SOFTWARE DEFINED RADIO-BASED TRANSCEIVER SYSTEM ON LOW POWER MULTI-PROCESSOR SYSTEM-ON-CHIP FOR INTERNET OF THINGS

By

DAREEN KUSUMA HALIM

A dissertation submitted to the Department of Electronic Engineering,
Faculty of Engineering and Green Technology,
Universiti Tunku Abdul Rahman,
in partial fulfillment of the requirements for the degree of
Master of Engineering Science
March 2018

For my mom and dad who never stop loving and believing in me,

And my sisters and brother

ABSTRACT

SOFTWARE DEFINED RADIO-BASED TRANSCEIVER SYSTEM ON LOW POWER MULTI-PROCESSOR SYSTEM-ON-CHIP FOR INTERNET OF THINGS

Dareen Kusuma Halim

Low error rate wireless link is essential for Internet of Things (IoT) technology as it provides data integrity as well as the all important power saving due to the less retransmission required. A widely used method to reduce error rate is Forward Error Correction (FEC) code which adds parity to the data block, allowing self-correction of the received data. Turbo Code is a widely adapted FEC that provides near Shannon-limit correction capability but has relatively expensive computational requirement. Meanwhile, the growth of IoT applications has pushed the number of development and improvement for wireless standards suitable for IoT applications. Software-defined Radio (SDR) offers reconfigurability to the wireless system to adapt to these developments without the need of hardware change.

In this research, a fully functional wireless transceiver system is built based on SDR implementation on a low power Multi-Processor System-on-Chip (MPSoC), paired with a programmable radio frontend. The

implementation comprises of Turbo encoder/decoder paired with a physical layer stack of basic modulation and demodulation as the building blocks to realize a whole transceiver system. Both the Turbo Code and the modulation/demodulation parts are fully implemented in software through proper optimization of the processor's architecture and resources. Preliminary studies and simulations on the modulation/demodulation, as well as the Turbo Code structure and parameters are also conducted before the actual deployment on the target hardware.

Actual testing of wireless transmission and reception with the transceiver modules are performed in a simplex manner, comparing the system with and without Turbo Code. The testing shown that the system is fully functional with the desired behavior in terms of error rate, which goes down along with the transmit power increment and shorter transmission range. Likewise, significantly lower error rate is exhibited by the system with the Turbo Code. However, the lowest BER of 10-1 achieved by the system is far from the practical value of 10-6. This is affected by loss of computation accuracy in the simplified algorithm as well as numerous factors such as design of RF region, test environment which are not in the scope of this work.

Despite having air time of 35-37%, the programmable RF frontend consumes five times the power consumed by the baseband processor. Hence, the system can be improved further by lowering the radio air time with faster baseband rate while still considering the error rate, or by prolonging the period of RF communication hence lowering the air time in long run.

ACKNOWLEDGEMENT

My utmost gratitude towards my supervisors, Mr. Tang Chong Ming, Prof. Lee Sze Wei, and Mr. Ng Mow Song for the time, effort, and immense knowledge contributed throughout my Master study and research. The completion of this dissertation would not have been possible without their guidance.

My sincere thanks to Dicky Hartono for his contributions and ideas throughout the research process as well as the friendship. My thanks also go to Felix Lokananta, Arya Wicaksana, and Lim Zhen Ning who has been my seniors and friends during the research period. To my friends, Mui Kai Meng, Thee Kang Wei, Ken Chow, and others, thank you for the good time during my stay.

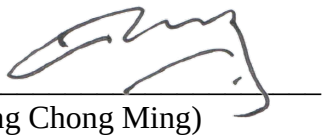
I would also like to thank Prof. Muliawati G. Siswanto and Mr. Kanisius Karyono, S.T., M.T. from UMN for the opportunity of joining this research study.

Lastly, my thanks to lecturers and staffs of UTAR Perak Campus, especially the Faculty of Green Engineering Technology (FEGT), the Department of International Student Services (DISS), and the Institute of Postgraduate Study and Research (IPSR), for their assistance during the study.

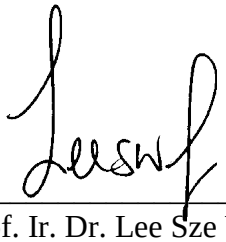
APPROVAL SHEET

This dissertation/thesis entitled **"SOFTWARE DEFINED RADIO-BASED TRANSCEIVER SYSTEM ON LOW POWER MULTI-PROCESSOR SYSTEM-ON-CHIP FOR INTERNET OF THINGS"** was prepared by DAREEN KUSUMA HALIM and submitted as partial fulfillment of the requirements for the degree of Master of Engineering Science at Universiti Tunku Abdul Rahman.

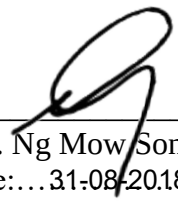
Approved by:



(Mr. Tang Chong Ming)
Date:…31-08-2018……..
Supervisor
Department of Electronic Engineering
Faculty of Engineering and Green Technology
Universiti Tunku Abdul Rahman



(Prof. Ir. Dr. Lee Sze Wei)
Date:…31-08-2018……..
Co-supervisor
Department of Electrical and Electronic Engineering
Faculty of Engineering and Science
Universiti Tunku Abdul Rahman



(Mr. Ng Mow Song)
Date:…31-08-2018……..
External Co-supervisor
GLX Technologies Sdn., Bhd.
Kuala Lumpur
Malaysia

FACULTY OF ENGENEERING AND GREEN TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

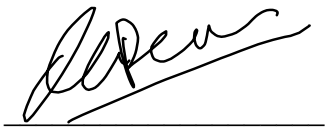
Date: 23 March 2018

SUBMISSION OF FINAL YEAR PROJECT /DISSERTATION/THESIS

It is hereby certified that **Dareen Kusuma Halim** (ID No: **15AGM01274**) has completed this dissertation entitled **"SOFTWARE DEFINED RADIO-BASED TRANSCEIVER SYSTEM ON LOW POWER MULTI-PROCESSOR SYSTEM-ON-CHIP FOR INTERNET OF THINGS"** under the supervision of Mr. Tang Chong Ming (supervisor) from the Department of Electronic Engineering, Faculty of Engineering and Green Technology, Prof. Ir. Dr. Lee Sze Wei (co-supervisor) from the Department of Electrical and Electronic Engineering, Faculty of Engineering and Science, and Mr. Ng Mow Song (external co-supervisor) from GLX Technologies Sdn., Bhd., Malaysia.

