represents a set of differential signal pairs (one pair for transmission, one pair for reception). To scale bandwidth, a

Link must aggregate multiple lanes denoted by xN where N may be any of the supported Link widths. An x8 link represents an aggregate bandwidth of 20 Gigabits/second of raw bandwidth in each direction. The implementation of the hardware in this thesis is **PCIe 1.0 x1**. It should be noted that each link must support a symmetric number of Lanes in each direction, i.e., a x16 link indicates there are 16 differential signal pairs in each direction.

PCIe link speed is based on the generation and number of lanes in the link as shown in Table 7.1. [30]

Table 7.1:PCI Express different generations speed comparison

Generation	Raw Bit Rate	Bandwidth Per Lane Each Direction	Total x16 Link Bandwidth#
Gen 1	2.5 GT/s	~ 250 MB/s	~ 8 GB/s
Gen 2	5.0 GT/s	~500 MB/s	~16 GB/s
Gen 3	8 GT/s	~ 1 GB/s	~ 32 GB/s

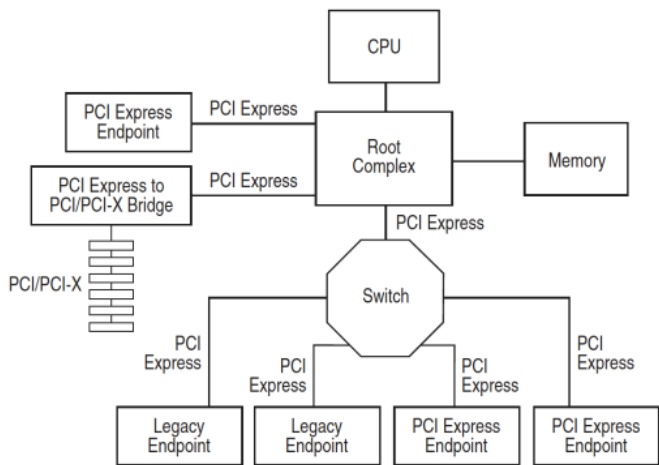
Link bandwidth in each direction x 2 (full duplex)

During hardware initialization, each PCI Express link is set up following a negotiation of lane widths and frequency of operation by the two agents at each end of the link. No firmware or operating system software is involved. [29]

7.2 PCIe Clock Recovery

At speeds starting at **2.5GHz**, the point-to-point architecture is still a challenge to get working because the duration of each bit is so short that timing jitter (the time uncertainty surrounding the arrival of each bit) becomes a problem. And even if each signal pair had an associated clock pair transmitted along with it, the clock pair would

also be
 jitter. So
 "clock
 Clock
 signal



subject to timing
 instead a new
 technique called
 recovery" is used.

recovery is simple.
 Basically, for each
 pair, the pair

receiver looks at the signal transitions (a bit 0 followed by a bit 1, or vice-versa), from which it can infer the position of surrounding bits. One problem is that if many successive bits are transmitted with the same value (like lots of 0's), no signal transition is seen. So extra bits are transmitted to ensure that signals transitions are not too far apart (which "re-synchronizes" the clock recovery mechanism).

The extra bits are sent using a scheme called **8b/10b encoding**, so that for each 8 bit of useful data, 10 bits are actually transmitted (a 20% overhead) in a specific way that guarantees enough signal transitions. But that also means that at 2.5GHz, we only get **250MB/s** of useful bandwidth per pair (instead of the 312MBps we would get without the encoding overhead), which results in 32-bit interface with 62.5 MHz clock

7.3 PCIe Fabric Topology

A fabric is composed of point-to-point Links that interconnect a set of components, an example fabric topology is shown in Figure 7.2. This figure illustrates a single fabric instance referred to as a hierarchy, composed of a Root Complex, multiple Endpoints (I/O devices), a Switch, and a PCI Express to PCI/PCI-X Bridge, all interconnected via PCI Express links.

Figure 7-2 PCI Express Topology

A Root Complex denotes the root of the I/O hierarchy that connects the CPU/memory

subsystem to the I/O. As illustrated in Figure 1.2, a Root Complex may support one or more PCI Express ports. Each interface defines a separate hierarchy domain. Each hierarchy domain may be composed of a single Endpoint or a sub-hierarchy containing one or more Switch components and Endpoints. Endpoint refers to a type of Function that can be the Requester or the Completer of a PCI Express transaction either on its own behalf or on behalf of a distinct non-PCI Express device (other than a PCI device or Host CPU), e.g., a PCI Express attached graphics controller or a PCI Express-USB host controller. Endpoints are classified as either legacy, PCI Express, or Root Complex Integrated Endpoints. [29]

7.4 PCIe Layering Overview

The architecture of PCI Express is specified in terms of three discrete logical layers: the Transaction Layer, the Data Link Layer, and the Physical Layer. Each of these layers is divided into two sections: one that processes outbound (to be transmitted) information and one that processes inbound (received) information, as shown in Figure 7.3.

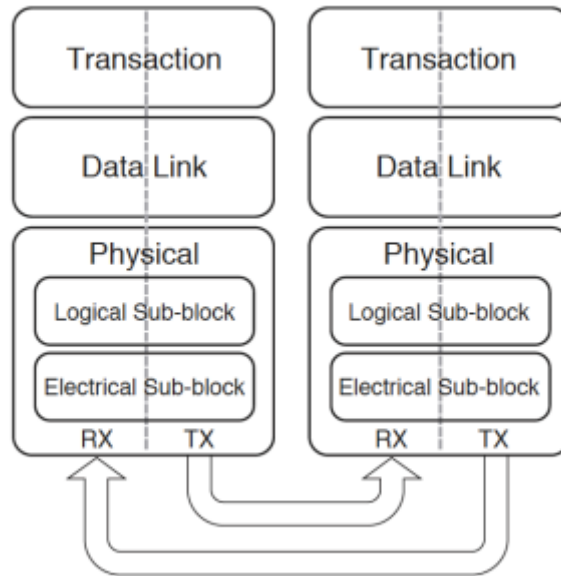


Figure 7-3 PCI Express Layering Diagram

PCI Express uses packets to communicate information between components. Packets are formed in the Transaction and Data Link Layers to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the other layers, they are extended with additional information necessary to handle packets at those layers. At the receiving side the reverse process occurs and packets get transformed from their Physical Layer representation to the Data Link Layer representation and finally (for Transaction Layer Packets) to the form that can be processed by the Transaction Layer of the receiving device, the procedure is similar to the encapsulation of packets in the network layers of the internet, such as the Transport layer (TCP), the Network layer (IP) and the Link layer.

PCI Express stack is composed of three layers.

1. The physical layer.
2. The data link layer.
3. The transaction layer.

The first two layers are the ones implemented for us in the PCI Express FPGA core (usually a combination of hard and soft core) and handling all the complexity. As a user, we work only in the transaction layer.

In more details:

1. The physical layer: that's where the pins are toggling. The 8b/10b encoding/decoding and the lanes disassembly/reassembly are done there.
2. The data link layer: that's where data integrity is checked (CRCs) and packets are re-transmitted if required.
3. The transaction layer: that's the user level. Once a packet arrives here, it is guaranteed to be good data.

7.4.1 Transaction layer

The upper layer of the architecture is the Transaction Layer. The Transaction Layer's primary responsibility is the assembly and disassembly of Transaction Layer Packets (TLPs). TLPs are the packets used to communicate transactions, such as read and write, as well as certain types of events. The Transaction Layer is also responsible for managing credit-based flow control for TLPs. Every request packet requiring a response packet is implemented as a split transaction. Each packet has a unique identifier that enables response packets to be directed to the correct originator. The packet format supports different forms of addressing depending on the type of the transaction (Memory, I/O, Configuration, and Message). The Packets may also have attributes such as No Snoop, Relaxed Ordering, and ID-Based Ordering (IDO). The Transaction Layer supports four address spaces: it includes the three PCI address spaces (**memory, I/O, and configuration**) and adds Message Space. [29] The Transaction Layer, in the process of generating and receiving TLPs, exchanges Flow Control information with its complementary Transaction Layer implementations on the other side of the link. It is also responsible for supporting both software and hardware-initiated power management. Initialization and configuration functions require the Transaction Layer to store the link's configuration that is generated by the processor, and the link capabilities generated by the physical layer hardware negotiation, such as width and operational frequency. A Transaction Layer's Packet generation and processing services require it to generate TLPs from device core requests and convert received requests TLPs into request for the specific device core.

7.4.2 Data link layer

The middle layer in the stack, the Data Link Layer, serves as an intermediate stage between the Transaction Layer and the Physical Layer. The primary responsibilities of the Data Link Layer include link management and data integrity, including error detection and error correction. The transmission side of the Data Link Layer accepts TLPs assembled by the Transaction Layer, calculates and applies a data protection code and TLP sequence number, and submits them to Physical Layer for transmission across the link. The receiving Data Link Layer is responsible for checking the integrity of received TLPs and for submitting them to the Transaction Layer for further processing. On detection of TLP errors, this layer is responsible for requesting re-transmission of TLPs until information is correctly received, or the link is considered to have failed. The Data Link Layer also generates and consumes packets that are used for Link management functions. To differentiate these packets from those used by the Transaction Layer (TLP), the term Data Link Layer Packet (DLLP) will be used when referring to packets that are generated and consumed at the Data Link Layer. Some of the services of the Data Link Layer regarding data protection, error checking and re-transmission are CRC generation, transmitted TLP storage for data link level retry, error checking, TLP acknowledgment are retry messages and error indication for error reporting and logging.

7.4.3 Physical layer

The Physical Layer includes all circuitry for interface operation, including driver and input buffers, parallel-to-serial and serial-to-parallel conversion, PLLs, and impedance matching circuitry. It includes also logical functions related to interface initialization and maintenance. The Physical Layer exchanges information with the Data Link Layer in an implementation specific format. This Layer is responsible for converting information received from the Data Link Layer into an appropriate serialized format and transmitting it across the PCI Express Link at a frequency and width compatible with the device connected to the other side of the Link. The PCI Express architecture has “hooks” to support future performance enhancements via speed upgrades and advanced encoding techniques. The future speeds, encoding techniques or media may only impact the Physical Layer definition. [29]

7.4.3.1 Logical Sub-block

Takes care of symbol encoding, framing, data scrambling, link initialization and training, lane to lane de-skew.

Electrical Sub-block

The electrical sub-block section defines the physical layer of the PCI Express that consists of reference clock, transmitter, receiver and channel. This section defines the electrical layer parameters required to guarantee the interoperability between the above listed PCI Express parameters.

7.5 Types of PCI Express Protocol

There are two types for implementation PCIe interface:

- **Soft**

In this type we design and implement all layer of PCIe protocol from Application layer until Physical layer using VHDL or Verilog language.

- **Hardened**

In this type we only design and implement the Application layer and the other layer already exist as hard IP block in the FPGA. [31]

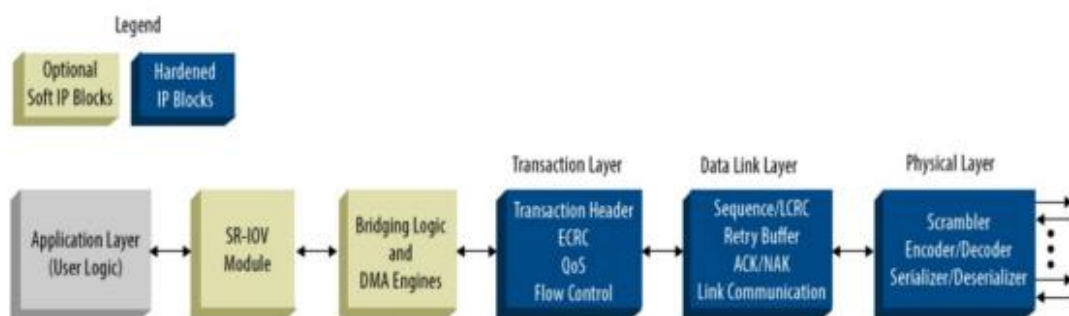


Figure 7-4 PCI Express soft and hardened implementation

7.6 Why PCI Express Interface?

PCIe is high-performance, scalable, and feature-rich serial protocol with data transfer rates starts from 2.5 GT/s to 8.0 GT/s and most of GPPs have native support for PCIe bus, so it is the best choice for this project where the main purpose is to offload the most complex unit (Turbo Decoder) to highly parallel platform (FPGA) to allow the implementation of L1 DSP computationally intensive processing on General Purpose Processor (GPP) based architectures. Since PCIe is supported by most GPPs and has high data transfer rate, it's the best choice for the project.

In our project, **The Spartan-6 FPGA SP605 Evaluation Kit** (shown in figure 7.5) from **Xilinx** is used to implement the turbo decoder and interface it with the workstation through PCIe interface [32]. Since the **Spartan-6 FPGA** has hard IP block for PCIe Integrated Endpoint, hardened implementation is used where only the application layer is developed and the PCI Express stack (Transaction layer, Data link layer, Physical layer) is generated as IP core using Xilinx Coregen.



Figure 7-5 The Spartan-6 FPGA SP605 Evaluation Kit [32]

The Spartan-6 FPGA Integrated Endpoint Block for PCI Express has the specifications shown in Figure 7.6 [33].

Product Name	1-lane Integrated Endpoint Block
FPGA Architecture	Spartan-6
User Interface Width	32
Lane Widths Supported	x1
Link Speeds Supported	2.5 GT/s
PCIe Base Specification Compliance	v1.1

Figure 7-6 Spartan-6 FPGA IP Core Specifications [33]

Chapter 8: PCI Express Linux Driver and RIFFA Framework

Device drivers take on a special role in the Linux kernel. They are distinct “blackboxes” that make a particular piece of hardware respond to a defined internal programming interface, additionally they hide completely the details of how the device works. User activities are performed by means of a set of standardized calls that are independent of the specific driver. Mapping those calls to device-specific operations that act on real hardware is the role of the device driver. This programming interface is such that drivers can be built separately from the rest of the kernel and then plugged in at runtime when needed. This modularity makes Linux drivers easy to write, to the point that there are now hundreds of them available. One of the features of the Linux operating system is the ability to extend at runtime the set of features offered by the kernel. This means that you can add functionality to the kernel (and remove functionality as well) while the system is up and running. Each piece of code that can be added to the kernel at runtime is called a module. The Linux kernel offers support for quite a few different types (or classes) of modules, including, but not limited to, device drivers. Each module is made up of object code that can be dynamically linked to the running kernel by the insmod program and can be unlinked by the rmmod program.

8.1 PCI Addressing

Each PCI peripheral is identified by a bus number, a device number, and a function number. The PCI specification permits a single system to host up to 256 buses, but because 256 buses are not sufficient for many large systems, Linux now supports PCI domains. Each PCI domain can host up to 256 buses. Each bus hosts up to 32 devices, and each device can be a multifunction board (such as an audio device with an accompanying CD-ROM drive) with a maximum of eight functions. [34]

During boot, the BIOS-type boot firmware (or the kernel itself if so configured) walks the PCI bus and assigns resources such as interrupt levels and I/O base addresses. The device driver gleans this assignment by peeking at a memory region called the PCI configuration space. PCI devices possess 256 bytes of configuration memory. The top

64 bytes of the configuration space is standardized and holds registers that contain details such as the status, interrupt line, and I/O base addresses. PCIe offers an extended configuration space of 4KB. [35]

PCI drivers register the vendor IDs, device IDs, and class codes that they support with the PCI subsystem from the configuration space as shown in Table 8-1 [35]

Table 8.1: PCI Configuration Space

Configuration Space Offset	Semantics	Values from the Dump Output for the Xircom Card
0	Vendor ID	0x115D
2	Device ID	0x0003
10	Class code	0x0200
16 to 39	Base address register 0 (BAR 0) to BAR5	0x3001...0000
44	Subvendor ID	0x115D
46	Subdevice ID	0x1181

Base Address Registers (BARs) serve two purposes. Initially they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, The Base Address registers are programmed with the addresses and the device uses this information to perform decoding.

8.2 Direct Memory Access

Direct memory access, or DMA, is the advanced topic that completes our overview of how to create a modern PCI driver. DMA is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor. Use of this mechanism can greatly increase throughput to and from a device, because a great deal of computational overhead is eliminated. [34]

Let's begin with the mechanism of how a DMA transfer takes place, considering only input transfers to simplify the discussion. Data transfer can be triggered in two ways: either the software asks for data (via a function such as read) or the hardware

asynchronously pushes data to the system. In the first case, the steps involved can be summarized as follows:

- 1- When a process calls read, the driver method allocates a DMA buffer and instructs the hardware to transfer its data into that buffer. The process is put to sleep.
- 2- The hardware writes data to the DMA buffer and raises an interrupt when it's done.
- 3- The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

The second case comes about when DMA is used asynchronously. This happens, for example, with data acquisition devices that go on pushing data even if nobody is reading them. In this case, the driver should maintain a buffer so that a subsequent read call will return all the accumulated data to user space. The steps involved in this kind of transfer are slightly different:

- 1- The hardware raises an interrupt to announce that new data has arrived.
- 2- The interrupt handler allocates a buffer and tells the hardware where to transfer its data.
- 3- The peripheral device writes the data to the buffer and raises another interrupt when it's done.
- 4- The handler dispatches the new data, wakes any relevant process, and takes care of housekeeping.