I understand that University will upload softcopy of my dissertation in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

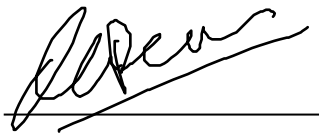
Yours truly,



(Dareen Kusuma Halim)

DECLARATION

I hereby declare that the dissertation is based on my original work except for quotations and citations which have been duly acknowledged. I also declare that it has not been previously or concurrently submitted for any other degree at UTAR or other institutions.



(Dareen Kusuma Halim)

Date 23 March 2018

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADC	Analog-to-Digital Converter
AES	Advanced Encryption Standard
AHB	Advanced High-performance Bus
ASIC	Application Specific Integrated Circuit
ASK	Amplitude-Shift Keying
AWGN	Additive White Gaussian Noise
BCJR	Bahl-Cocke-Jeinek-Raviv
BER	Bit Error Rate
BPSK	Binary Phase-Shift Keying
CCC	Communication and Configuration Controller
CMSIS	Cortex Microcontroller Software Interface Standard
DAC	Digital-to-Analog Converter
DMA	Direct Memory Access
DRP	Dithered Relative Prime
DSP	Digital Signal Processor
DSSS	Direct Sequence Spread Spectrum
FEC	Forward Error Correction
FHSS	Frequency Hopping Spread Spectrum
FPGA	Field Programmable Gate Array
FSK	Frequency-Shift Keying
GPIO	General Purpose Input Output
GPU	Graphic Processing Unit
I/O	Input-Output

IC	Integrated Circuit
IoT	Internet of Things
IQ	In-phase & Quadrature
LDPC	Low-Density Parity-Check
LLR	Log-Likelihood Ratio
LO	Local Oscillator
LUT	Lookup Table
MAC	Multiply-Accumulate
MCU	Microcontroller Unit
MPSoC	Multi-Processor System-on-Chip
NoC	Network-on-Chip
PCIE	Peripheral Component Interconnect-Express
PLL	Phase-Locked-Loop
PN	Pseudo-Noise
PSK	Phase-Shift Keying
RF	Radio Frequency
RSC	Recursive Systematic Convolutional
RTL	Register Transfer Level
SDR	Software-Defined Radio
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SRAM	Static Random-Access-Memory
UMTS	Universal Mobile Technology System
USB	Universal Serial Bus

CHAPTER 1

INTRODUCTION

1.1 Background and Motivation

Advancement in Integrated Circuit (IC) and wireless technology has enabled a new computing paradigm. Being tiny, inexpensive and powerful enough, computing devices can be deployed virtually anywhere in daily objects and environment. Paired with sensing and communication capabilities, these devices are working together to bring intelligence into daily objects and environment. This paradigm is better known as Internet of Things (IoT), where objects or environments are attached or embedded with smart nodes. A basic smart node should at least consist of processing unit, wireless module, power management system, and sensor. Low cost and ultra-low power are fundamental properties as these nodes will be deployed in large numbers and should last for years or decades on a small battery or harvested energy.

Contrary to personal computing devices that could perform tasks individually, IoT devices work together by wirelessly communicating data and coordinating operations. As the communication backbone among devices, the wireless system needs to satisfy a few main requirements which are related one to another. It must cover long range, operate in low power, and be robust to interference (Kuo & Kung, 2014).

A wireless system's range is defined by various factors such as operating frequency, receiver's sensitivity, and transmission power. Higher receiver sensitivity allows better data reception from weak signal. This leads to lower transmission power over same range or longer range for the same transmission power, a property desired by every radio standard especially IoT. One of several significant factors affecting a receiver's sensitivity is the modulation scheme. While lower Radio Frequency (RF) undoubtedly leads to longer propagation of signal, wideband and narrowband signal has been in constant comparison for their effectiveness in improving receiver sensitivity (Lassen, 2014).

Compared to other smart node's components such as processing unit and sensor which can easily operate in ultra-low power, low power operation of the wireless module yet remained a challenge (Burdett, Spring 2015). Wireless transceiver utilizes high power during transmission and reception operations, which are performed occasionally and swiftly in IoT application. IoT devices can exploit this operational characteristic by activating transmitter only when sending data is necessary, while intelligent protocol can be implemented in receiver to determine when it should receive. Such protocols are widely research, ranging from periodical receive (Dam & Langendoen, 2003) to channel sensing receive (Hu, Ma, & Sun, 2011), intending on maximizing wireless module sleep duration.

Wireless system's robustness against interference is commonly measured through its bit error rate. A low error rate wireless system provides better data integrity and requires less retransmission. This leads to longer sleep duration of the wireless module, thus saving power. Forward error correction

(FEC) code has been widely used to significantly reduce wireless system's error rate. It adds parity to data payload, providing the system with self-error correction capability at the expense of additional data overhead. This trade-off is completely acceptable in IoT application where low error rate is necessary, while low data rate is sufficient (Roth, Dore, Ros, & Berg, 2015). A particular FEC that has been extensively adapted into wireless standards due to its near-Shannon limit correction capability is Turbo code (Berrou, Glavieux, & Thitimajshima, 1993). Being relatively computational expensive, numbers of works has been done to simplify Turbo code (Lin, et al., 2006; Li, et al., 2013), allowing implementation on low power Application-Specific Integrated Circuit (ASIC).

The rapid growth and interest in IoT has pushed the development and improvement of wireless standards to satisfy the main requirements of IoT's wireless system described above. Re-configurability of a wireless module is the key to accommodating these varying and evolving standards without the need of hardware change (Jondral, Wiesle, & Machauer, 2000). Software-defined radio (SDR) has been proposed as a solution for flexible signal processing (Becker, Luk, & Cheung, 2009), deployed in diverse processing unit such as Digital Signal Processor (DSP), Field-Programmable Gate Array (FPGA), general purpose Central Processing Unit (CPU), Graphic Processing Unit (GPU), and programmable Application Specific Integrated Circuit (ASIC).