The processing steps in all of these cases emphasize that efficient DMA handling relies on interrupt reporting. While it is possible to implement DMA with a polling driver, it wouldn't make sense, because a polling driver would waste the performance benefits that DMA offers over the easier processor-driven I/O. As far as interrupts are concerned, PCI is easy to handle. By the time Linux boots, the computer's firmware has already assigned a unique interrupt number to the device, and the driver just needs to use it. The interrupt number is stored in configuration register 60 (PCI_INTERRUPT_LINE), which is one byte wide.

8.2.1 DMA mappings

A DMA mapping is a combination of allocating a DMA buffer and generating an address for that buffer that is accessible by the device. The PCI code distinguishes between two types of DMA mappings, depending on how long the DMA buffer is expected to stay around, Coherent and Streaming DMA mappings.

Coherent DMA mappings usually exist for the life of the driver. A coherent buffer must be simultaneously available to both the CPU and the peripheral. As a result, coherent mappings must live in cache-coherent memory. Coherent mappings can be expensive to set up and use.

Streaming mappings are usually set up for a single operation. The kernel developers recommend the use of streaming mappings over coherent mappings whenever possible, and there are two reasons for this recommendation. The first is that, on systems that support mapping registers, each DMA mapping uses one or more of them on the bus. Coherent mappings, which have a long lifetime, can monopolize these registers for a long time, even when they are not being used. The other reason is that, on some hardware, streaming mappings can be optimized in ways that are not available to coherent mappings. [34]

8.2.2 How a PCIe driver works

The BAR address space (mapped in memory or I/O space) is used for control registers. The driver allocates buffers in RAM. The addresses of these buffers are written in control registers. The device reads and writes from the buffer via DMA. All this is timed out and orchestrated via control registers and interrupts. [36]

8.3 RIFFA Framework

RIFFA (Reusable Integration Framework for FPGA Accelerators) is a simple framework for communicating data from a host CPU to a FPGA via a PCI Express bus. The framework requires a PCIe enabled workstation and a FPGA on a board with a PCIe connector. RIFFA supports Windows and Linux, Altera and Xilinx, with bindings for C/C++, Python, MATLAB and Java.

RIFFA provides high bandwidth, low latency communication and synchronization between FPGA devices and computers equipped with a PCIe connection. It provides

this via simple software APIs and a FIFO hardware interface. On the software side there are two main functions: data send and data receive. These functions are exposed via user library in C. The driver supports multiple FPGAs (up to 5) per system. Users can communicate with FPGA IP cores by writing only a few lines of code. On the hardware side, users access an interface with independent transmit and receive signals. The signals provide transaction handshaking and a first word fall through FIFO interface for reading/writing data to the host. No knowledge of bus addresses, buffer sizes, or PCIe packet formats is required. Simply send data on a FIFO interface and receive data on a FIFO interface. RIFFA does not rely on a PCIe Bridge and therefore is not subject to the limitations of a bridge implementation. Instead, RIFFA works directly with the PCIe Endpoint and can run fast enough to saturate the PCIe link.

RIFFA communicates data using direct memory access (DMA) transfers and interrupt signaling. This achieves high bandwidth over the PCIe link.

8.3.1 RIFFA 1.0

The initial version of RIFFA is based on a set of components provided by Xilinx. It relies on a PCIe Endpoint, a PCIe Bridge, and a DMA core available in Xilinx's Embedded Development Kit. It supports a single FPGA per host PC. [37]

8.3.2 RIFFA 2.1

RIFFA 2.1 is a complete rewrite of the original release. It supports most modern FPGA devices from Xilinx and Altera across PCIe Gen 1, Gen 2, and Gen 3. The original release only supports the Xilinx Virtex 5 family.

RIFFA 1.0 requires the use of a Xilinx PCIe Processor Local Bus (PLB) Bridge core. Xilinx has since moved away from PLB technology and deprecated this core. The PLB Bridge core limited the PCIe configuration to a Gen 1 \times 1 link. Additionally, the bridge core did not support overlapping PLB transactions. This did not have an effect on the upstream direction because upstream transactions are one way. Downstream transactions, however, must be sent by the core and serviced by the host PC's root complex. Not being able to overlap transactions on the PLB bus results in only one outstanding downstream PCIe transaction at a time. This limits the maximum throughput for upstream and downstream transfers to 181MB/s and 25MB/s,

respectively. The relatively low downstream bandwidth was a chief motivator for improving upon RIFFA 1.0.

RIFFA 1.0 made use of a simple DMA core that uses PLB addressing to transfer data. The hardware interface exposes a set of DMA request signals that must be managed by the user core in order to complete DMA transfers. RIFFA 2.1 exposes no bus addressing or DMA transfer signals in the interface. Data is read and written directly from and to FWFT FIFO interfaces on the hardware end. On the software end, data is read and written from and to byte arrays. The software and hardware interfaces have been significantly simplified since RIFFA 1.0.

On the host PC, contiguous user space memory is typically scattered across many noncontiguous pages in physical memory. This is an artifact of memory virtualization and makes transfer of user space data difficult. Earlier versions of RIFFA had a single packet DMA engine that required physically scattered user space data be copied between a physically contiguous block of memory when being read or written to. Though simpler to implement, this limits transfer bandwidth because of the time required for the CPU to copy data. RIFFA 2.1 supports a scatter gather DMA engine. The scatter gather approach allows data to be read or written to directly from/to the physical page locations without the need to copy data.

RIFFA 1.0 supports only a single FPGA per host PC with C/C++ bindings for Linux. Version 2.1 supports up to 5 FPGAs that can all be addressed simultaneously from different threads. Additionally, RIFFA 2.1 has bindings for C/C++, Java, Python, and Matlab for both Linux and Windows. Lastly, RIFFA 2.1 is capable of reaching 97% maximum achievable PCIe link utilization during transfers. RIFFA 1.0 is not able to exceed more than 77% in the upstream direction or more than 11% in the downstream direction.

8.3.3 Design

RIFFA is based on the concept of communication channels between software threads on the CPU and user cores on the FPGA. A channel is similar to a network socket in that it must first be opened, can be read and written, and then closed. However, unlike a network socket, reads and writes can happen simultaneously (if using two threads). Additionally, all writes must declare a length so the receiving side knows how much data to expect. Each channel is independent. RIFFA supports up to 12 channels per

FPGA. Up to 12 different user cores can be accessed directly by software threads on the CPU, simultaneously. Designs requiring more than 12 cores per FPGA can share channels. This increases the number of effective channels, but requires users to manually multiplex and de-multiplex access on a channel. Before a channel can be accessed, the FPGA must be opened. RIFFA supports multiple FPGAs per system (up to five). This limit is software configurable. Each FPGA is assigned an identifier on system start up. Once opened, all channels on that FPGA can be accessed without any further initialization. Data is read and written directly from and to the channel interface. On the FPGA side, this manifests as a First Word Fall Through (FWFT) style FIFO interface for each direction. On the software side, function calls support sending and receiving data with byte arrays.

Memory read/write requests and software interrupts are used to communicate between the workstation and FPGA. The FPGA exports a configuration space accessible from an operating system device driver. The device driver accesses this address space when prompted by user application function calls or when it receives an interrupt from the FPGA. This model supports low-latency communication in both directions. Only status and control values are sent using this model. Data transfer is accomplished with large payload PCIe transactions issued by the FPGA. The FPGA acts as a bus master scatter gather DMA engine for both upstream and downstream transfers. In this way, multiple FPGAs can operate simultaneously in the same workstation with minimal CPU system load.

The details of the PCIe protocol, device driver, DMA operation, and all hardware addressing are hidden from both the software and hardware. This means some level of flexibility is lost for users to configure custom behaviors. For example, users cannot set up custom PCIe Base Address Register (BAR) address spaces and map them directly to a user core. Nor can they implement quality of service policies for channels or PCIe transaction types. However, we feel any loss is more than offset by the ease of programming and design.

8.3.4 RIFFA Software

8.3.4.1 Software Architecture

On the host PC is a kernel device driver and a set of language bindings. The device driver is installed into the operating system and is loaded at system startup. It handles

registering all detected FPGAs configured with RIFFA cores. Once registered, a small memory buffer is preallocated from kernel memory. This buffer facilitates sending scatter gather data between the workstation and FPGA.

A user library provides language bindings for user applications to be able to call into the driver. The user library exposes the software interface described in Section 8.3.4.2, when an application makes a call into the user library, the thread enters the kernel driver and initiates a transfer. [38]

At runtime, a custom communication protocol is used between the kernel driver and the RX Engine. The protocol is encoded in PCIe payload data and address offset. The protocol consists of single word reads and writes to the FPGA BAR address space. The FPGA communicates with the kernel driver by firing a device interrupt. The driver reads an interrupt status word from the FPGA to identify the conditions of each channel. The conditions communicated include **start of a transfer**, **end of a transfer**, and **request for scatter gather elements**. The protocol is designed to be as lightweight as possible. For example, a write of three words is all that is needed to start a downstream transfer. Once a transfer starts, the only communication between the driver and RIFFA is to provide additional scatter gather elements or signal transfer completion. [38]

8.3.4.2 Software Interface

The interface on the software side is consisted by a few functions. Data transfers can be initiated by both sides, PC functions initiate downstream transfers and hardware cores initiate upstream transfers. The function of the RIFFA 2.1 software interface is listed in Table 8-2 (for the C/C++ API). [38]

Table 8.2: Functions of RIFFA API

Function	Arguments
fpga_list	fpga_info_list * list
fpga_open	int id
fpga_close	fpga_t * fpga
fpga_send	fpga_t * fpga, int chnl, void * data, int len, int destoff, int last, long long timeout
fpga_rcv	fpga_t * fpga, int chnl, void * data, int len, long long timeout
fpga_reset	fpga_t * fpga

There are four primary functions in the API: open, close, send, and receive. The API supports accessing individual FPGAs and individual channels on each FPGA. There is

also a function to list the RIFFA-capable FPGAs installed on the system. A reset function is provided that triggers the FPGA channel reset signal. The RIFFA 2.1 library and device driver provide useful messages about transfer events. The messages will print to the operating system's kernel log [37]. Here are the library functions provided by the driver to the user space applications using RIFFA:

- **int fpga_list(fpga_info_list * list);**

Populates the fpga_info_list pointer with all FPGAs registered in the system.

Returns 0 on success, a negative value on error.

- **fpga_t * fpga_open(int id);**

Initializes the FPGA specified by id. On success, returns a pointer to an fpga_t struct. On error, returns NULL. Each FPGA must be opened before any channel can be accessed. Once opened, any number of threads can use the fpga_t struct.

- **void fpga_close(fpga_t * fpga);**

Cleans up memory/resources for the FPGA specified by the fd descriptor.

- **int fpga_send(fpga_t * fpga, int chnl, void * data, int len, int destoff, int last, long long timeout);**

Sends len words (4 byte words) from data to FPGA channel chnl using the fpga_t struct. The FPGA channel will be sent len, destoff, and last. If last is 1 the channel should interpret the end of this send as the end of a transaction. If last is 0, the channel should wait for additional sends before the end of the transaction. If timeout is non-zero, this call will send data and wait up to timeout ms for the FPGA to respond (between packets) before timing out. If timeout is zero. Multiple threads sending on the same channel may result in corrupt data or error. On success, returns the number of words sent. On error returns a negative value.

- **int fpga_recv(fpga_t * fpga, int chnl, void * data, int len, long long timeout);**

Receives data from the FPGA channel `chnl` to the data pointer, using the `fpga_t` struct. The FPGA channel can send any amount of data, so the data array should be large enough to accommodate this data. The `len` parameter specifies the actual size of the data buffer in words (4 byte words). The FPGA channel will specify an offset which will determine where in the data array the data will start being written. If the amount of data (plus offset) exceed the size of the data array (`len`), then that data will be discarded. If `timeout` is non-zero, this call will wait up to `timeout` ms for the FPGA to respond (between packets) before timing out. If `timeout` is zero, this call may block indefinitely. Multiple threads receiving on the same channel may result in corrupt data or error. On success, it returns the number of words written to the data array. On error returns a negative value.

- **`void fpga_reset(fpga_t * fpga);`**

Resets the state of the FPGA and all transfers across all channels. This is meant to be used as an alternative to rebooting if an error occurs while sending/receiving.

NOTE: Calling this function while other threads are sending or receiving will result in unexpected behavior.

8.3.5 Hardware Interface

A single RIFFA channel has two sets of signals, one for receiving data (RX) and one for sending data (TX). RIFFA has simplified the interface to use a minimal handshake and receive/send data using a FIFO with first word fall through semantics (valid+read interface). The clocks used for receiving and sending can be asynchronous from each other and from the PCIe interface (RIFFA clock). The tables 8-3, 8-4 below describes the ports of the interface. The input/output designations are from your our core's perspective. The interface is pretty similar to the AXI-4 interface provided by the Xilinx core, but it has added functionality for the current use case.

Table 8.3: Hardware Interface Receive Ports

Name	Description
CHNL_RX_CLK	Provide the clock signal to read data from the incoming FIFO.
CHNL_RX	Goes high to signal incoming data. Will remain high until all incoming data is written to the FIFO.
CHNL_RX_ACK	Must be pulsed high for at least 1 cycle to acknowledge the incoming data transaction.
CHNL_RX_LAST	High indicates this is the last receive transaction in a sequence.
CHNL_RX_LEN[31:0]	Length of receive transaction in 4 byte words.
CHNL_RX_OFF[30:0]	Offset in 4 byte words indicating where to start storing received data (if applicable in design).
CHNL_RX_DATA[DWIDTH-1:0]	Receive data.
CHNL_RX_DATA_VALID	High if the data on CHNL_RX_DATA is valid.
CHNL_RX_DATA_REN	When high and CHNL_RX_DATA_VALID is high, consumes the data currently available on CHNL_RX_DATA.

Table 8.4: Hardware Interface Transmit Ports

Name	Description
CHNL_TX_CLK	Provide the clock signal to write data to the outgoing FIFO.
CHNL_TX	Set high to signal a transaction. Keep high until all outgoing data is written to the FIFO.
CHNL_TX_ACK	Will be pulsed high for at least 1 cycle to acknowledge the transaction.
CHNL_TX_LAST	High indicates this is the last send transaction in a sequence.
CHNL_TX_LEN[31:0]	Length of send transaction in 4 byte words.
CHNL_TX_OFF[30:0]	Offset in 4 byte words indicating where to start storing sent data in the PC thread's receive buffer.
CHNL_TX_DATA[DWIDTH-1:0]	Send data.
CHNL_TX_DATA_VALID	Set high when the data on CHNL_TX_DATA valid. Update when CHNL_TX_DATA is consumed.
CHNL_TX_DATA_REN	When high and CHNL_TX_DATA_VALID is high, consumes the data currently available on CHNL_TX_DATA.

The timing diagram in figure 8-1 shows the RIFFA channel receiving a data transfer of 16 (4byte) words (64 bytes). When CHNL_RX is high, CHNL_RX_LAST, CHNL_RX_LEN, and CHNL_RX_OFF will all be valid. In this example, CHNL_RX_LAST is high, indicating to the user core that there are no other transactions following this one and that the user core can start processing the received data as soon as the transaction completes. CHNL_RX_LAST may be set low if multiple transactions will be initiated before the user core should start processing received data. Of course, the user core will always need to read the data as it arrives, even if CHNL_RX_LAST is low.