ASIC-based implementation of a wireless module has always been appealing due to its superiority in power efficiency, affordability, and size (Jalier, et al., 2010). These characteristics are highly desirable in every wireless

system, especially IoT. Implementations are usually realized in a fixed, single-purpose System-on-Chip (SoC) which trades flexibility for power efficiency. However, recent advancement in semiconductor technology has allowed programmable-type SoC to pack much more processing capability while still maintaining its low cost, low power, and small size characteristic. Texas Instruments' Keystone II SoC is an industrial-level example for such system (Instruments, 2012). This enables programmable ASIC as a compelling platform for SDR implementation which provides reconfigurable radio system while still maintaining all of ASIC benefits, such as SDR-based ZigBee protocol implementation (Ghazi, Boutellier, Hannuksela, Silvén, & Janhunen, 2013).

While various SDR platforms are widely available, the design and implementation of an SDR-based wireless system particularly for IoT application poses many challenges. Consisting of at least a radio front-end and a digital baseband processor, SDR-based wireless system highly relies on the later part. Proper understanding of the digital processor's architecture is required during implementation and optimization to achieve the necessary performance while still maintaining low-power operation. Choice of radio front-end hardware also requires careful assessment, as its capability particularly in power management is inherently an inseparable factor for enabling low-power communication standards and protocols stack.

1.2 Research Goals and Approach

Driven by the need for a wireless system that is both programmable and adhering to IoT's wireless link's requirements described in the previous section,

a fully functioning SDR-based wireless transceiver system is proposed in this research. The SDR will be implemented in a low-power programmable Multi-Processor System on Chip (MPSoC) and paired with a programmable radio front-end. The SDR-based wireless transceiver system built in this research functions at the physical layer. It consists of two identical modules, one as transmitter and the other as receiver, communicating in simplex manner.

In this work, the Lime LMS6002D Reference Development Kit is chosen as the programmable radio front-end, handling analog signal operations both in baseband and Radio Frequency (RF) region. Functioning as the digital baseband processor of the wireless module is the RUMPS401, a four-core ARM M0-based MPSoC fully designed by a team from the UTAR's VLSI Research Center. The author is a member of the team.

As described previously, a low error rate system provides benefits of data integrity and low power operation due to less retransmission required. The Turbo coding hence will be implemented to provide lower error rate, along with IQ modulation/demodulation, pulse shaping, and receiver synchronization to provide a functional transceiver system working on a physical layer stack. Considering the RUMPS401 which naturally is a low-power MPSoC as the target hardware, the Turbo code and modulation/demodulation are relatively computational and resources expensive to implement. Complete understanding of the RUMPS401's architecture is required to fully exploit its capability and achieve optimized performance, while comprehension on the Turbo coding and modulation/demodulation are as important for proper implementation on the target hardware.

As the modulation scheme, particularly bandwidth, affects the range of a wireless system, initial study and simulation will be carried out for modulation schemes, focusing on wideband versus narrowband. Turbo code structure and its sub-optimal approaches are also studied and simulated. These studies and simulations serve as the base for determining parameters to be implemented in the actual wireless transceiver system. The RUMPS401's physical design process is a significant part of this work. Collaborative development and testing of the chip with the UTAR VLSI Research Center are also part of this work. These design and testing process play an important role towards deep understanding of the RUMPS401 architecture, which will serve as the base for the software implementation and optimization. While the RUMPS401 architecture and silicon process inherently provides the capability of low power operation, it must be paired by proper software that is programmed to fully utilize its functionality, especially in complex application such as SDR.

Albeit the optimization-oriented implementation, we are aware that due to its processing capability, the RUMPS401 would not be able to provide competitive performance in terms of speed. It however could provide reasonable performance, sufficient for the implementation of a fully functioning SDR-based transceiver system. Moreover, to our best knowledge there has not been any work done by utilizing SoC of processing power compatible to the RUMPS401 as a digital baseband processor. Comparison of implemented transceiver system against other works is therefore not a focus of this project. System performance however, is evaluated throughout the implementation and optimization process.

This research serves as a proof of concept for a flexible yet low-power wireless transceiver system by implementing SDR on a low-power MPSoC. Through proper comprehension and exploitation of its architecture, the RUMPS401 could function as a baseband processor with reasonable performance, even without any dedicated DSP hardware. Its usage also serves as an important step in UTAR VLSI Research Center's long-term effort in development of the RUMPS401, by putting the SoC in a practical yet complex application.

Objectives of this research are:

- To acquire complete ASIC design methodology skills, accentuating on "First-time-success" design flow and practice.
- To study the SDR concept and apply it to the physical layer stack of a wireless transceiver system.
- To demonstrate the capability of UTAR VLSI research center's MPSoC, the RUMPS401 in a critical, complex application with emphasis on parallel processing.
- To design and implement a fully-functional SDR-based wireless transceiver system on a low-power programmable MPSoC, structured and adapted for low power IoT applications.

1.3 Dissertation Outline

This dissertation consists of seven chapters. The first chapter introduces the background and motivation of the research, along with its goals and

objectives. The second chapter reviews related past works and details about the hardware used in this work. Initial study and simulation of modulation schemes as well as Turbo coding are presented in Chapter Three. The fourth chapter presents and analyzes optimization process of Turbo decoding on the RUMPS401. Chapter Five describes the implementation of modulation/demodulation part of the transceiver system on target hardware. Testing and analysis of the full transceiver system is provided in Chapter Six. Finally, Chapter Seven concludes the project work and write-up.

CHAPTER 2

LITERATURE REVIEW

In line with the research background and the proposed system described previously, this chapter reviews some past works on Software-Defined-Radio (SDR) transceiver and Turbo coding implementation. The review on SDR implementations emphasizes on the widely varying hardware platforms particularly the digital processor's architecture, while the review on Turbo coding implementations centers around works which hardware platforms are Application-Specific Integrated Circuit (ASIC) based. General Turbo coding structure and algorithm will also be discussed for a better and more coherent understanding of the fundamentals of the optimization process. Lastly, as the hardware used in this research, the RUMPS401 chip and Lime LMS6002D Reference Development Kit will be described in detail.

2.1 SDR Implementation

Digital signal processing technique represents real world analog signals as digital sampled data, then uses digital processors to analyze and process information from them (Ifeachor & Jervis, 2002). It has been widely used in various applications due to its attractive advantages of affordability and programmability. As described by (Ifeachor & Jervis, 2002), digital signal processing benefits from its reliance on digital domain processing and Integrated Circuits (IC) which are reprogrammable, consistent, low cost, low power, tiny, and fast. A significant amount of effort has been put into

researching SDR techniques which are implemented on various hardware platforms. While the choice of processor types varies widely, a common trend is the adoption of parallel processing schemes using multi-processor platforms.

KUAR (Minden, et al., 2007) is a complete SDR platform, consisting of digital baseband processors and a radio front-end. Employing a multi-processor system for digital baseband processing, it consists of a 1.4GHz Pentium-M Embedded PC as Control Processor Host (CPH) and a Xilinx Virtex II Pro P30 Field-Programmable Gate Array (FPGA), interconnected through Peripheral Component Interconnect-Express (PCIe) channels. The Xilinx FPGA possesses programmable logic gates and two PowerPC 405 processors. KUAR's architecture provides design flexibility where communication system in most common cases can be deployed entirely in hardware or software, or a hybrid of both. Full hardware implementation can be done through the FPGA's logic gates, while full software implementation is deployed in the CPH as it is powerful enough for significant signal processing. Hybrid implementation utilizes both CPH and FPGA, along with the two PowerPC 405 processors embedded in the FPGA.

FAUST is a Multi-Processor System-on-Chip (MPSoC) designed as a high-performance baseband processor (Lattard, et al., 2008). It consists various task-specific signal-processing hardware accelerators hardwired to a host Central Processing Unit (CPU) subsystem with the ARM946ES processor core, shown in Figure 2.1. Connections are achieved through their own high-performance Network-on-Chip (NoC) interconnection, designed and optimized for telecom baseband application.

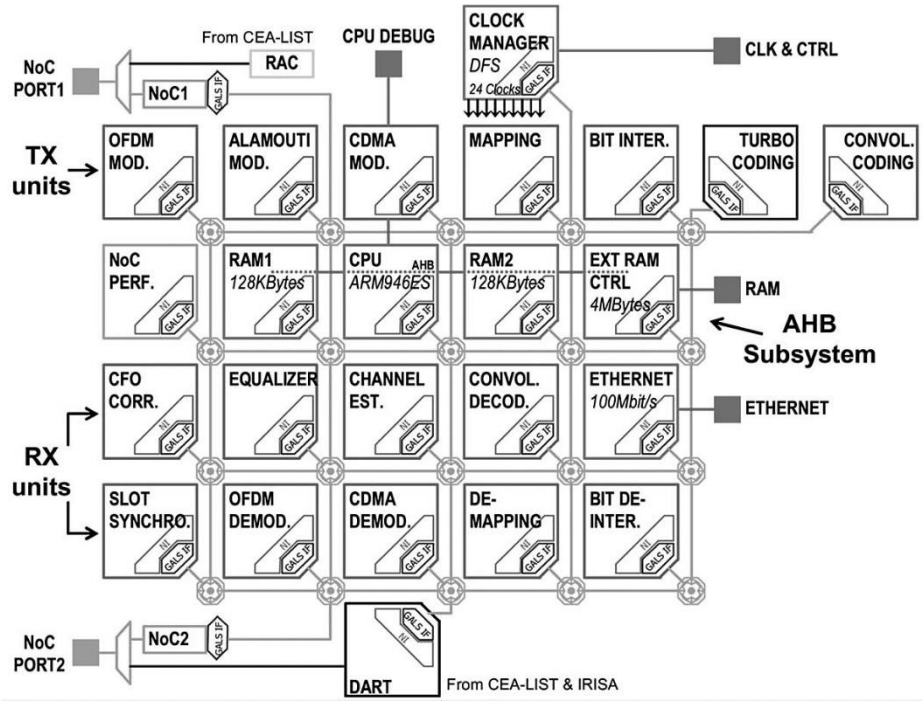


Figure 2.1 FAUST Chip Diagram (Lattard, et al., 2008)

Lattard et al. (2008) highlighted the importance of NoC as the backbone of the MPSoC architecture, especially in a high data rate application such as baseband processing. FAUST implements architecture where data flows between functional blocks are administered in distributive manner, leaving the host CPU with only pipeline control task. This platform is reconfigurable by changing data path, in which some functional blocks may or may not be used depending on the implemented radio protocol. MAGALI (Clermidy, Lemaire, Popon, Kténas, & Thonnart, 2009) is designed on a similar architecture, with the addition of a Communication and Configuration Controller (CCC) on each NoC's network interface, to dynamically accept and reconfigure functional blocks according to the host CPU's instruction. Figure 2.2 illustrates MAGALI's architecture of mesh interconnected components.

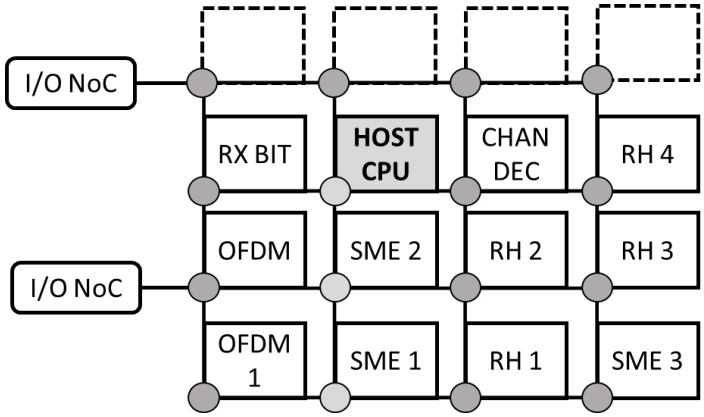


Figure 2.2 MAGALI Mesh-connected Baseband MPSoC (Clermidy, Lemaire, Popon, Kténas, & Thonnart, 2009)

While the implementations mentioned above try to decentralize control through the design of data processing flow, their own heterogeneous architecture particularly the existence of the host CPU itself poses inherent central control characteristic, thus introducing bottlenecks. A homogeneous MPSoC for baseband application, GENEPY (Jalier, Lattard, Jerraya, Sassatelli, Benoit, & Torres, 2010) is built based on multiple instances of a Smart ModEm Processor (SMEP) unit connected by an NoC. The basic SMEP unit consists of a processing cluster of two Digital Signal Processors (DSPs), a memory unit called Smart Memory Engine (SME), an NoC interface, and a CCC. By replacing hardware processing blocks with programmable DSPs, GENEPY provides more flexibility during implementation.