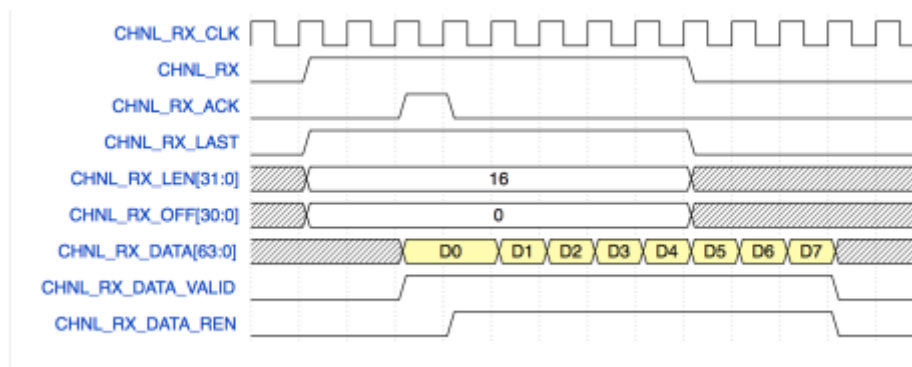


Figure 8-1:Receive Timing Diagram

In the example CHNL_RX_OFF is 0. However, if the PC specified a value for offset when it initiated the send, that value would be present on the CHNL_RX_OFF signal. The 31 least significant bits of the 32 bit integer specified by the PC thread are

transmitted. The CHNL_RX_OFF signal is meant to be used in situations where data is transferred in multiple sends and the user core needs to know where to write the data (if, for example it is writing to BRAM or DRAM).

The user core must pulse the CHNL_RX_ACK signal high for at least one cycle to acknowledge the receive transaction. The RIFFA channel will not recognize that the transaction has been received until it receives a CHNL_RX_ACK pulse. The combination of CHNL_RX_DATA_VALID high and CHNL_RX_DATA_REN high consumes the data on CHNL_RX_DATA. New data will be provided until the FIFO is drained. Note that the FIFO may drain completely before all the data has been received. The CHNL_RX signal will remain high until all data for the transaction has been received into the FIFO. Note that CHNL_RX may go low while CHNL_RX_DATA_VALID is still high. That means there is still data in the FIFO to be read by the user core. Attempting to read (asserting CHNL_RX_DATA_REN high) while CHNL_RX_DATA_VALID is low, will have no affect on the FIFO. The user core may want to count the number of words received and compare against the value provided by CHNL_RX_LEN to keep track of how much data is expected. [39]

The diagram in figure 8-2 shows the RIFFA channel sending a data transfer of 16 (4 byte) words (64bytes). It's nearly symmetric to the receive example. The user core sets CHNL_TX high and asserts values for CHNL_TX_LAST, CHNL_TX_LEN, and CHNL_TX_OFF for the duration CHNL_TX is high. CHNL_TX must remain high until all data has been consumed. RIFFA will expect to read CHNL_TX_LEN words from the user core. Any more data provided may be consumed, but will be discarded. The user core can provide less than CHNL_TX_LEN words and drop CHNL_TX at any point. Dropping CHNL_TX indicates the end of the transaction. Whatever data was consumed before CHNL_TX was dropped will be sent and reported as received to the software thread. [39]

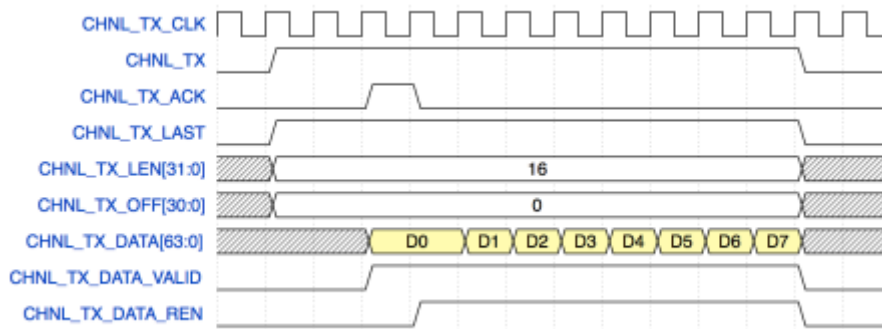


Figure 8-2: Transmit Timing Diagram.

As with the receive interface, setting CHNL_TX_LAST high will signal to the PC thread to not wait for additional transactions (after this one). Setting CHNL_TX_OFF will cause the transferred data to be written into the PC thread's buffer starting CHNL_TX_OFF 4 bytes words from the beginning. This can be useful when sending multiple transactions and needing to order them in the PC thread's receive buffer. CHNL_TX_LEN defines the length of the transaction in 4 byte words. As the CHNL_TX_DATA bus can be 32 bits, 64 bits, or 128 bits wide, it may be that the number of 32 bit words the user core wants to transfer is not an even multiple of the bus width. In this case, CHNL_TX_DATA_VALID must be high on the last cycle CHNL_TX_DATA has at least 1 word to send. The channel will only send as many words as is specified by CHNL_TX_LEN. So any additional data consumed, past the last word, will be discarded.

Shortly after CHNL_TX goes high, the RIFFA channel will pulse high the CHNL_TX_ACK and begin to consume the CHNL_TX_DATA bus. The combination of CHNL_TX_DATA_VALID high and CHNL_TX_DATA_REN high will consume the data currently on CHNL_TX_DATA. New data can be consumed every cycle. After all the data is consumed, CHNL_TX can be dropped. Keeping CHNL_TX_DATA_VALID high while CHNL_TX_DATA_REN is low will have no effect.

8.3.6 Architecture

On the FPGA, the RIFFA architecture is a scatter gather bus master DMA design connected to a vendor-specific PCIe Endpoint core, could be the core from Xilinx that we are using or an equivalent core from Altera. The PCIe Endpoint core drives the gigabit transceivers and exposes a bus interface for PCIe formatted packet data.

RIFFA cores use this interface to translate between payload data and PCIe packets. A set of RIFFA channels provide read and write asynchronous FIFOs to user cores that deal exclusively with payload data.

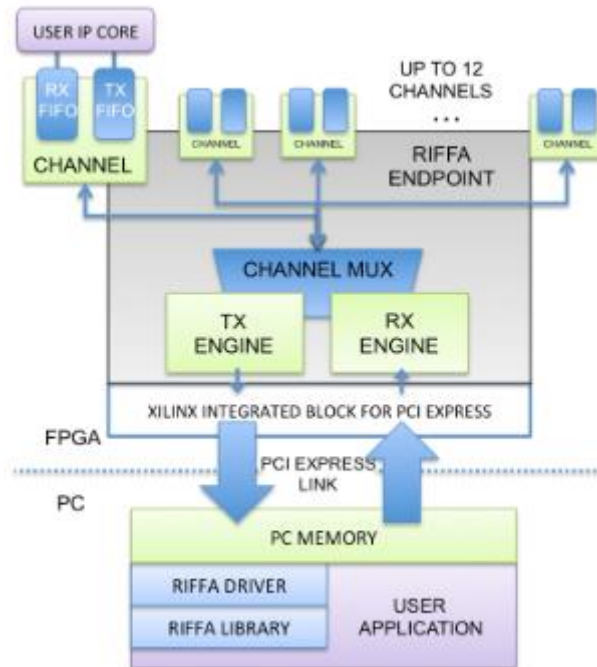


Figure 8-3: RIFFA Hardware Architecture.

The RIFFA cores are driven by a clock derived from the PCIe reference clock. This clock's frequency is a product of the PCIe link configuration. It runs fast enough to saturate the PCIe link if data were sent every cycle. User cores do not need to use this clock for their CHNL_TX_CLK or CHNL_RX_CLK. Any clock can be used by the user core. The frequency of the clock is determined by the Xilinx core and is chosen by the configuration of the PCIe Link we are going to support, for example if we create an endpoint for a **1 lane** setup we need to use **62.5 MHz clock**.

The PCIe link configuration also determines the width of the PCIe data bus. This width can be 32, 64, or 128 bits wide. Writing a DMA engine that supports multiple widths requires different logic when extracting and formatting PCIe data. For example, with a 32-bit interface, header packets can be generated one 4-byte word per cycle. Only one word can be sent/received per cycle. Therefore, the DMA engine only needs to process one word at a time, containing either header or payload data. However, with a 128-bit interface, a single cycle presents four words per cycle. This may require processing three header packets and the first word of payload in a single

cycle. It is possible (and simpler) to design a scatter gather DMA engine that does not perform such advanced and flexible processing. However, the result is a much lower performing system that does not take advantage of the underlying link as efficiently. There are many examples of this in research and industry. [38]

Upstream transfers are initiated by the user core via the CHNL_TX_* ports. Data written to the TX FIFO is split into chunks appropriate for individual PCIe write packets. RIFFA will attempt to send the maximum payload per packet. It must also avoid writes that cross physical memory page boundaries, as this is prohibited by the PCIe specification. In order to send the data, the locations in host PC memory need to be retrieved. This comes in the form of scatter gather elements. Each scatter gather element defines a physical memory address and size. These define the memory locations into which the payload data will be written. Therefore, each channel first requests a read of list of scatter gather elements from the host. Once the channel has the scatter gather elements, they issue write packets for each chunk of data. Channels operate independently and share the upstream PCIe direction. The TX Engine provides this multiplexing. [38]

The TX Engine drives the upstream direction of the vendor-specific PCIe Endpoint interface. It multiplexes access to this interface across all channels. Channel requests are serviced in a round-robin fashion. The TX Engine also formats the requests into full PCIe packets and sends them to the vendor-specific PCIe Endpoint. The TX Engine is fully pipelined and can write a new packet every cycle. Throttling on data writes only occurs if the vendor specific PCIe Endpoint core cannot transmit the data quickly enough. The Endpoint may apply back pressure if it runs out of transmit buffers. As this is a function of the host PC's root complex acknowledgment scheme, it is entirely system dependent. [38]

Downstream transfers are initiated by the host PC via the software APIs and manifest on the CHNL_RX_* ports. Once initiated, the channel cores request scatter gather elements for the data to transfer. Afterward, individual PCIe read requests are made for the data at the scatter gather element locations. Care is also taken to request data so as to not overflow the RX FIFO. Each channel throttles the read request rate to match the rate at which the RX FIFO is draining. Channel requests are serviced by the TX Engine. When the requested data arrives at the vendor Endpoint, it is forwarded to

the RX Engine. There the completion packet data is reordered to match the requested order. Payload data is then provided to the channel.

The RX Engine core is connected to the downstream ports on the vendor-specific PCIe Endpoint. It is responsible for extracting data from received PCIe completions and servicing various RIFFA device driver requests. It also demultiplexes the received data to the correct channel. The RX Engine processes incoming packets at line rate. It therefore never blocks the vendor-specific PCIe Endpoint core. Data received by the Endpoint will be processed as soon as it is presented to the RX Engine, avoiding the possibility of running out of buffer space. After extracting payload data, the RX Engine uses a Reordering Queue module to ensure the data is forwarded to the channel in the order it was requested. [38]

8.3.6.1 Upstream transfers

A sequence diagram for an upstream transfer is shown in Figure 8-4. An upstream transfer is initiated by the FPGA. However, data cannot begin transferring until the user application calls the user library function `fpga_rcv`. Upon doing so, the thread enters the kernel driver and begins the pending upstream request. If the upstream request has not yet been received, the thread waits for it to arrive. The user can set a timeout parameter upon calling the `fpga_rcv` function. On the diagram, the user library and device driver are represented by the single node labeled “RIFFA Library.

Servicing the request involves building a list of scatter gather elements that identify the pages of physical memory corresponding to the user space byte array. The scatter gather elements are written to a small shared buffer. This buffer location and content length are provided to the FPGA so that it can read the contents. Each page enumerated by the scatter gather list is pinned to memory to avoid costly disk paging. The FPGA reads the scatter gather data, then issues write requests to memory for the upstream data. If more scatter gather elements are needed, the FPGA will request additional elements via an interrupt. Otherwise, the kernel driver waits until all the data is written. The FPGA provides this notification, again via an interrupt. [38]

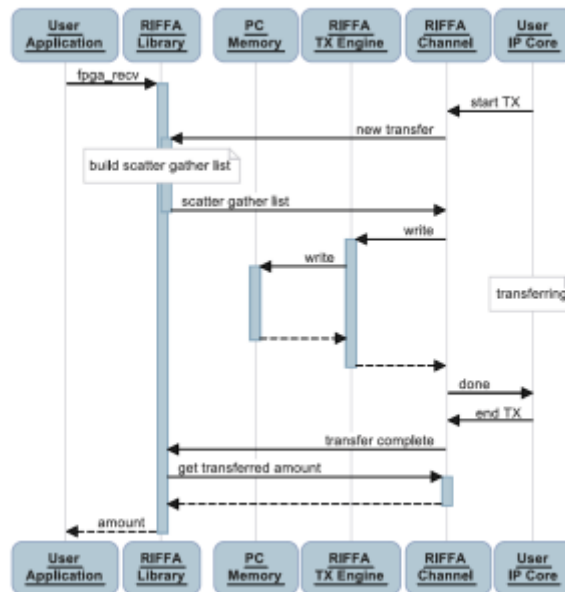


Figure 8-4: Upstream Data Transfer

After the upstream transaction is complete, the driver reads the FPGA for a final count of data words written. This is necessary as the scatter gather elements only provide an upper bound on the amount of data that is to be written. This completes the transfer and the function call returns to the application with the final count. [38]

8.3.6.2 Downstream transfers

A similar sequence exists for downstream transfers. Figure 8-5 illustrates this sequence. In this direction, the application initiates the transfer by calling the library function `fpga_send`. The thread enters the kernel driver and writes to the FPGA to initiate the transfer. Again, a scatter gather list is compiled, pages are pinned, and the FPGA reads the scatter gather elements. The elements provide location and length information for FPGA issued read requests. The read requests are serviced and the kernel driver is notified only when more scatter gather elements are needed or when the transfer has completed. [10]

Upon completion, the driver reads the final count read by the FPGA. In error-free operation, this value should always be the length of all the scatter gather elements. This count is returned to the user application. The kernel driver is thread safe and supports multiple threads in multiple transactions simultaneously. For a single channel, an upstream and downstream transaction can be active simultaneously, driven by two different threads. But multiple threads cannot simultaneously attempt a

transaction in the same direction. The data transfer will likely fail as both threads attempt to service each other's transfer events.

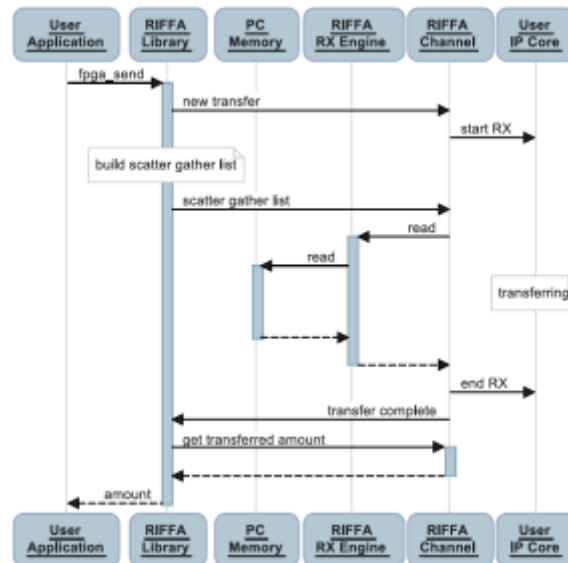


Figure 8-4: Downstream Data Transfer.

8.3.7 RIFFA 2.1 FPGA Support

RIFFA 2.1 relies on a Vendor PCIe Endpoint core to drive the transceivers. These are lowest-level interface that FPGA vendors provide. RIFFA 2.1 is tested with the following Xilinx and Altera Endpoint cores:

- Xilinx Spartan 6 Integrated Block for PCI Express ver. 2.4
- Xilinx Virtex 6 Integrated Block for PCI Express ver. 2.5
- Xilinx 7 Series Integrated Block for PCI Express vers. 1.6, 1.8, 2.1
- Altera IP Compiler for PCI Express (Stratix IV, Cyclone IV)
- Altera HardIP For PCI Express (Stratix V)

8.3.8 RIFFA 2.1 Bandwidth

RIFFA 2.1 is significantly more efficient than its predecessors. The RIFFA 2.1 is able to saturate the PCIe link for nearly all link configurations supported. Figure 8-5 shows the performance of designs using the 32 bit, 64 bit, and 128 bit interfaces. The colored bands show the bandwidth region between the theoretical maximum and the maximum achievable. PCIe Gen 1 and 2 use 8 bit / 10 bit encoding which limits the maximum achievable bandwidth to 80% of the theoretical. Our experiments show that

RIFFA can achieve 80% of the theoretical bandwidth in nearly all cases. The 128 bit interface achieves 76% of the theoretical maximum.

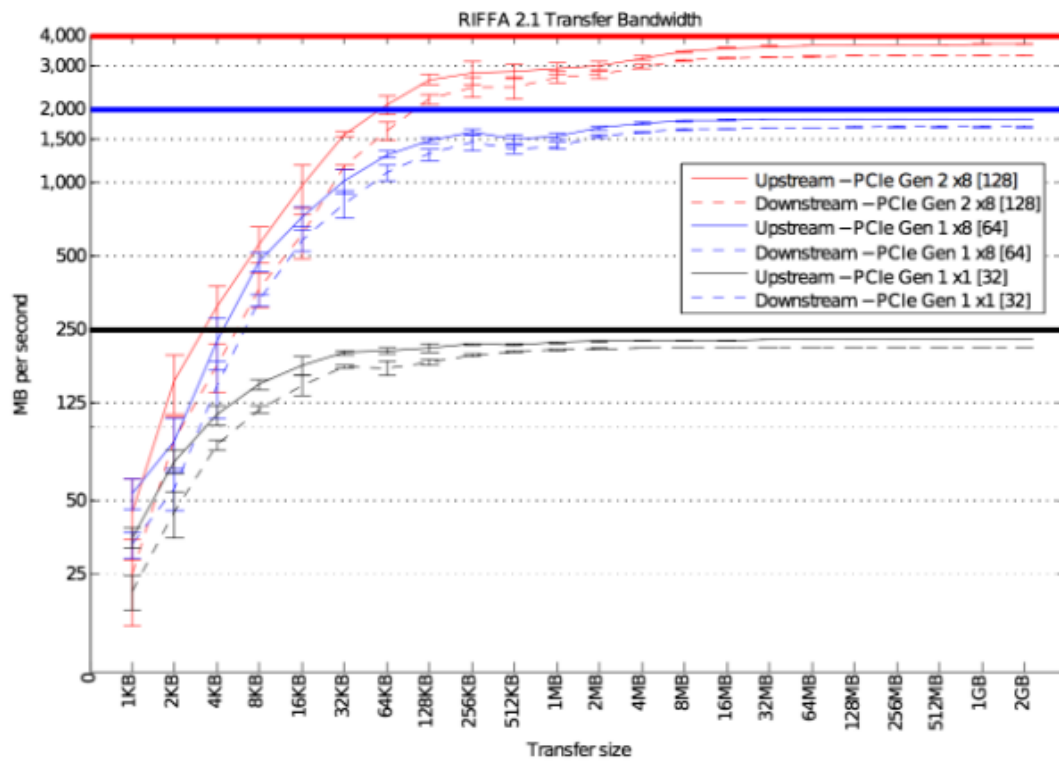


Figure 8-5: RIFFA bandwidth.

In our project, we used Xilinx Coregen to generate the PCIe Endpoint core (Xilinx Spartan 6 Integrated Block for PCI Express ver. 2.4) for **Spartan-6 (LX45T) FPGA SP605 Evaluation Kit** and Combine the PCIe Endpoint core’s source HDL with the RIFFA 2.1 HDL. More details about generating the PCIe Endpoint core and combining with RIFFA 2.1 HDL are illustrated in Chapter 9.

As illustrated in this chapter that RIFFA framework can support up to 12 channels per FPGA, where each channel communicates to a thread in the CPU. In our design 2 channels are used to interface the TURBO Decoder with 2 threads running on the CPU. More details about the interfacing are shown in Chapter 9.

Chapter 9: Turbo Interfacing

In this chapter, the process of implementing the Turbo decoder HDL (developed in chapter 6) on **Spartan-6 FPGA SP605 Evaluation Kit** and interfacing it through

PCIe link with workstation that has Linux kernels 2.6.27+ (versions between 2.6.32 - 3.X) is illustrated in the next sections.

9.1 Hardware Interface

9.1.1 PCIe Endpoint Core

The PCIe Endpoint core for Spartan 6 FPGAs is the **Spartan 6 Integrated Block for PCI Express**. This core is licensed by the Xilinx End User License Agreement and is provided with the Xilinx ISE Design suite with no additional charge. In this section, steps for generating the PCIe Endpoint core and then merging it with the RIFFA 2.1 source HDL are:

1. Using Xilinx CORE Generator to generate the PCIe Endpoint core.
2. Combining the PCIe Endpoint core's source HDL with the RIFFA 2.1 HDL.

Detailed instructions on how to do each step follow.

1. Using Xilinx CORE Generator to generate the PCIe Endpoint core

In this step CORE Generator is used to generate Verilog source for the Spartan 6 Integrated Block for PCI Express ver.2.4. Unless otherwise described, the default values on each wizard screen should be left as they are presented.

- Open the CORE Generator

Start → All Programs → Xilinx ISE Design Suite 14.7 →

ISE Design Tools → Tools → CORE Generator

- Create a new project; select File → New Project

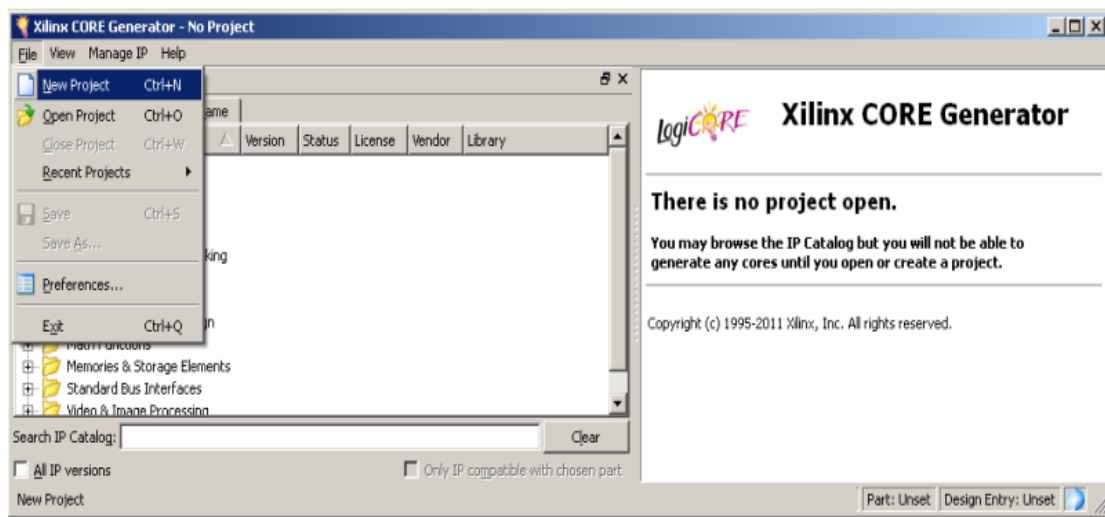


Figure 9-1: Xilinx CORE Generator.

- The Project options will appear as shown in Figure 9-2, set the project settings to generate Verilog code for the **XC6SLX45t-3FGG484** as follows:
 - Family: Spartan6
 - Device: xc6slx45t
 - Package: fgg484
 - Speed Grade: -3

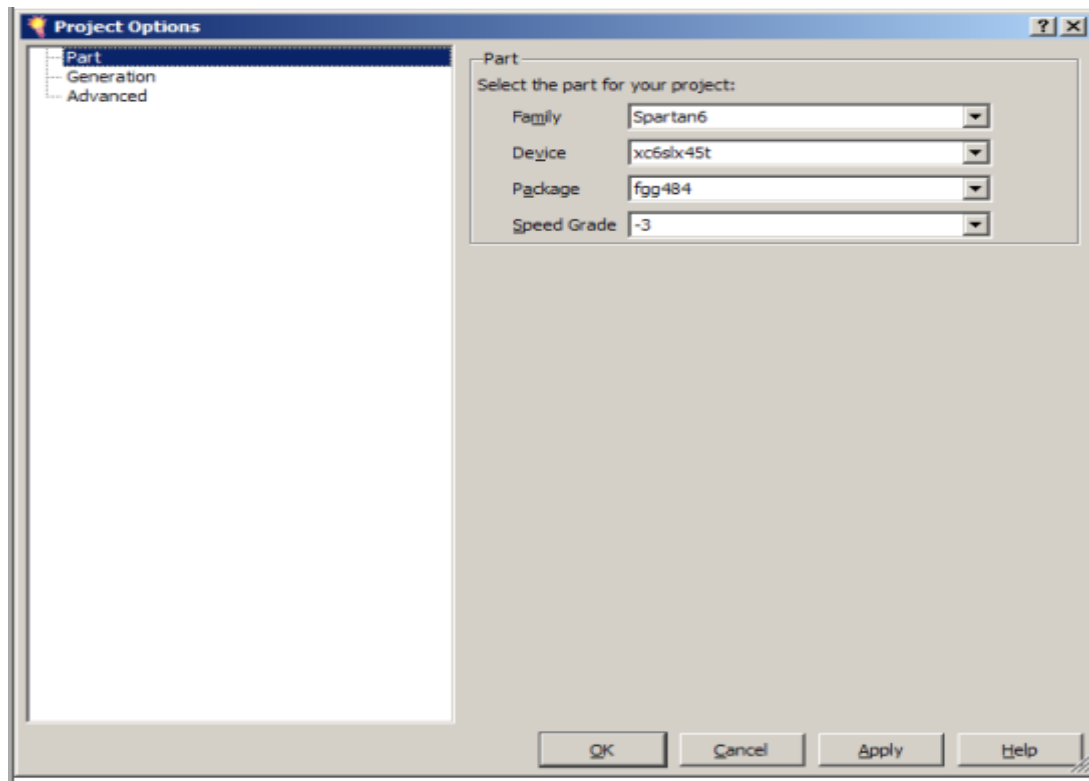


Figure 9-2: Xilinx CORE Generator Project Options.

- Select Generation : Set the Design Entry to Verilog

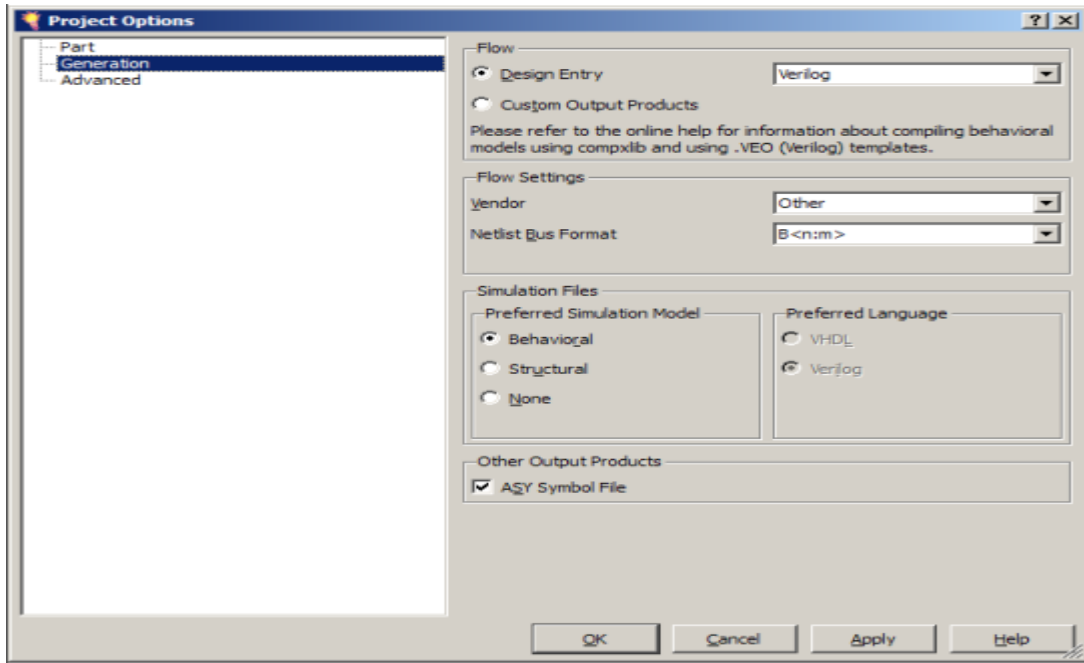


Figure 9-3: Xilinx CORE Generator Design Entry.

- Right click on the Spartan-6 Integrated Block for PCI Express, Version 2.4: Select Customize and Generate as shown in Figure 9-4.

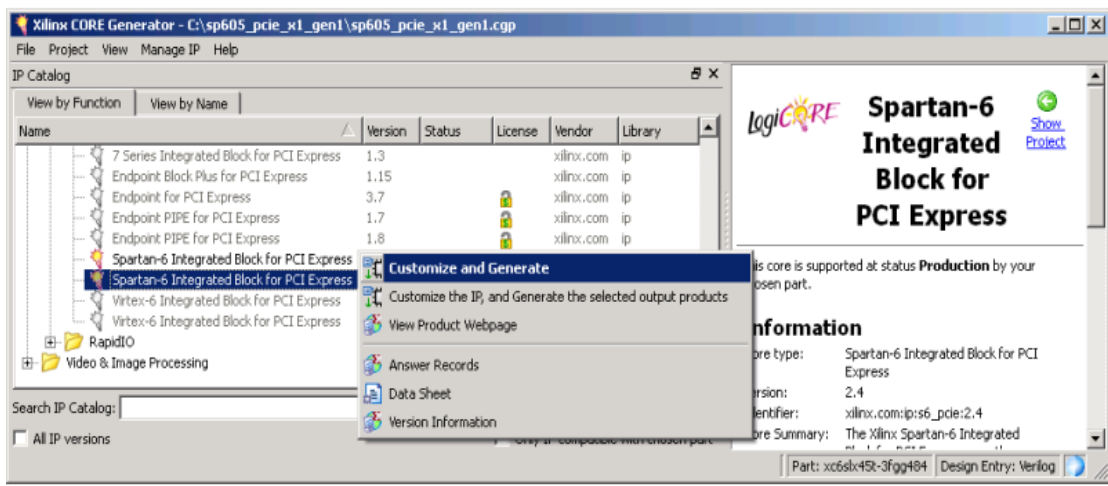


Figure 9-4: Xilinx CORE Generator generating IP CORE.

- As shown in Figure 9-5, there is no selection to make for lane width or link speed as this core only supports one lane at **2.5 GT/s**. This results in a **32 bit interface** and a **62.5 MHz interface frequency clock**.

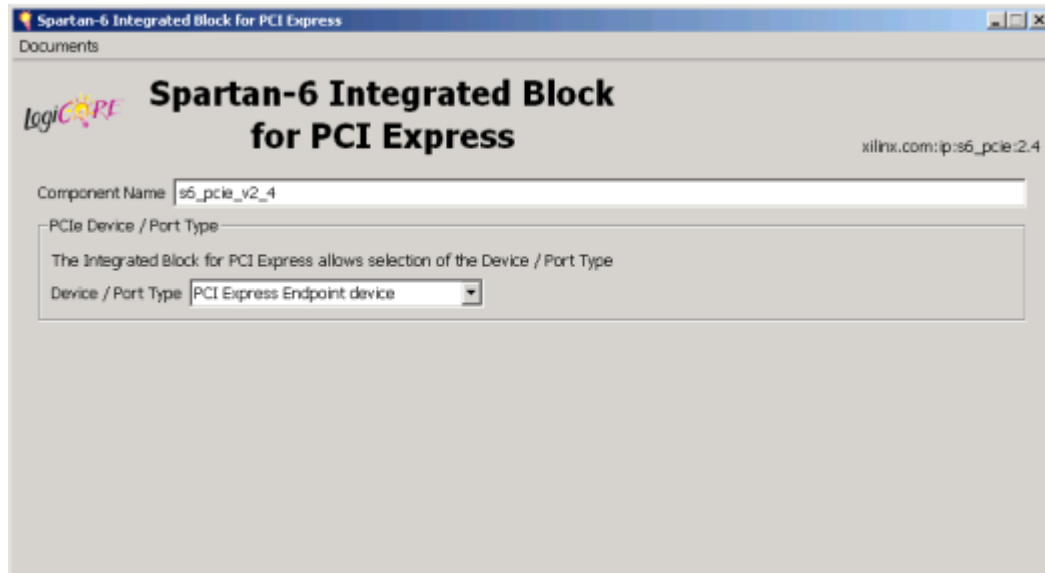


Figure 9-5: Spartan-6 Integrated Block for PCI Express.

- Select only Bar0 and set to a size of 1 KB. Deselect Bar2 as shown in Figure 9-6.

Spartan-6 Integrated Block for PCI Express xilinx.com:ip:s6_pcie:2.4

Base Address Registers

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses to assign to the device, the Base Address Registers are programmed with addresses and the device uses this information to perform address decoding.