Analyzing performance between heterogeneous and homogeneous MPSoC platforms, Jalier et al. (2010) presented two versions of GENEPY platform namely v0 and v1, and compared them against the MAGALI platform (Clermidy, Lemaire, Popon, Kténas, & Thonnart, 2009). Compared to the MAGALI which utilizes various functional blocks for computations and a

global controlling host processor, GENEPY v0 employs multiple SMEP units for computing and a host processor to assume global control task, which is shown in Figure 2.3. GENEPY v1 implements a fully homogeneous structure based on multiple SMEPv1 units with no host processor as shown in Figure 2.4, allowing equally distributed control among all units thus improving flexibility and scalability. SMEPv1 is a basic SMEP unit with a 32-bit MIPS processor added, allowing individual control management on each unit. Comparison of these platforms for LTE application by Jalier et al. resulted in performance speed-up of 3%, silicon area reduction of 14%, and power saving of 18% in favor of the homogeneous platform GENEPY. Two versions of GENEPY are comparable with GENEPY v0 achieving smaller silicon area of 3% and GENEPY v1 consumes 8% less power.

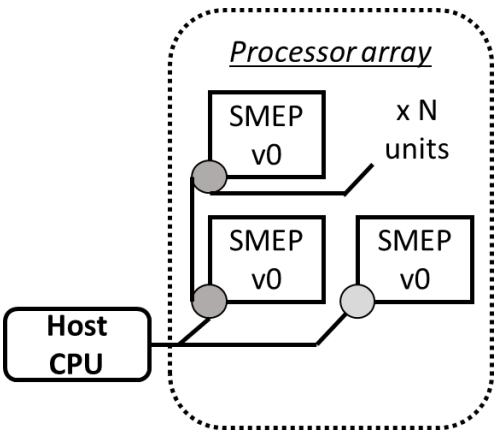


Figure 2.3 GENEPY v0 Architecture (Jalier, et al., 2010)

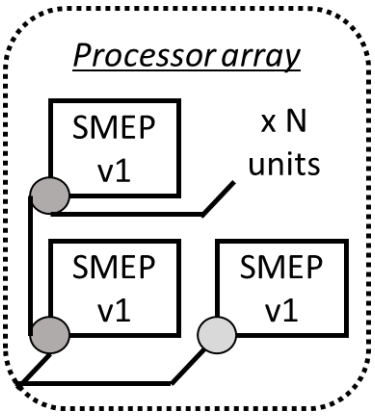


Figure 2.4 GENEPY v1 Architecture (Jalier, et al., 2010)

GNU Radio (About GNU Radio) is an open source software library and toolkit for digital signal processing aimed for SDR implementation on General-Purpose Processor (GPP) such as Intel x86, running mostly on Linux system. While GNU Radio with complementary radio front-end provides the daily computer the capability as a flexible radio, implementation on ASIC is still very desirable due to its size, efficiency, and power consumption. Ma, Marojevic, Balister, & Reed, (2014) proposed porting GNU Radio to run on ARM-based processor, with Texas Instrument Keystone II SoC as target hardware. Ma et al. set a long-term goal of a complete SDR software platform that would map functional block defined in GNU Radio library to any target hardware's native instructions in an optimized manner. This long-term objective will enable the harnessing of matured and widely-used GNU Radio environment on MPSoC-based SDR implementation.

2.2 Turbo Coding

Turbo code was first introduced as a new type of error correction scheme based on convolutional code (Berrou, Glavieux, & Thitimajshima, Near

Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes (1), 1993). It concatenates in parallel two Recursive Systematic Convolutional (RSC) encoders. A binary rate RSC code has one systematic output equals to the input bit, and one parity bit as a function of the feedback loop. A Turbo code encoder consists of two RSC encoders, where the first encoder generates one systematic bit along with one or more parity bits from each input bit, while the second encoder accepts interleaved version of the input data and outputs the same number of parity bits, as shown in Figure 2.5. The output of the Turbo encoder for each input bit are the input bit itself, accompanied by one or more parity bits according to the encoder structure. The interleaving simply permutates the sequence of input bits pseudo-randomly.

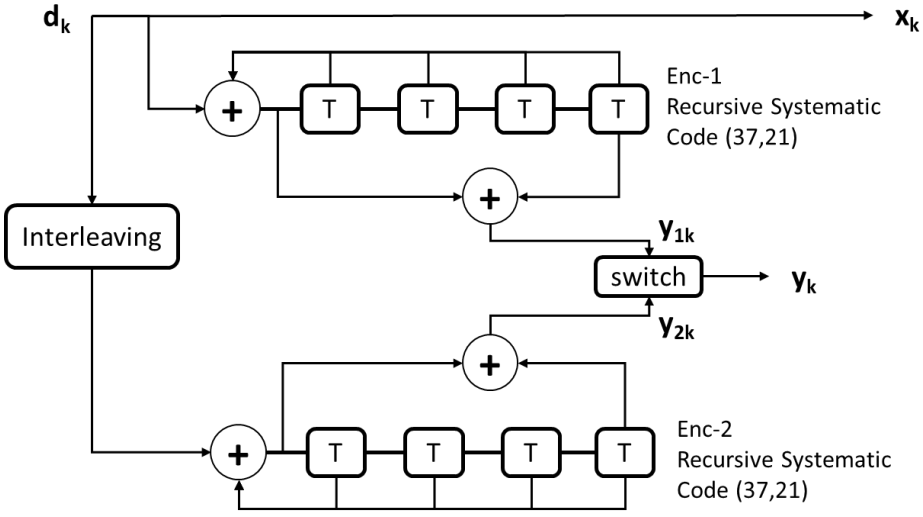


Figure 2.5 Turbo Encoder Architecture (Berrou, Glavieux, & Thitimajshima, 1993)

On the decoding side, Turbo code employs two RSC decoders both running the BCJR algorithm (Bahl, Cocke, Jeinek, & Raviv, 1974) chained in serial, with a feedback loop from the second decoder to the first decoder as

depicted in Figure 2.6. Berrou et al. (1993) proposed the feedback loop to fully expose redundant information produced by the two encoders, to both decoders. Each decoder takes the systematic bit and the corresponding parity bits generated by its encoder counterpart, i.e. first decoder takes parity bits generated by the first encoder and so does the second decoder, which then calculates the Log-Likelihood Ratio (LLR) of the incoming bits value.