BAR 0 Options

Bar0 Type: Memory 64 bit Prefetchable
 Size: 1 Kilobytes
 Value: FFFFFFFC00 (Hex)

BAR 1 Options

Bar1 Type: N/A 64 bit Prefetchable
 Size: 1 Bytes
 Value: 00000000 (Hex)

BAR 2 Options

Bar2 Type: N/A 64 bit Prefetchable
 Size: 128 Bytes
 Value: 00000000 (Hex)

BAR 3 Options

Bar3 Type: N/A 64 bit Prefetchable
 Size: 1 Bytes
 Value: 00000000 (Hex)

BAR 4 Options

Bar4 Type: N/A 64 bit Prefetchable
 Size: 1 Bytes
 Value: 00000000 (Hex)

BAR 5 Options

Bar5 Type: N/A Prefetchable
 Size: 1 Kilobytes
 Value: 00000000 (Hex)

Expansion ROM Base Address Register

Expansion Rom Size: 2 Kilobytes
 Value: 00000000 (Hex)

[Datasheet](#) < Back Page 2 of 9 Next > Generate Cancel Help

Figure 9-6: PCI Express CORE Base Address Registers.

- As shown in Figure 9-7, ID Initial Values are:
 - Vendor ID = **10EE**
 - Device ID = **0007**
 - Revision ID = **00**
 - Subsystem vendor ID = **10EE**
 - Subsystem ID = **0007**

Spartan-6 Integrated Block for PCI Express

Documents

Spartan-6 Integrated Block for PCI Express

xilinx.com:ip:s6_pcie:2.4

- ID Initial Values

Vendor ID	10EE	Range: 0000..FFFF
Device ID	0007	Range: 0000..FFFF
Revision ID	00	Range: 00..FF
Subsystem Vendor ID	10EE	Range: 0000..FFFF
Subsystem ID	0007	Range: 0000..FFFF

- Class Code

Base Class	05	Range: 00..FF
Sub-Class	00	Range: 00..FF
Interface	00	Range: 00..FF
Class Code	050000	(Hex)

- Class Code Lookup Assistant

Base Class	Simple communication controllers
Base Class	07h
Sub-Class/Interface Value	Generic XT compatible serial controller
Sub-Class	00h
Interface	00h

- Cardbus CIS Pointer

Cardbus CIS Pointer	00000000	Range: 00000000..FFFFFFFF
---------------------	----------	---------------------------

Datasheet < Back Page 3 of 9 Next > Generate Cancel Help

Figure 9-7: PCI Express CORE ID Initial Values.

- Select Performance Level High. Additionally, set the Max Payload Size to the maximum value offered as shown in Figure 9-8. These changes are not necessary for RIFFA 2.1 to function. They are required to achieve maximum performance.

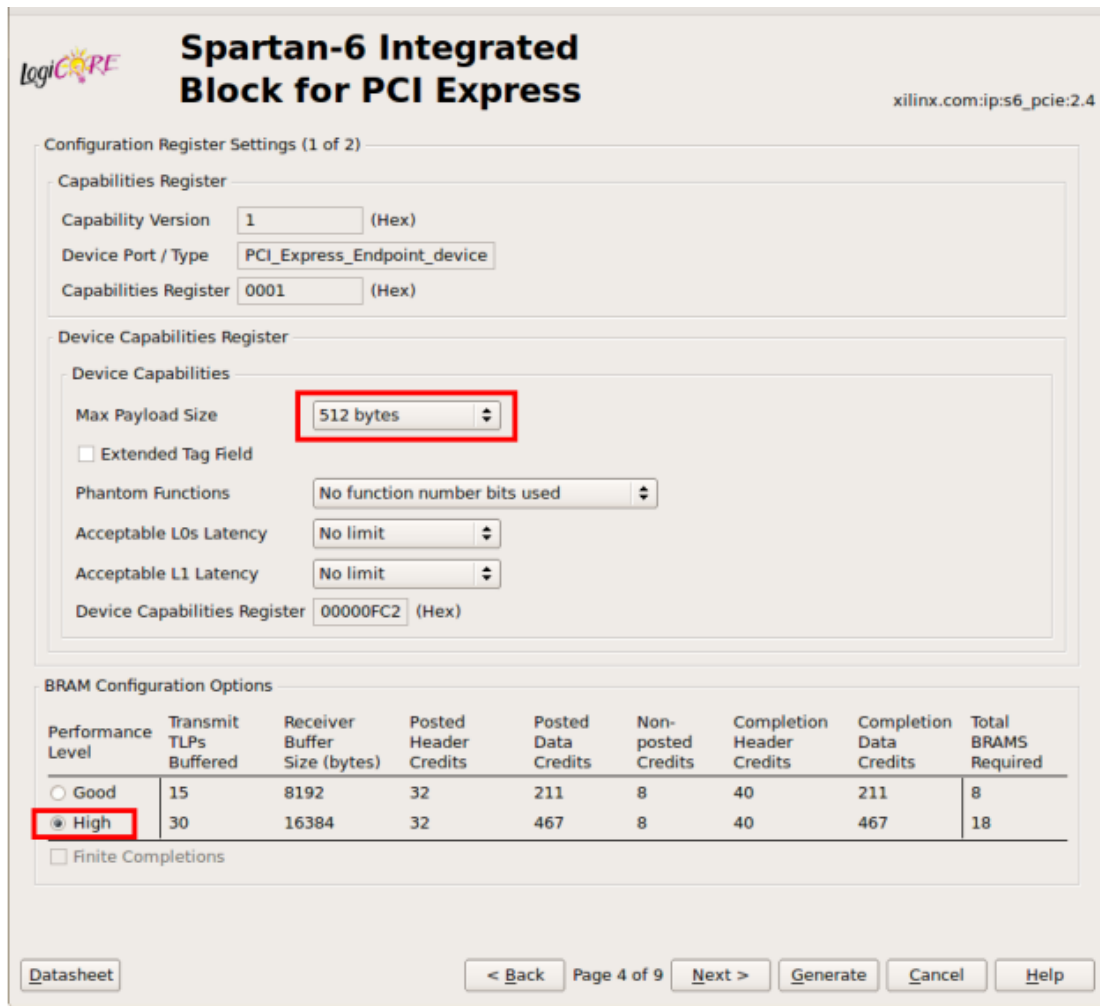


Figure 9-8: PCI Express CORE Max Payload Size.

- Select SP605 as shown in Figure 9-9.

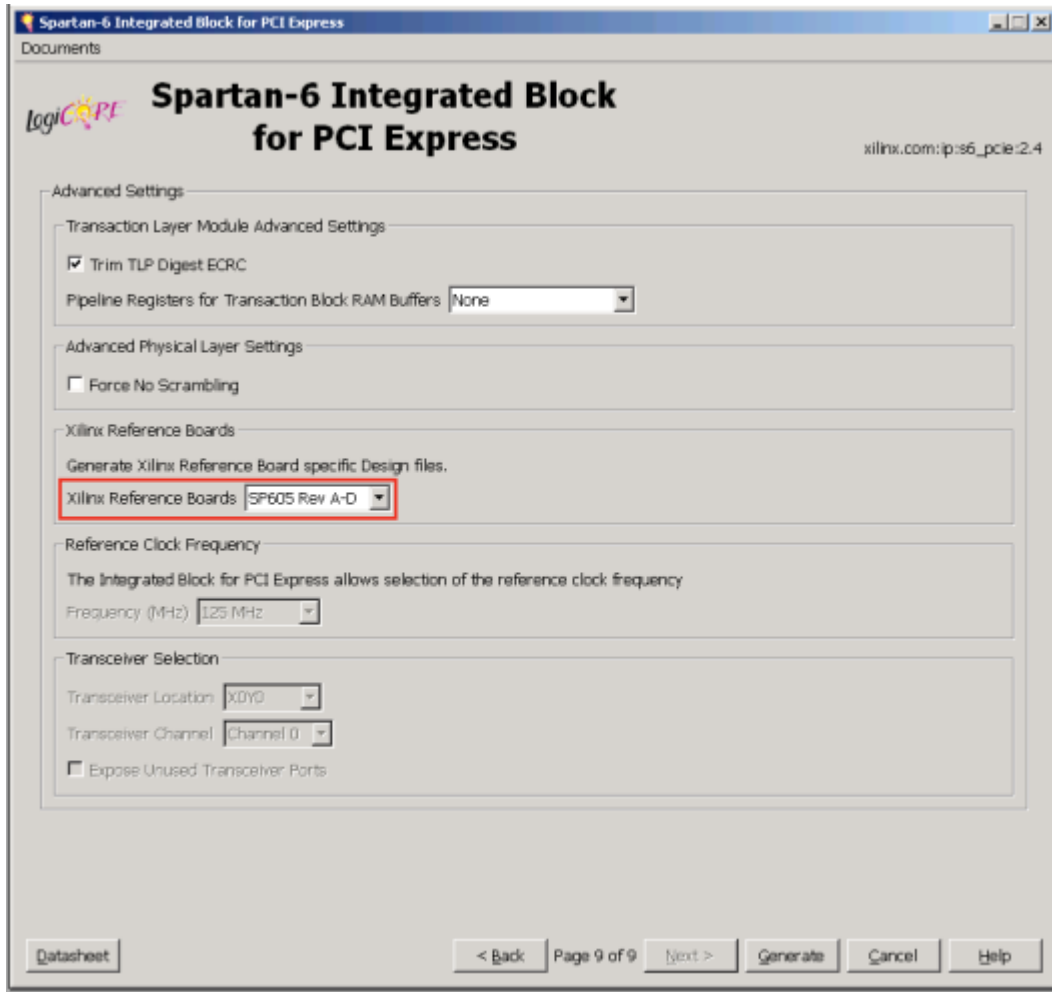


Figure 9-9: PCI Express Xilinx Reference Boards.

2. Combining the PCIe Endpoint core's source HDL with the RIFFA 2.1 HDL.

CORE Generator will produce a directory structure similar to what is shown in Figure 9-10. Once completed, combine all the source HDL files from the **source** directory with the RIFFA 2.1 HDL files from the distribution [39] into a new directory of your choosing. Also, into this new directory, copy the top level and adapter module HDL files for this board from the RIFFA distribution.

 doc	2011/02/24 16:00	File folder	
 example_design	2011/02/24 16:00	File folder	
 implement	2011/02/24 16:00	File folder	
 simulation	2011/02/24 16:00	File folder	
 source	2011/02/24 16:00	File folder	
 s6_pcie_v2_4_readme.txt	2012/10/12 16:00	Text Document	6 KB

Figure 9-10: PCIe IP CORE Directory.

9.1.2 Turbo Timing Constraints

Post Placing and routing the Turbo decoder design on **Spartan-6 XC6SLX45T FPGA**, the **critical path** of the design is **27.045ns**, Figure 9-11 shows the period of the critical path from the Post-PAR Static Timing Report generated by ISE Design suite.

```

-----
Total                               27.045ns (12.051ns logic, 14.994ns route)
                                       (44.6% logic, 55.4% route)
-----

```

Figure 9-11: Turbo decoder critical path delay

PERIOD Timing constraint is applied in the ucf (user constraints file) with period larger than the critical path delay, where **PERIOD = 27.5 ns (36.36 MHz)**. The design met the constraint with **Slack (setup path) = 0.172 ns** as shown in Figure 9-12.

Slack (setup path):0.172ns (requirement - (data path - clock path skew + uncertainty))			
Source: app/Turbo/TURBO_CU/TIMER/count_4 (FF)		clk: app/Turbo/CLK_BUF0 rising at 0.000ns	
Destination: app/Turbo/TURBO_DECODER_TD/SOFT_DELTA_MEMORY_ID_1/Mram_ram_memory8 (RAM)		clk: app/Turbo/CLK_BUF0 rising at 27.500ns	
Requirement	Data Path Delay	Clock Path Skew:	Clock Uncertainty
27.500ns	27.045ns (Levels of Logic = 18)	0.004ns (0.504 - 0.500)	0.287ns

Figure 9-12: Timing Constraints met

The Clock that drives the turbo design (**TURBO CLOCK**) = **36 MHz**. The SP605 has differential 200 MHz oscillator [40], so DCM IP Core is generated and configured using Xilinx Clocking Wizard to get the required clock. Figure 9-13 shows configuring the input clock and configuring Clocking Features:

- Input Clock = 200 MHz
- Input Jitter = 7.44 ps [41]
- Minimize output jitter

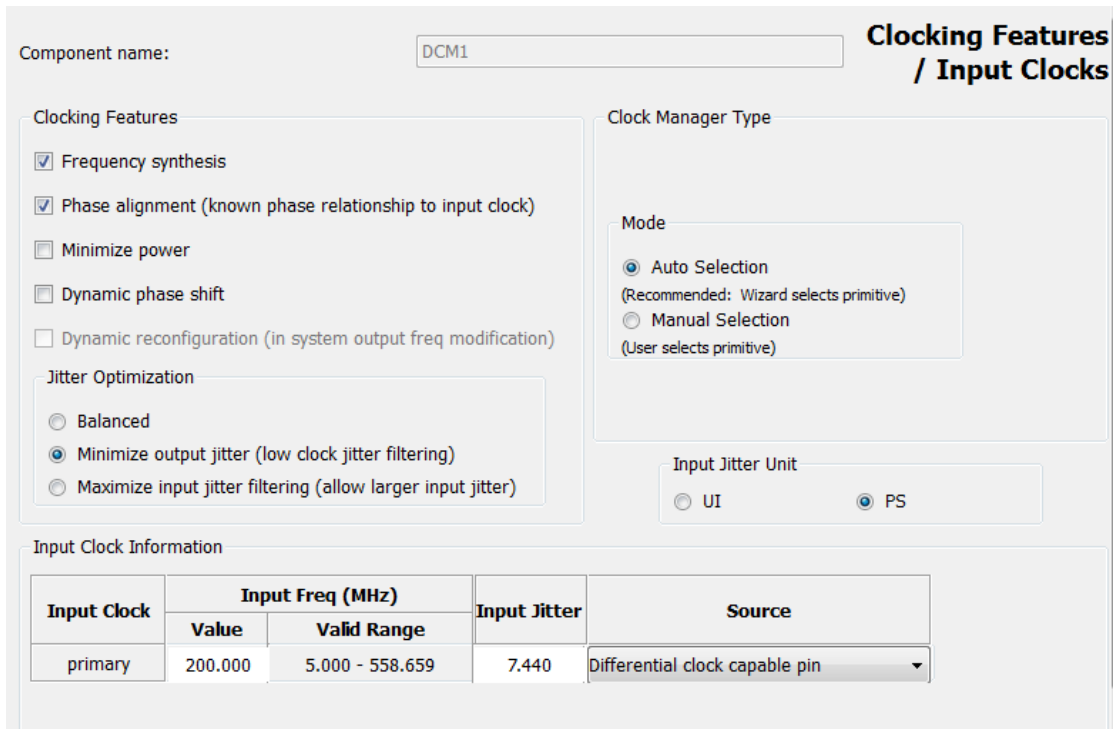


Figure 9-13: Clocking Features for Spartan-6 FPGA

Figure 9-14 shows the configuration of the output clock of the DCM Core:

- Output Clock = 36 MHz
- Duty Cycle = 50%

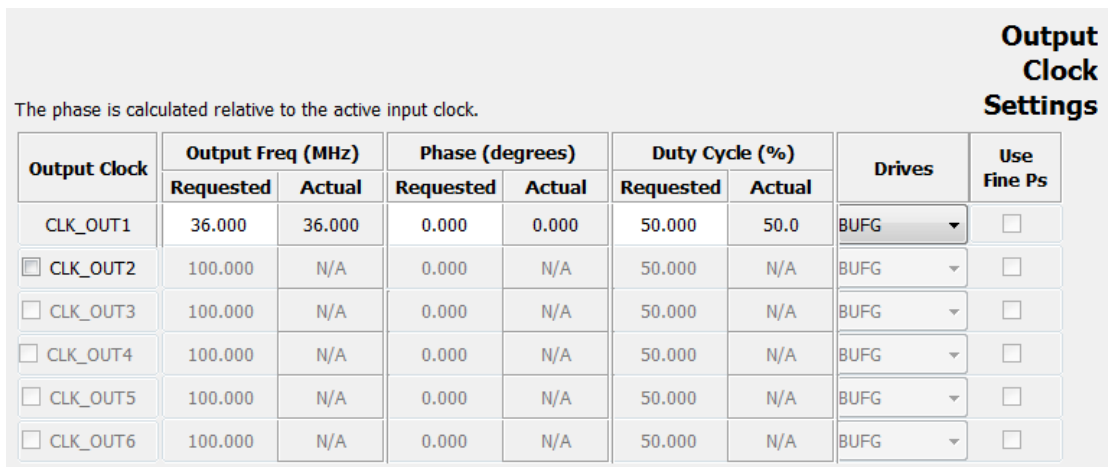
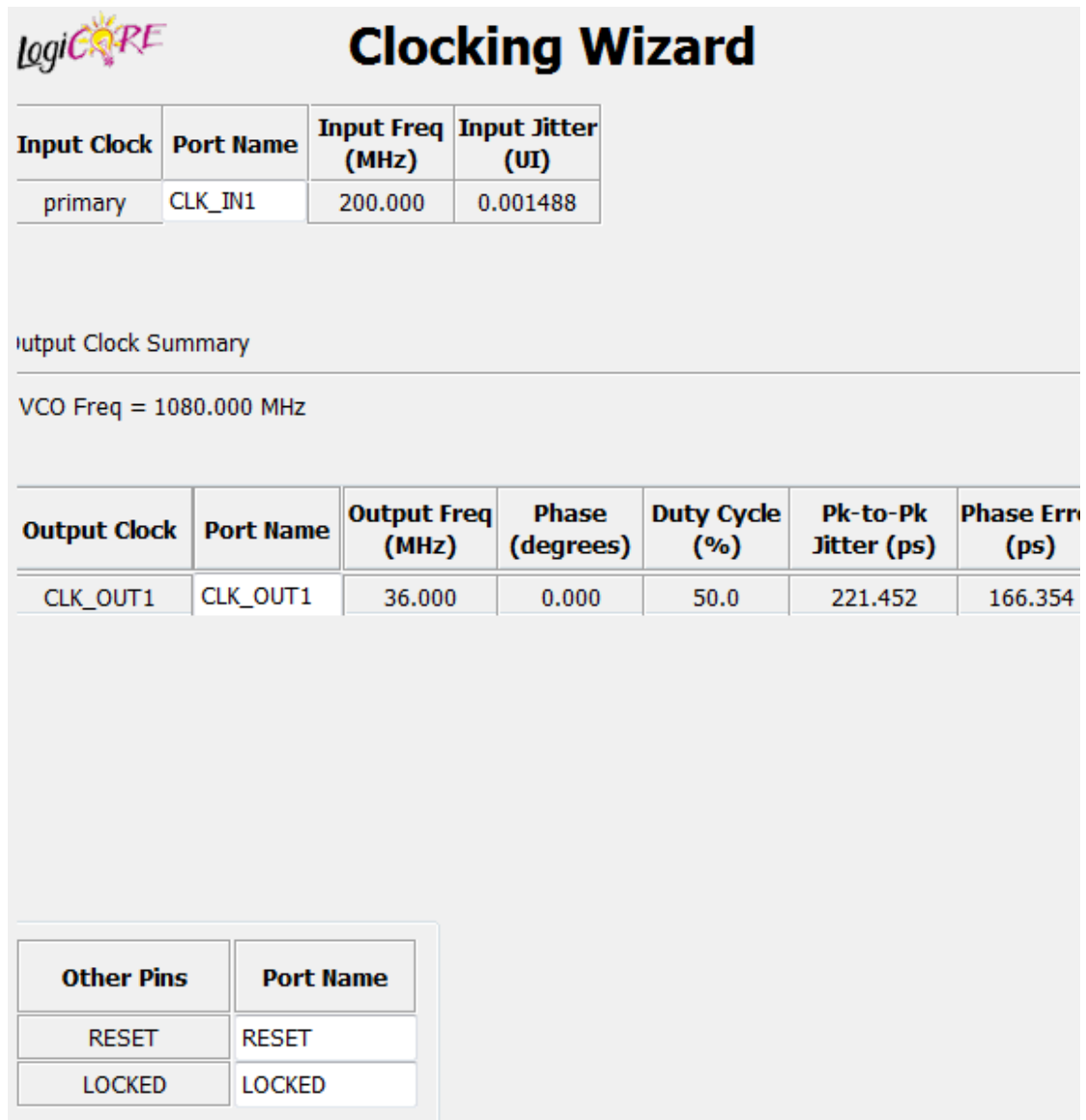


Figure 9-14: Output Clock Settings Screen

Figure 9-15 shows summary information about the input and output clocks. The DCM Core has 2 additional signals reset and locked. **Reset**: When asserted, asynchronously clears the internal state of the primitive, and causes the primitive to re-initiate the locking sequence when released. **Locked**: When asserted, indicates that the output clocks are stable and usable by downstream circuitry.



The image shows the 'Clocking Wizard' summary screen. At the top left is the 'LogiCORE' logo. The title 'Clocking Wizard' is centered at the top. Below the title is a table for input clock information. Underneath is the 'Output Clock Summary' section, which includes the VCO frequency and a table for output clock details. At the bottom left is a section for 'Other Pins' with a table for RESET and LOCKED signals.

Input Clock	Port Name	Input Freq (MHz)	Input Jitter (UI)
primary	CLK_IN1	200.000	0.001488

Output Clock Summary

VCO Freq = 1080.000 MHz

Output Clock	Port Name	Output Freq (MHz)	Phase (degrees)	Duty Cycle (%)	Pk-to-Pk Jitter (ps)	Phase Err (ps)
CLK_OUT1	CLK_OUT1	36.000	0.000	50.0	221.452	166.354

Other Pins	Port Name
RESET	RESET
LOCKED	LOCKED

Figure 9-15: Clock Summary Screen

9.1.3 Design

The Hardware design for interfacing the Turbo decoder implemented on Spartan-6 FPGA via PCIe is illustrated in this section.

As mentioned in chapter 7 that Spartan-6 FPGA sp605 kit supports PCIe Gen1 x1 which results in 32-bit interface with 62.5 MHz clock frequency and from Section 9.1.2 it is shown that Turbo decoder operates with 36 MHz clock frequency, so dual-port memories (FIFOs) are required to store data with high frequency clock (PCIe Interface) and read data with low frequency clock (Turbo Clock). Interfacing unit is also required to read data from memory, feed the turbo with the encoded symbols and I/O signals (illustrated in chapter 6) and read the decoded bits from the turbo after finishing.

Since RIFFA framework can support up to 12 channels per FPGA As mentioned in chapter 8, RIFFA HDL is configured to have 2 channels. Each channel communicates with PC thread. The First channel is used for receiving the Frame size, required iterations and Encoded symbols from the first PC thread. The second channel is used for sending the output of the Turbo (Decoded Bits) to the second PC thread.

Figure 9-16 shows the hardware designed for interfacing the Turbo decoder.

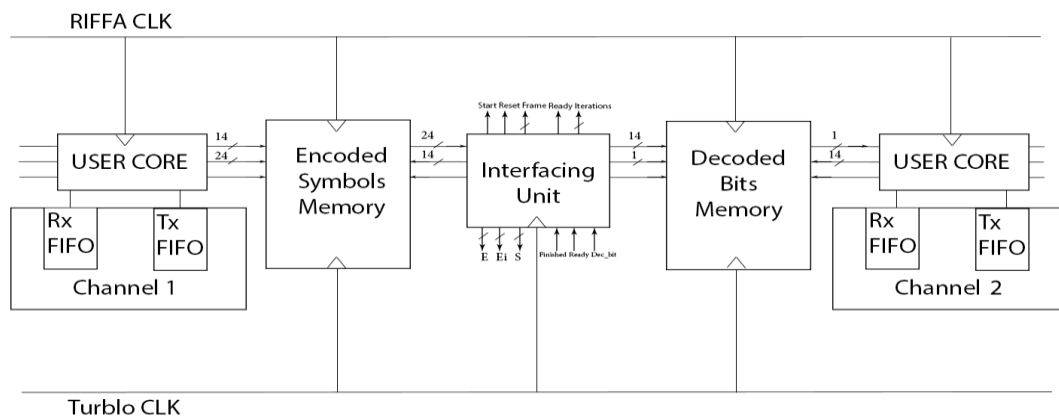


Figure 9-16: Turbo Interfacing design

The description of each component in the design follows:

- **User Core connected to channel 1**

This core reads the data received by RIFFA channel 1 through the Rx FIFO interface and writes the received data in the Encoded Symbols Memory with the same order. When a frame is written, the core notifies the interfacing unit to start reading the symbols and feed the turbo.

- **Encoded Symbols Memory**

This memory is dual-port Block RAM that has data width = 24-bit, and address width = 14-bit. This memory can store 10 frames each of 1024 bit. The first data of each frame contains the frame size and required iterations. The following data contain the encoded symbols (systematic, Encoded and Encoded interleaved). Each of these symbols is 8-bit width and stored in the memory using little Indian.

- **Interfacing Unit**

This component is responsible for interfacing with the turbo decoder directly. The interfacing unit reads the configuration and symbols of each frame from the Encoded Symbols Memory and feed the turbo with them, wait for the turbo to finish and finally writes the decoded bits into the Decoded Bits Memory and notifies the user core (connected to channel 2) after each frame written.

- **Decoded Bits Memory**

This memory is dual-port Block RAM that has data width = 1-bit (represents the decoded bit), and address width = 14-bit. This memory can store 10 frames each of 1024 bit.

- **User Core connected to channel 2**

This core writes the data to RIFFA channel 2 through the Tx FIFO interface. The core is notified by the Interfacing Unit when a frame is written in the Decoded Bits Memory and ready to be sent.

9.1.4 Synthesizing the Design

The design is synthesized using XST (Xilinx synthesis tool) provided by ISE Design suite. The target device is Spartan-6 XC6SLX45T. Figure 9-17 shows the resources used by the Turbo Decoder, Interfacing Core, RIFFA Core and PCIe Endpoint Core.

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	8616	54576	15%
Number of Slice LUTs	11869	27288	43%
Number of fully used LUT-FF pairs	6021	14464	41%
Number of bonded IOBs	12	296	4%
Number of Block RAM/FIFO	105	116	90%
Number of BUF _{FG} /BUF _{CTRLS}	7	16	43%
Number of DSP48A1s	6	58	10%
Number of PLL _{ADVs}	2	4	50%

Figure 9-17: FPGA Resources Summary

9.1.5 Configure Target Device

The design is implemented (translation, mapping, placing and routing) using Xilinx ISE Design Suite. Configuration options available for Spartan-6 FPGA SP605 Evaluation Kit are shown in Figure 9-18. Since the Kit needs to configure fast on start-up from non-volatile source, JTAG isn't an option. Configuration is done by SPI x4 Flash. [42]

Configuration Mode	M[1:0]	Bus Width	CCLK Direction	Configuration Solution	User Guide Section
Master Serial/SPI	01	1, 2, 4 ⁽¹⁾	Output	SPI X4 Memory U32 (J46 on), or External SPI Header J17 (J46 off)	3. SPI x4 Flash
Master SelectMAP/BPI ⁽²⁾	00	8, 16	Output	Linear Flash Memory U25 (BPI)	4. Linear BPI Flash
JTAG ⁽³⁾	xx	1	Input (TCK)	Xilinx Platform Cable USB plugged into J4	6. USB JTAG
Slave SelectMAP ⁽²⁾	10	8, 16	Input	System ACE CF Controller and CompactFlash Card	5. System ACE CF and CompactFlash Connector
Slave Serial ⁽⁴⁾	11	1	Input	Not Supported	-

Figure 9-18: sp605 Configuration options

For the sp605 kit to be configured from SPI x4 Flash, FPGA start-up clock (from the Generate Programming File process properties) must be set to CCLK as shown in Figure 9-19.

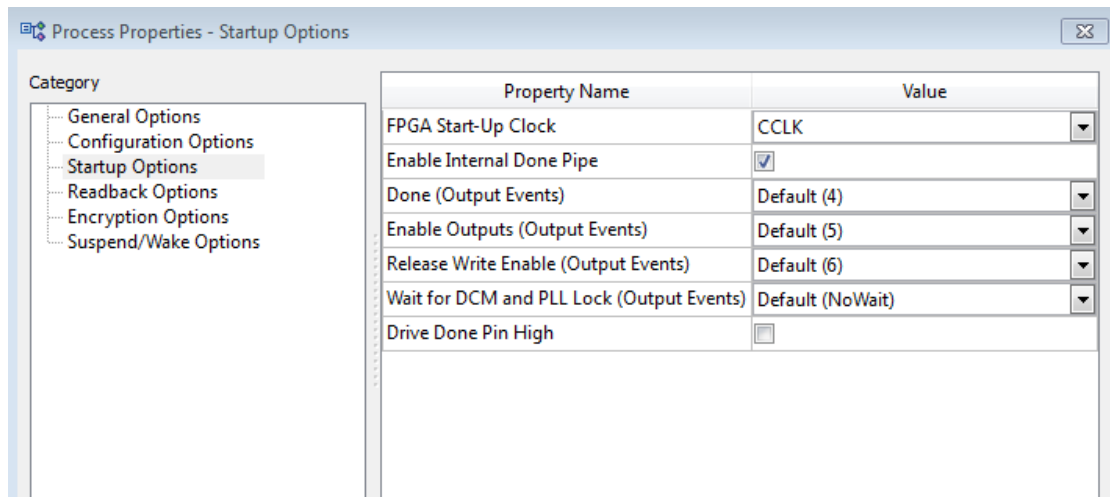


Figure 9-19: FPGA Startup Options

The generated configuration file (.bit) is converted to a PROM file (.MCS) using IMPACT to program the SPI x4 Flash with the design [42]. The sp605 kit is programmed with the generated (.MCS) file using IMPACT and configured to load the configuration file from SPI x4 Flash on start-up. [42]

9.2 Software Interface

9.2.1 Installing the RIFFA Driver

To install the RIFFA driver in linux, it must be built against the installed version of the Linux kernel. RIFFA 2.1 comes with a **makefile** that will install the necessary linux kernel headers and the driver. This **makefile** will also build and install the C/C++ native library. To install RIFFA 2.1 in linux, follow these instructions:

1. Open a terminal in linux and navigate to the **RIFFA 2.1/source/driver/linux** directory.
2. Ensure you have the kernel headers installed, run:

\$ sudo make setup

This will attempt to install the kernel headers using your system's package manager. You can skip this step if you've already installed the kernel headers.

3. Compile the driver and C/C++ library:

```
$ make
```

or

```
$ make debug
```

Using make debug will compile in code to output debug messages to the system log at runtime. These messages are useful when developing your design. However they pollute your system log and incur some overhead. So you may want to install the non-debug version after you've completed development.

4. Install the driver and library:

```
$ sudo make install
```

The system will be configured to load the driver at boot time. The C/C++ library will be installed in the default library path. The header files will be placed in the default include path. You will need to reboot after you've installed for the driver to be (re)loaded.

5. If the driver is installed and there is a RIFFA 2.1 configured FPGA when the computer boots, the driver will detect it. Output in the system log will provide additional information.
6. The C/C++ code must include the **riffa.h** header.
7. When compiling (using GCC/G++, etc.) you must link with the RIFFA libraries using the **-lriffa** flag

```
$ gcc -g -c -o test.o test.c // Compiling with gcc to produce the object file
```

```
$ gcc -g -lriffa -o test test.o // Linking with riffa library
```

9.2.2 User Space Application

The user space application uses the functions in the RIFFA C Library illustrated in Chapter 6 for testing the driver with the hardware. Since the application cannot send to RIFFA channel and receive from it simultaneously, the application had to use

POSIX thread libraries (standards based thread API for C/C++) to make use of the full duplex communication feature of PCI Express. The application creates 2 threads that run simultaneously. The first thread sends the data to RIFFA channel 1 on the FPGA to be decoded by the Turbo Decoder. The second thread receives the decoded bits from channel 2 in the FPGA. Both threads timeout is set to 0, so they wait for the FPGA to respond.