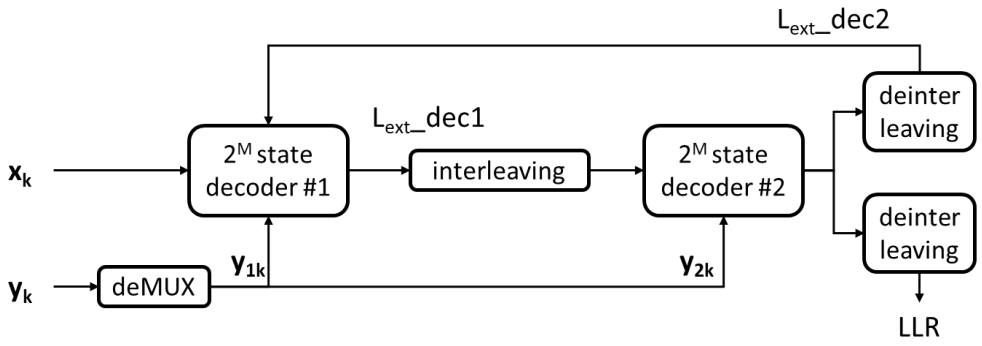


Figure 2.6 Turbo Decoder Architecture (Berrou, Glavieux, & Thitimajshima, 1993)

The LLR is exchanged between decoders as a priori LLR, to provide other decoder with additional parity information only available to the providing decoder, i.e. the second decoder gets parity information of the first encoder through this LLR. A single iteration Turbo decoding requires the input bits to be run through both decoders, where the LLR values from the second decoder are then used for deciding whether the originally sent bits were one or zero. The decoding is usually run in multiple iterations where the second decoder's LLR values must be fed back to the first decoder to be used in the next iteration, or used for bit decision on the last iteration.

2.3 Turbo Coding Implementation on ASIC

While providing a near-Shannon coding capacity limit, Turbo code imposes a relatively high computing resource requirement on its decoder, which primarily stems from the BCJR algorithm used in each decoder coupled with the iterative decoding process. In attempts to bring the BCJR algorithm into practical applications, there are a few important works on simplifying the original BCJR algorithm. The first work (Robertson, Hoeher, & Villebrun, 1997) simplifies the BCJR algorithm by approximating its calculations in logarithmic domain, which translates multiplications into additions hence simplifying the computational load by a significant amount in expense of error rate. This approximation yields a suboptimal version of the original BCJR algorithm, widely known as Max-Log BCJR algorithm. Another crucial work is the introduction of sliding-window BCJR decoding (Marandian, Fridman, Zvonar, & Salehi, 2001), which divides the received data frame into smaller windows and decodes them separately thus significantly reducing the memory requirements. Most works on Turbo decoder implementation on real hardware are based on these two concepts of simplification.

Some works implement Turbo decoder on powerful and power-hungry hardware such as DSP, CPU (Huang, et al., 2011), GPU (Liu, Bie, Chen, & Jiao, 2013) due to the highly programmable property they possess. However, such implementation is not suitable for Internet of Things (IoT) nodes where cost and power consumption are critical. Other works attempt to satisfy both cost and power consumption aspect by sacrificing programmability through the design

of ASIC hardware decoder blocks, where the trade-off between its efficiency and programmability are varying across implementations.

Condo, Martina, & Masera (2012) proposed an NoC-based Low-Density Parity-Check (LDPC) / Turbo decoder block, offering an efficient yet rather flexible decoder block that could accommodate both LDPC and Turbo decoding on a single platform. The work focuses on the design of the NoC as the backbone between parallel processing elements, done by extensive studies and simulations of various network topologies and complex routing algorithm.

Another work proposes a low complexity Turbo decoding block intended for wireless sensors (Li, Maunder, Al-Hashimi, & Hanzo, 2013) which exploits the simplification of the BCJR algorithm. The logarithmic domain approximation of the BCJR algorithm allows the block to be mostly composed of addition rather than multiplication circuit, which cuts down the number of logic gates required significantly. Meanwhile, the sliding window BCJR concept is adapted into parallel processing by pipelining different windows on different stage of the BCJR algorithm's calculation, whose intra-processing elements communications relies on an NoC. As shown by the work and majority of ASIC-based Turbo decoder implementation (Li, et al., 2016; Ahmed, Awais, A. u. Rehman, Maurizio, & Masera, 2011), highly efficient and tiny hardware blocks are the results by trading off its programmability.

2.4 The RUMPS401

The Rahman University Multi-Processor System 401 (RUMPS401) is a four-core MPSoC designed by the UTAR's VLSI Research Center, which basic architecture is based on an MPSoC Register Transfer Level (RTL) modeling platform by (Hartono, 2014) and the corresponding verification platform (Lim, 2015). The UTAR Adaptive NoC (Lokananta, 2015) provides reliable connections among the cores. Each core is an ARM Cortex-M0, a small footprint and highly power efficient ARM processor (ARM, 2009), connected to various locally available peripherals through the ARM's AHB Lite bus. Peripherals include flash memory, Static Random-Access Memory (SRAM), General Purpose Input Output (GPIO), timer, and others as depicted by the RUMPS401 architecture in Figure 2.7.

Emphasizing on low power design, the RUMPS401 employs an on-chip power management system that allows each core to be put to sleep independently and waken up through an NoC or GPIO interrupt. The quad-core MPSoC normally runs on an external 16MHz clock. When all the cores are put to sleep it ticks over on a locally generated 32kHz clock. The chip's power consumption is 32mA in normal operation and 13uA in sleep mode. The clock source transition is managed by clock gating in the power management module.