After installing the RIFFA driver and library as shown in Section 9.2.1, the developed C code is compiled using GCC and linked using both the RIFFA library and POSIX threads library as follows:

1. Compiling with GCC to produce the object file

```
$ gcc -g -c -o TurboDecoder.o TurboDecoder.c
```

2. Linking with RIFFA library and pthread library

```
$ gcc -g -o TurboDecoder TurboDecoder.o -lriffa -pthread
```

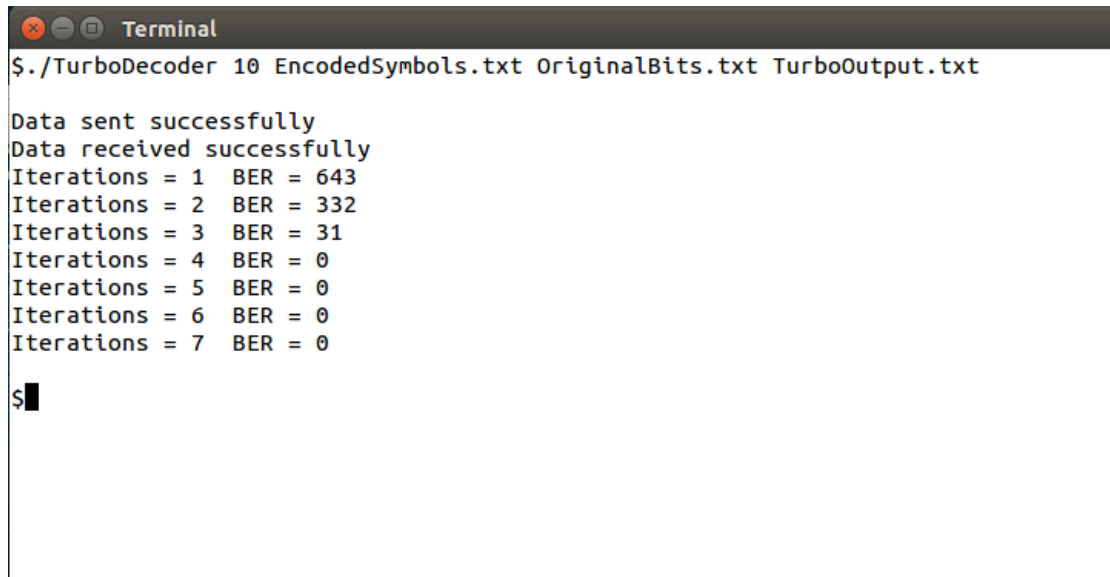
The input arguments to the user application are:

- **< Number of Frames >** Number of Frames to be sent to the Turbo Decoder
- **< Symbols File >** Text file contains each frame size, required iterations and encoded symbols
- **< Original File >** Text file contains the original bits to calculate the BER
- **< Output File >** Text file to write the Turbo output

The following example shows how to use the application

```
$ ./TurboDecoder < Number of Frames > < Symbols File > < Original  
File > < Output File >
```

Figure 9-20 shows the BER of **10 Frames** each of size **1024** sent to the Turbo Decoder implemented on **SP605 Evaluation Kit** plugged into PCI Express slot on PC. The BER is shown for different iterations.

A terminal window titled "Terminal" with a dark background and light text. The window shows the execution of a Turbo Decoder program. The command executed is `./TurboDecoder 10 EncodedSymbols.txt OriginalBits.txt TurboOutput.txt`. The output shows that data was sent and received successfully. The BER (Bit Error Rate) is reported for 7 iterations: Iterations = 1 BER = 643, Iterations = 2 BER = 332, Iterations = 3 BER = 31, Iterations = 4 BER = 0, Iterations = 5 BER = 0, Iterations = 6 BER = 0, and Iterations = 7 BER = 0. The prompt `$` is visible at the end of the output.

```
Terminal
$./TurboDecoder 10 EncodedSymbols.txt OriginalBits.txt TurboOutput.txt

Data sent successfully
Data received successfully
Iterations = 1 BER = 643
Iterations = 2 BER = 332
Iterations = 3 BER = 31
Iterations = 4 BER = 0
Iterations = 5 BER = 0
Iterations = 6 BER = 0
Iterations = 7 BER = 0
$
```

Figure 9-20: Turbo Performance on Hardware

Chapter 10: **Conclusion**

C-RAN is a promising solution to the challenges mentioned above. By using new technologies, we can change the network construction and deployment ways, fundamentally change the cost structure of mobile operators, and provide more flexible and efficient services to end users. With the distributed RRH and centralized BBU architecture, a very computationally heavy block of this technology is turbo coding, an error correction code for reaching near Shannon limit (optimum) coding performance. Due to the computational complexity of this block, it is suggested to offload this block especially when trying to realize multiple cells processing or what so called “C-RAN”. So building a pluggable FPGA-based LTE coprocessor that could be connected to many-cores GPP platform using PCI interface was our solution.

Turbo codes are a class of convolution code which exhibit the properties of large block codes through the use of recursive coders. Coder performance is heavily dependent on the design of the interleaver, which must ensure adequate weight for at least one of the codes. Soft decoders are used with turbo codes to allow the a posteriori probability to be passed between decoder iterations, Sliding window implementations of 3G turbo decoder were presented, the BER performance results demonstrate that while both decoders can achieve small BERs at low signal to noise ratios, sliding window SOVA based decoder has better performance and can achieve faster decoding speeds than Max-Log-MAP.

RIFFA proved to be one of the best implementations of PCIe endpoints, that is opensource and free to use, and in comparison with intellectual property implementations is more advanced and uses state of the art techniques in its code. The bandwidth measured is pretty close to the peak limit of the PCIe link, especially for big transfers. The software interface supports most of the popular programming languages and was easy to use.

The process of implementing the Turbo decoder HDL on Spartan-6 FPGA SP605 Evaluation Kit and interfacing it through PCIe link with workstation that has Linux kernels 2.6.27+ was completed and proved to possess a great potential for C-RAN hardware acceleration.

References

- [1] H. Taoka, "Views on 5G," DoCoMo, WWRF21, Dusseldorf, Germany, Tech. Rep., Oct 2011.
- [2] China Mobile Research Institute, C-RAN: The Road Towards Green RAN White Paper Version 2.5 (Oct, 2011)
- [3] Jun Wu, Zhifeng Zhang, Yu Hong, and Yonggang Wen, Cloud Radio Access Network (C-RAN): A Primer January/February 2015
- [4] C. E. Shannon, "A Mathematical Theory of Communication", Bell System Technical Journal, Vol. 27, pp. 379-423 (Part One), pp. 623-656 (Part Two), Oct. 1948.
- [5] S. Lin and D. J. Costello, Jr., "Error Control Coding: Fundamentals and Applications", Prentice Hall: Englewood Cliffs, NJ, 1983.
- [6] B. Vucetic and J. Yuan, "Turbo Codes Principles and Applications", 1st ed. New York, 2000.
- [7] <http://web.mit.edu/6.02/www/f2010/handouts/lectures/L8.pdf> [Accessed 16 Dec. 2016].
- [8] B. W. Werling, "A Hardware Implementation of the Soft Output Viterbi Algorithm for Serially Concatenated Convolutional Codes", Kansas Uni.
- [9] <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-02-introduction-to-eecs-ii-digital-communication-systems-fall-> [Accessed 16 Dec. 2016].
- [10] <https://theses.lib.vt.edu/theses/available/etd-71897-15815/unrestricted/chap3.pdf> [Accessed 16 Dec. 2016].
- [11] <https://theses.lib.vt.edu/theses/available/etd-71897-15815/unrestricted/chap4.pdf> [Accessed 16 Dec. 2016].
- [12] Berrou, C., Glavieux, A., and Thitimajshima, P., "Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes," Proceedings of ICC 1993, Geneva, Switzerland, pp. 1064-1070, May 1993.
- [13] Baail, G., Berrou, C., and Glavieux, A., "Psuedo-Random Recursive Convolutional Coding for Near-Capacity Performance," GLOBECOM 1993, pp. 23-27, Dec. 1993.
- [14] 3GPP Technical Specification: Group Radio Access Network, Evolved Universal Terrestrial Radio Access, Multiplexing and Channel Coding (Release 10), TS 36.212 v10.1.0, March 2011.

- [15] 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. 248-287, March 1974.
- [16] G. Battail, "Ponderation des Symboles Dtcodes par SAgorithmem de Viterbi," *Ann. Teleconimiin., Fr.*, vol. 42, no. 1-2, pp. 3 1 -38, Jan. 1987.
- [17] Z. Wang and K. Parhi, \High performance, high throughput turbo/SOVA decoder design," *Communications, IEEE Transactions on*, vol. 51, pp. 570{579, April 2003.
- [18] L. Sabeti, "New Design of a MAP Decoder." Slides, 2004.
- [19] Hagenauer, J. and Hoehner, P., "A Viterbi Algorithm with Soft-Decision Outputs and Its Applications," *GLOBECOM 1989*, Dallas, Texas, pp.1680-1686, Nov. 1989.
- [20] Berrou, C., Adde, P. Angui, E., and Faudeil, S., "A Low Complexity Soft-Output Viterbi Decoder Architecture," *Proceedings of ICC 1993*, Geneva, Switzerland, pp. 737-740, May 1993.
- [21] Hagenauer, J., Robertson, P., and Papke, L., "Iterative ("Turbo") Decoding of Systematic Convolutional Codes with the MAP and SOVA Algorithms," *Proceeding of ITG*, pp. 21-29, Oct. 1994.
- [22] Hagenauer, J., "Source-Controlled Channel Decoding," *IEEE Transactions on Communications*, Vol. 43, No. 9, pp. 2449-2457, Sept. 1995.
- [23] O. Joeressen and H. Meyr. A 40 Mb/s soft-output viterbi decoder. *IEEE Journal of Solid-State Circuits* 30(7):812–817, July 1995.
- [24] E. Yeo, S. Augsberger, W. R. Davis, and B. Nikolić. 500 Mb/s soft-output viterbi decoder. *ESSCIRC* pp. 523–526, 2002.
- [25] O. Joeressen, M. Vaupel, and H. Meyr. High-speed VLSI architectures for soft-output viterbi decoding. *Proceedings of the International Conference on Applications Specific Array Processors* pp. 373–384, August 1992.
- [26] Z. Wang, Z. Chi, and K. K. Parhi, "Area efficient high-speed decoding schemes for turbo decoders," *IEEE Trans. VLSI Syst.*, vol. 10, pp. 902–912, Dec. 2002.
- [27] Michel, H. and When, N., "Turbo-Decoder Quantization for UMTS," *IEEE Communications Letters*, pp. 55-57, February 2001.
- [28] Xilinx.com. (2017). ISE Design Flow Overview. [Online] Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_c_fpga_design_flow_overview.htm [Accessed 16 Jul. 2017].
- [29] PCI SIG. PCI Express® Base Specification Revision 2.1. PCI SIG, 2009.

- [30]
http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130812_PreConfD_Onufryk.pdf [Accessed 16 Dec. 2016].
- [31] Altera.com. (2017). FPGA CPLD and ASIC from Intel PSG. [Online] Available at: <https://www.altera.com/products/intellectual-property/ip/interface-protocols/mpci-express-protocol.html> [Accessed 16 Jul. 2017].
- [32] Xilinx.com. (2017). Spartan-6 FPGA SP605 Evaluation Kit. [Online] Available at: <https://www.xilinx.com/products/boards-and-kits/ek-s6-sp605-g.html#overview> [Accessed 16 Jul. 2017].
- [33]
https://www.xilinx.com/support/documentation/ip_documentation/s6_pcie_ds718.pdf
[Accessed 16 Dec. 2016].
- [34] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. Linux Device Drivers, 3rd Edition. O'Reilly Media, Inc., 2005.
- [35] Essential Linux device drivers (Prentice Hall)
- [36] Eli Billauer. The anatomy of a pci/pci express kernel driver. <http://haifux.org/lectures/256/>.
- [37] UNIVERSITY OF CALIFORNIA, SAN DIEGO Smart Frame Grabber: A Hardware Accelerated Computer Vision Framework. A dissertation submitted in partial satisfaction of the requirements for the degree of Doctor of Philosophy in Computer Science by Matthew Daniel Jacobsen.
- [38] Matthew Jacobsen, Dustin Richmond, Matthew Hogains, and Ryan Kastner. Riffa 2.1: A reusable integration framework for fpga accelerators. ACM Trans. Reconfigurable Technol. Syst., 8(4):22:1–22:23, September 2015.
- [39] Riffa: Reusable integration framework for fpga accelerators. <http://riffa.ucsd.edu>.
- [40] https://www.xilinx.com/support/documentation/boards_and_kits/ug526.pdf
- [41] SiTime *SiT9102 Data Sheet*
- [42] https://www.xilinx.com/support/documentation/boards_and_kits/xtp065.pdf

Appendix A: Interleaver Table

<i>I</i>	<i>K</i>	f_1	f_2	<i>i</i>	<i>K</i>	f_1	f_2	<i>i</i>	<i>K</i>	f_1	f_2	<i>i</i>	<i>K</i>	f_1	f_2
1	40	3	10	48	416	25	52	95	1120	67	140	142	3200	111	240
2	48	7	12	49	424	51	106	96	1152	35	72	143	3264	443	204
3	56	19	42	50	432	47	72	97	1184	19	74	144	3328	51	104
4	64	7	16	51	440	91	110	98	1216	39	76	145	3392	51	212
5	72	7	18	52	448	29	168	99	1248	19	78	146	3456	451	192
6	80	11	20	53	456	29	114	100	1280	199	240	147	3520	257	220
7	88	5	22	54	464	247	58	101	1312	21	82	148	3584	57	336
8	96	11	24	55	472	29	118	102	1344	211	252	149	3648	313	228
9	104	7	26	56	480	89	180	103	1376	21	86	150	3712	271	232
10	112	41	84	57	488	91	122	104	1408	43	88	151	3776	179	236
11	120	103	90	58	496	157	62	105	1440	149	60	152	3840	331	120
12	128	15	32	59	504	55	84	106	1472	45	92	153	3904	363	244
13	136	9	34	60	512	31	64	107	1504	49	846	154	3968	375	248
14	144	17	108	61	528	17	66	108	1536	71	48	155	4032	127	168
15	152	9	38	62	544	35	68	109	1568	13	28	156	4096	31	64
16	160	21	120	63	560	227	420	110	1600	17	80	157	4160	33	130
17	168	101	84	64	576	65	96	111	1632	25	102	158	4224	43	264
18	176	21	44	65	592	19	74	112	1664	183	104	159	4288	33	134
19	184	57	46	66	608	37	76	113	1696	55	954	160	4352	477	408
20	192	23	48	67	624	41	234	114	1728	127	96	161	4416	35	138
21	200	13	50	68	640	39	80	115	1760	27	110	162	4480	233	280
22	208	27	52	69	656	185	82	116	1792	29	112	163	4544	357	142
23	216	11	36	70	672	43	252	117	1824	29	114	164	4608	337	480
24	224	27	56	71	688	21	86	118	1856	57	116	165	4672	37	146
25	232	85	58	72	704	155	44	119	1888	45	354	166	4736	71	444
26	240	29	60	73	720	79	120	120	1920	31	120	167	4800	71	120
27	248	33	62	74	736	139	92	121	1952	59	610	168	4864	37	152
28	256	15	32	75	752	23	94	122	1984	185	124	169	4928	39	462
29	264	17	198	76	768	217	48	123	2016	113	420	170	4992	127	234
30	272	33	68	77	784	25	98	124	2048	31	64	171	5056	39	158
31	280	103	210	78	800	17	80	125	2112	17	66	172	5120	39	80
32	288	19	36	79	816	127	102	126	2176	171	136	173	5184	31	96
33	296	19	74	80	832	25	52	127	2240	209	420	174	5248	113	902
34	304	37	76	81	848	239	106	128	2304	253	216	175	5312	41	166
35	312	19	78	82	864	17	48	129	2368	367	444	176	5376	251	336
36	320	21	120	83	880	137	110	130	2432	265	456	177	5440	43	170
37	328	21	82	84	896	215	112	131	2496	181	468	178	5504	21	86
38	336	115	84	85	912	29	114	132	2560	39	80	179	5568	43	174
39	344	193	86	86	928	15	58	133	2624	27	164	180	5632	45	176
40	352	21	44	87	944	147	118	134	2688	127	504	181	5696	45	178
41	360	133	90	88	960	29	60	135	2752	143	172	182	5760	161	120
42	368	81	46	89	976	59	122	136	2816	43	88	183	5824	89	182
43	376	45	94	90	992	65	124	137	2880	29	300	184	5888	323	184
44	384	23	48	91	1008	55	84	138	2944	45	92	185	5952	47	186
45	392	243	98	92	1024	31	64	139	3008	157	188	186	6016	23	94
46	400	151	40	93	1056	17	66	140	3072	47	96	187	6080	47	190
47	408	155	102	94	1088	171	204	141	3136	13	28	188	6144	263	480

Appendix B: Path metric derivation

The fundamental Viterbi algorithm searches for the state sequence $\mathbf{S}^{(m)}$ or the information sequence $\mathbf{u}^{(m)}$ that maximizes the a-posteriori probability $P(\mathbf{S}^{(m)}|\mathbf{y})$. For binary ($k=1$) trellises, m can be either 1 or 2 to denote the survivor and the competing paths respectively. By using Bayes' Theorem, the a-posteriori probability can be expressed as

Since the received sequence \mathbf{y} is fixed for metric computation and does not depend on m , it can be discarded. Thus, the maximization results to

$$\max_m p(\mathbf{y} | \mathbf{S}^{(m)})P(\mathbf{S}^{(m)})$$

The probability of a state sequence terminating at time t is $P(\mathbf{S}_t)$. This probability can be calculated as

$$P(\mathbf{S}_t) = P(\mathbf{S}_{t-1})P(\mathbf{S}_t) = P(\mathbf{S}_{t-1})P(u_t)$$

Where $P(\mathbf{S}_t)$ and $P(u_t)$ denote the probability of the state and the bit at time t respectively. The maximization can then be expanded to

$$\max_m p(\mathbf{y} | \mathbf{S}^{(m)})P(\mathbf{S}^{(m)}) = \max_m \left\{ \prod_{i=0}^t p(y_i | \mathbf{S}_{i-1}^{(m)}, \mathbf{S}_i^{(m)})P(\mathbf{S}_t^{(m)}) \right\}$$

Where $(\mathbf{S}_{i-1}^{(m)}, \mathbf{S}_i^{(m)})$ denotes the state transition between time $i-1$ and time i and y_i denotes the associated received channel values for the state transition. After substituting and rearranging,

$$\max_m p(\mathbf{y} | \mathbf{S}^{(m)})P(\mathbf{S}^{(m)}) = \max_m \left\{ P(\mathbf{S}_{t-1}^{(m)}) \prod_{i=0}^{t-1} p(y_i | \mathbf{S}_{i-1}^{(m)}, \mathbf{S}_i^{(m)})P(u_t^{(m)})p(y_t | \mathbf{S}_{t-1}^{(m)}, \mathbf{S}_t^{(m)}) \right\}$$

Note that

$$p(y_t | \mathbf{S}_{t-1}^{(m)}, \mathbf{S}_t^{(m)}) = \prod_{j=1}^N p(y_{t,j} | x_{t,j}^{(m)})$$

Thus, the maximization becomes

$$\max_m \left\{ P(S_{t-1}^{(m)}) \prod_{j=0}^{t-1} p(y_j | S_{i-1}^{(m)}, S_i^{(m)}) P(u_t^{(m)}) \prod_{j=1}^N p(y_{t,j} | x_{t,j}^{(m)}) \right\}$$

This maximization is not changed if logarithm is applied to the whole expression, multiplied by 2, and added two constants that are independent of m. This leads to

$$\max_m \{M_t^{(m)}\} = \max_m \left\{ M_{t-1}^{(m)} + [2 \ln P(u_t^{(m)}) - C_u] + \sum_{j=1}^N [2 \ln p(y_{t,j} | x_{t,j}^{(m)}) - C_y] \right\}$$

Where

$$\frac{M_{t-1}^{(m)}}{2} = \ln \left(P(S_{t-1}^{(m)}) \prod_{i=0}^{t-1} p(y_i | S_{i-1}^{(m)}, S_i^{(m)}) \right)$$

And for convenience, the two constants are

$$C_u = \ln P(u_t = +1) + \ln P(u_t = -1)$$

$$C_y = \ln(p(y_{t,j} | x_{t,j} = +1)) + \ln(p(y_{t,j} | x_{t,j} = -1))$$

After substitution of these two constants, the SOVA metric is obtained as

$$M_t^{(m)} = M_{t-1}^{(m)} + \sum_{j=1}^N x_{t,j}^{(m)} \ln \frac{p(y_{t,j} | x_{t,j} = +1)}{p(y_{t,j} | x_{t,j} = -1)} + u_t^{(m)} \ln \frac{P(u_t = +1)}{P(u_t = -1)}$$

And is reduced to

$$M_t^{(m)} = M_{t-1}^{(m)} + \sum_{j=1}^N x_{t,j}^{(m)} L_c y_{t,j} + u_t^{(m)} L(u_t)$$

For systematic codes, this can be modified to become

$$M_t^{(m)} = M_{t-1}^{(m)} + u_t^{(m)} L_c y_{t,1} + \sum_{j=2}^N x_{t,j}^{(m)} L_{\alpha,j} y_{t,j} + u_t^{(m)} L(u_t)$$

As seen from the two previous equations, the SOVA metric incorporates values from the past metric, the channel reliability, and the source reliability (a-priori value).

Appendix C: MATLAB Code

Main Function for test

```
clc;
clear all;

% Configurations
L_total = 1024;           % Frame Length
rate = 1/3;              % Code rate
a = 1;                   % Fading amplitude; a=1 in AWGN channel
SNRdB = 0:0.5:4;        % SNRdB Values
iterations = 10;        % Number of Turbo iterations
frames = 100000;        % Number of frames

% Init
errs = zeros(1,length(SNRdB));
ber = zeros(1,length(SNRdB));

for k = 1:length(SNRdB)

    SNR = 10^(SNRdB(k)/10); % convert SNRdB from unit db to Watts
    L_c = 4*a*SNR*rate;    % reliability value channel
    sigma = 1/sqrt(2*rate*SNR); % standard deviation of AWGN noise

    for frame = 1:frames

        data = round(rand(1, L_total)); % generate info bits
        encoded = LTE_Turbo_Encoder(data); % channel encoding
        modulated = 2 * encoded - ones(size(encoded)) ; % BPSK Modulation
        r = modulated +sigma*randn(1,L_total*3+12); % adding noise
        rec_s = L_c*r; % reliability scaling

        % Decoding using Turbo Decoder
        decoded_bits = LTE_Turbo_Decoder( rec_s,iterations);

        % Number of bit errors in current iteration
        err = sum(xor(decoded_bits(1:L_total),data(1:L_total)));

        % Total number of bit errors for all iterations
        errs(k) = errs(k) + err;

        % Monitoring simulation
        progress = 100*(frame/frames);
        if mod(progress,10) == 0
            fprintf('%d %% \n',progress);
        end

    end

    fprintf('SNR: %d is done \n',SNRdB(k));
    ber(k) = errs(k)/(frames*L_total); % ber = total errors/total bits

end

% Plot BER graph
figure
semilogy(SNRdB,ber);
hold on;
title('Turbo Decoder');
xlabel('SNRdB in db');
ylabel('BER');
grid on
```

Turbo Encoder Function

```
function [out] = LTE_Turbo_Encoder(input)

len = length(input);
out = zeros(1,3*len+12);

enc1 = RSC_Encoder(input);           % Parity 1
int  = internal_interleaver(input,len); % Interleaving data
enc2 = RSC_Encoder(int);             % Parity 2

% Multiplexing the encoded stream including tail bits
for k=1:len+3
    out(3*k-2)=enc1(2*k-1);
    out(3*k-1)=enc1(2*k);
    out(3*k)=enc2(2*k);
end

% special handling for interleaved systematic data
out(3*len+10) = enc2(2*len+1);
out(3*len+11) = enc2(2*len+3);
out(3*len+12) = enc2(2*len+5);
end
```

Recursive Systematic Encoder Function

```
function [output] = RSC_Encoder(input)

% Reset Shift Registers
D1 = 0;
D2 = 0;
D3 = 0;
output = zeros(1,2*length(input));

for k=1:length(input)

    % Evaluate XOR Operations
    fb0 = xor(D2,D3);
    fb1 = xor(fb0,input(k));

    out0 = input(k);
    out1 = xor(fb1,xor(D1,D3));

    output(2*k-1:2*k) = [out0 out1];

    % Shift Left
    D3 = D2;
    D2 = D1;
    D1 = fb1;
end

% tail termination
for k=length(input)+1:length(input)+3

    fb0 = xor(D2,D3);
    fb1 = xor(fb0,fb0);

    out0 = fb0;
    out1 = xor(fb1,xor(D1,D3));

    output(2*k-1:2*k) = [out0 out1];

    D3 = D2;
    D2 = D1;
    D1 = fb1;
end
end
```

Turbo Interleaver Function

```
function [out] = internal_interleaver(in, K)

% Supported sizes by LTE and corresponding interleaving parameters
K_table =
[40,48,56,64,72,80,88,96,104,112,120,128,136,144,152,160,168,176,184,192,200,208,216,2
24,232,240,248,256,264,272,280,288,296,304,312,320,328,336,344,352,360,368,376,384,392
,400,408,416,424,432,440,448,456,464,472,480,488,496,504,512,528,544,560,576,592,608,6
24,640,656,672,688,704,720,736,752,768,784,800,816,832,848,864,880,896,912,928,944,960
,976,992,1008,1024,1056,1088,1120,1152,1184,1216,1248,1280,1312,1344,1376,1408,1440,14
72,1504,1536,1568,1600,1632,1664,1696,1728,1760,1792,1824,1856,1888,1920,1952,1984,201
6,2048,2112,2176,2240,2304,2368,2432,2496,2560,2624,2688,2752,2816,2880,2944,3008,3072
,3136,3200,3264,3328,3392,3456,3520,3584,3648,3712,3776,3840,3904,3968,4032,4096,4160,
4224,4288,4352,4416,4480,4544,4608,4672,4736,4800,4864,4928,4992,5056,5120,5184,5248,5
312,5376,5440,5504,5568,5632,5696,5760,5824,5888,5952,6016,6080,6144];

f1_table =
[3,7,19,7,7,11,5,11,7,41,103,15,9,17,9,21,101,21,57,23,13,27,11,27,85,29,33,15,17,33,1
03,19,19,37,19,21,21,115,193,21,133,81,45,23,243,151,155,25,51,47,91,29,29,247,29,89,9
1,157,55,31,17,35,227,65,19,37,41,39,185,43,21,155,79,139,23,217,25,17,127,25,239,17,1
37,215,29,15,147,29,59,65,55,31,17,171,67,35,19,39,19,199,21,211,21,43,149,45,49,71,13
,17,25,183,55,127,27,29,29,57,45,31,59,185,113,31,17,171,209,253,367,265,181,39,27,127
,143,43,29,45,157,47,13,111,443,51,51,451,257,57,313,271,179,331,363,375,127,31,33,43,
33,477,35,233,357,337,37,71,71,37,39,127,39,39,31,113,41,251,43,21,43,45,45,161,89,323
,47,23,47,263];

f2_table =
[10,12,42,16,18,20,22,24,26,84,90,32,34,108,38,120,84,44,46,48,50,52,36,56,58,60,62,32
,198,68,210,36,74,76,78,120,82,84,86,44,90,46,94,48,98,40,102,52,106,72,110,168,114,58
,118,180,122,62,84,64,66,68,420,96,74,76,234,80,82,252,86,44,120,92,94,48,98,80,102,52
,106,48,110,112,114,58,118,60,122,124,84,64,66,204,140,72,74,76,78,240,82,252,86,88,60
,92,846,48,28,80,102,104,954,96,110,112,114,116,354,120,610,124,420,64,66,136,420,216,
444,456,468,80,164,504,172,88,300,92,188,96,28,240,204,104,212,192,220,336,228,232,236
,120,244,248,168,64,130,264,134,408,138,280,142,480,146,444,120,152,462,234,158,80,96,
902,166,336,170,86,174,176,178,120,182,184,186,94,190,480];

out = zeros(1,K);

% Determine f1 and f2
for n=0:length(K_table)-1
    if(K == K_table(n+1))
        f1 = f1_table(n+1);
        f2 = f2_table(n+1);
        break;
    end
end

% output the interleaved data
for n=0:length(in)-1
    out(n+1) = in(mod(f1*n + f2*(n^2), K)+1);
end
end
```

Turbo Decoder Function

```
function [decoded_bits] = LTE_Turbo_Decoder(input,iterations)

len = (length(input)-12)/3;
r0 = zeros(1,len+3);           % Systematic
r0_bar = zeros(1,len+3);       % Interleaved Systematic
r1 = zeros(1,len+3);           % first encoder
r2 = zeros(1,len+3);           % second encoder (interleaved)

% tail bits
for k=1:len+3
    r0(k) = input(3*k-2);
    r1(k) = input(3*k-1);
    r2(k) = input(3*k);
end

r0_bar(1:len) = internal_interleaver(r0(1:len), len); % interleaving systematic
r0_bar(len+1:len+3) = input(3*len+10:3*len+12); % interleaved tail bits

%Initialize extrinsic information
L_e1 = zeros(1,len+3);
L_e2 = zeros(1,len+3);

for iter = 1:iterations

    % Decoder one
    L_e2_Int = [internal_deinterleaver(L_e2(1:len), len) zeros(1,3)]; % a priori
info.
    tmp_in1 = reshape([r0;r1], 1, []);
    L1 = SOVA(tmp_in1, L_e2_Int); % First Decoder
    L_e1 = L1 - r0 - L_e2_Int; % extrinsic info.
    scale = 0.5;
    L_e1 = L_e1*scale; % scaling soft output

    % Decoder two
    L_e1_Int = [internal_interleaver(L_e1(1:len), len) zeros(1,3)]; % a priori info.
    tmp_in2 = reshape([r0_bar;r2], 1, []);
    L2 = SOVA(tmp_in2, L_e1_Int); % Second Decoder
    L_e2 = L2 - r0_bar - L_e1_Int; % extrinsic info.
    L_e2 = L_e2*scale; % scaling soft output
end

% Estimate the info. bits.
LLR_est = internal_deinterleaver(L2(1:len), len);

% Hard decision output
decoded_bits = (sign(LLR_est)+1)/2;

end
```


SOVA Decoder Function

```
function softOutput = SOVA(input,LLR)

% length of input
N_bits = length(input);
infoLen = N_bits/2;

% states = [1 2 3 4 5 6 7 8]; %states(decimal) [000 -> 1 111 -> 8]
prv_0 = [1 4 5 8 2 3 6 7];
prv_1 = [2 3 6 7 1 4 5 8];
out_0 = [-1 -1 ; -1 -1 ; -1 1 ; -1 1 ; -1 1 ; -1 1 ; -1 -1 ; -1 -1];
%output when input = 0
out_1 = [1 1 ; 1 1 ; 1 -1 ; 1 -1 ; 1 -1 ; 1 -1 ; 1 1 ; 1 1];
%output when input = 1

% Init vectors
pathMetricOld = zeros(1,8);
pathMetricNew = zeros(1,8);
branchMetric0 = zeros(1,8);
branchMetric1 = zeros(1,8);

pathMetric0 = zeros(8, infoLen);
pathMetric1 = zeros(8, infoLen);

delta = zeros(8, infoLen);
ML = zeros(1,infoLen+1);
competingPath = zeros(1,infoLen+1);
output = zeros(1,infoLen);
output_c = zeros(1,infoLen);
softOutput = zeros(1,infoLen-3);

Inf = 1E5;
pathMetricOld = -Inf*ones(1,8);
pathMetricOld(1) = 0;

% Trellis Diagram
for i=1:infoLen

    symbol = input(2*i-1:2*i);
    % Evaluating the branch metrics
    for cur_state = 1:8
        branchMetric0(cur_state) = (symbol*transpose(out_0(cur_state, :))) - LLR(i);
        branchMetric1(cur_state) = (symbol*transpose(out_1(cur_state, :))) + LLR(i);
    end

    % Evaluating the path metrics
    for cur_state = 1:8
        PM0 = pathMetricOld(prv_0(cur_state)) + 0.5*branchMetric0(prv_0(cur_state));
        PM1 = pathMetricOld(prv_1(cur_state)) + 0.5*branchMetric1(prv_1(cur_state));
        delta(cur_state,i) = abs(PM1 - PM0);

        pathMetric0(cur_state,i+1) = PM0;
        pathMetric1(cur_state,i+1) = PM1;
        pathMetricNew(cur_state) = max(PM0, PM1);
    end

    pathMetricOld = pathMetricNew;
end

end
```

```

% Finding state of max. path metric
[~,ML(infoLen+1)] = max(pathMetricOld);

% Trace back to find the Most-likely path (ML)
state = ML(infoLen+1);
for i=infoLen:-1:1

    if pathMetric0(state,i+1) > pathMetric1(state,i+1)
        output(i) = 0;
        softOutput(i) = -1*delta(state,i);
        state = prv_0(state);
        ML(i) = state;
    else
        output(i) = 1;
        softOutput(i) = +1*delta(state,i);
        state = prv_1(state);
        ML(i) = state;
    end
end

% Updating the soft Output
delta_vector = abs(softOutput);

% Find the last state in competing path sequence
for m = 1:infoLen

    if prv_0(ML(m+1)) == ML(m)
        competingPath(m) = prv_1(ML(m+1));
    else
        competingPath(m) = prv_0(ML(m+1));
    end

    updateDelta = delta_vector(m);
    state = competingPath(m);

    % Trace back to find the competing path
    for i=m-1:-1:1
        if pathMetric0(state,i+1) > pathMetric1(state,i+1)
            state = prv_0(state);
            output_c(i) = 0;
        else
            state = prv_1(state);
            output_c(i) = 1;
        end

        if output_c(i) ~= output(i) && updateDelta < delta_vector(i)
            delta_vector(i) = updateDelta;
        end
    end

end

end

% Output the soft data multiplied by the corresponding sign
for i=1:infoLen
    if(output(i) == 0)
        softOutput(i) = -1*delta_vector(i);
    else
        softOutput(i) = +1*delta_vector(i);
    end
end

softOutput = softOutput(1:infoLen);

end

```