Each core in the RUMPS401 has identical flash memory of 32kB and SRAM of 8kB for program storage and execution. This allows each core to be programmed and run independently, which along with the locally available GPIO will be able to function as a single microcontroller. Multi-processor usage

scenario can be easily accommodated through the NoC connection among the cores. As can be seen from Figure 2.7, each core possesses an identical set of peripherals and another peripheral specifically available only to each of them. While the specific peripheral attached to each core defines the core's intended functionality, this does not limit the usage of that core to any function. Due to the fact that most of peripherals are identical across all cores, a general program can be run identically on all cores.

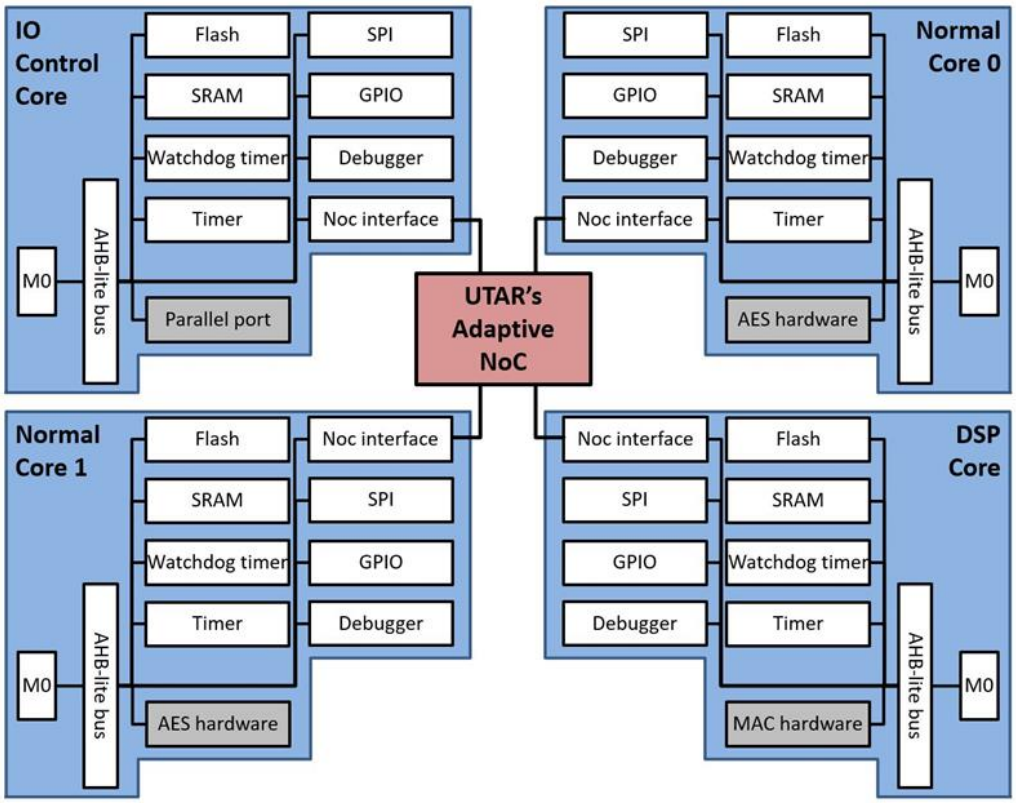


Figure 2.7 RUMPS401 Architecture

The IO Control Core is designed to handle all communication between the RUMPS401 and any external devices, thus it is equipped with more GPIO pins and a parallel port. It also functions as the bootloader for the chip, hence has its flash memory divided into two program code partitions, one for its own

application code and another for the bootloader code. Application code for all cores are sent to the IO Control Core, which then are loaded to respective cores by the bootloader through an internal NoC bootload mechanism. Both the Normal Cores and the DSP Core have their flash memory fully usable for their own partition code and have less GPIO pins. They are however equipped with task-specific hardware accelerator thus are more suitable for computational function. The Normal Cores come with the 128-bit Advance Encryption Standard (AES) block and the DSP Core with the Multiply-Accumulate (MAC) block.

2.5 Lime LMS6002D Reference Development Kit

The Lime LMS6002D Reference Development Kit is a ready-to-use RF transceiver board based on the Lime LMS6002D transceiver chip. It is a fully-integrated, multi-band, multi-standard, programmable RF transceiver packed into a single IC solution (Microsystem, 2012). This transceiver covers the 0.3-3.8GHz frequency band and the 1.5-28MHz modulation bandwidth. The built-in Analog-digital-converter (ADC) and Digital-analog-converter (DAC) which sampling rates are programmable, provides digital In-phase & Quadrature (IQ) interface to the baseband region. This allows full control over the modulation scheme to be implemented in the RF application.

Operating on either 1.8V or 3.3V power supply, the Lime LMS6002D is programmed by writing into its registers through the Serial Peripheral Interface (SPI). This enables on-the-fly adjustment, and most importantly

allows the transceiver to be turned-off when not in use and re-programmed right after power up.

2.6 Summary

While the past works on SDR implementation were done on widely varying hardware platforms, they all possess one common key feature, the use of multi-processor system along with the need for powerful and reliable connection among the processors. The multi-processor system is crucial as parallel processing that significantly improves performance, while the NoC is important in providing robust inter-core communications. This feature is also adopted in the works on ASIC-based Turbo code implementations, which stress on the design of highly parallel processing elements and the NoC that interconnects them. Parallelism in Turbo code implementation must be coupled with the simplification of the algorithm to allow practical deployment in ASIC for improved performance. Implementation on the RUMPS401 will be based on these key concepts.

The past works of SDR implementation also highlighted the key trade-off between flexibility and efficiency of the system. The more flexible approaches utilize highly programmable and powerful digital processor, where most of the transceiver functionality are implemented in software. On the other hand, an SDR system can be implemented using less powerful digital processors that are attached with specialized hardware accelerators. Configurability is the sacrifice. However, this approach offers more efficiency in which to achieve the same level of performance, the specialized hardware requires less logic gates

and can be operated at lower clock speeds, thus consuming less power compared to more powerful and flexible digital processors.

As described in the previous section, the RUMPS401 is based on four ARM Cortex-M0 which is the smallest and the least complex processor in the ARM Cortex's lineup. The M0 is intended for use in embedded application where complex calculations are not expected, thus it has no native hardware for multiplication or division, let alone floating point or complex signal processing block. In terms of peripherals, the DSP core is the most suitable for rather complex calculation due to its single-cycle Multiplier-Accumulator (MAC) hardware. In later chapters, it will be shown how the MAC significantly improved the RUMPS401's performance for baseband processing. However, it should be noted that the existence of MAC hardware would not bring the RUMPS401 to a performance on par with any former works. A common DSP accelerator provides functionalities such as multiplication, division, exponential, logarithm, trigonometry, which are performed at high speed. While these functionalities can be approximated by MAC function, it can be inferred that these approximations would certainly take longer time compared to the common DSP accelerator.

Being programmable yet low-power, the RUMPS401 is an interesting but challenging platform for SDR implementation. To our best knowledge, no work has been done thus far to implement SDR on such hardware. It will be shown in later chapters that a flexible yet low-power SDR implementation is feasible, particularly in IoT applications where data rate is less important than power consumption.

CHAPTER 3

INITIAL STUDY OF MODULATION SCHEMES AND TURBO CODE

As described previously in the first chapter, the SDR-based wireless transceiver system built in this work consists of two parts. The first part is the implementation of Turbo code, while the second part is the implementation of IQ modulation/demodulation, pulse shaping, and receiver synchronization which provides the functionality of the basic physical layer stack. Both parts will be fully implemented in software which is developed specifically for the RUMPS401 as the digital baseband processor.

As discussed in Chapter 2, the more flexible SDR implementation generally employs most if not all of its radio functionality via software. While this kind of implementation provides broader choice of implementable radio system over a single hardware platform, it lacks the efficiency that the specialized hardware block offers. However, the lack of efficiency can be minimized by optimizing the software specifically for the target processor as shown by the past works. This is especially the case for the RUMPS401 due to its naturally low-performance hardware for ultimate power saving.

The software optimization for SDR application requires proper understanding of the processor's architecture and the wireless system. In-depth understanding of the processor's architecture is necessary as the software must be able to fully utilize the hardware resources and must be coded as close as possible to the metal. Understanding of the RUMPS401's architecture is given

because its design, development, and testing are parts of this research work. It will be shown in the following chapters that the SDR software is developed by capitalizing fully on RUMPS401's hardware resources. On the other hand, understanding of the wireless system is as important to allow the breakdown of various signal processing functions into smaller and simpler tasks which can be performed in parallel. This enables the full exploitation of the RUMPS401's multi-processor system.

Preceding the actual development of the SDR software for the RUMPS401, an initial reference study and simulation regarding wireless system was carried out, which includes modulation scheme and Turbo code. This initial work provides the basis for deciding what wireless system to implement, along with its parameters. The simulation model also plays an important role for developing the functional-level of the actual SDR software. It is built in the Matlab's matured signal processing environment, which allows easier and faster development of the functional-level software. This initial study and high-level modeling are very important as the actual SDR software will deploy specific wireless system, which is coded precisely for the RUMPS401.

This chapter covers the initial reference study and simulation performed in this work, which is organized into four parts. The first part discusses the reference study on various digital modulation method, while the second part provides the comparison of narrowband and wideband wireless system based on reference study and simulation model. Turbo coding simulation as well as the results are discussed in the third part of this chapter. The last part summarizes the initial study and simulation process.

3.1 Digital Modulation

Modulation refers to the process of embedding information signal into much higher frequency carrier signal, to propagate the information wirelessly (Sklar, 2001). The modulation process serves two main purposes, which are to allow multiple transmissions and to allow the use of smaller antenna. When viewed in frequency domain the modulated signal occupies a spectrum with width equal to the information signal's frequency and centered at the carrier's frequency. This allows multiple signals to be transmitted at the same time over varying frequency range. Wireless transmission of an electromagnetic wave depends on the use of antenna designed to properly launch and receive the wave, allowing it to propagate through longer distance. The basic equation of $c = \lambda f$, where f is the frequency, λ is the wavelength, and c is the speed of light defines the inverse relation between a signal's frequency and wavelength i.e. the lower the frequency, the longer the wavelength and hence a larger antenna required to properly transmit and receive the signal. For example, a 10kHz information signal has a wavelength of around 30km, assuming the antenna size should be at least half of the wavelength, a 15km long antenna must be used to properly transmit this signal. Meanwhile, a 500MHz carrier signal only requires 30cm long antenna for proper transmission. 'Piggy-backing' the information signal to the higher frequency carrier allows the transmission to be done with drastically shorter antenna size.

Digital modulation is the technique of embedding digital symbols, i.e. bits into the carrier signal. Sequences of digital bits are used to transform the carrier signal by means of manipulating the signal's amplitude, frequency,

phase, or a combination of them (Sklar, 2001). Amplitude-Shift-Keying (ASK) modulation alter carrier signal's amplitude according to the input bits which results in non-constant envelope signal. On the other hand, Frequency-Shift-Keying (FSK) modulation uses input bits to modify the carrier signal's frequency accordingly, yielding modulated signal which has constant envelope. Similarly, Phase-Shift-Keying (PSK) modulation yields constant envelope by altering the carrier signal's phase. Figure 3.1 depicts the three most common digital modulation methods.

As in analog modulation, the non-constant envelope modulation such as ASK is more prone to noise interference compared to the constant-envelope modulation such as PSK and FSK. If implementation simplicity is not an issue ASK loses out in error rate performance. PSK and FSK are equal in terms of error rate and spectral efficiency. PSK provides lower error rate for the same Signal-to-Noise Ratio (SNR) while yielding lower spectral efficiency compared to FSK (Mulally & Lefevre, 1991). Applications that emphasizes on low error rate would adapt PSK modulation, while FSK is more preferred when spectrum availability is scarce. As this research attempts to reduce the error rate through the implementation of Turbo Code, PSK is a more suitable choice compared to FSK.

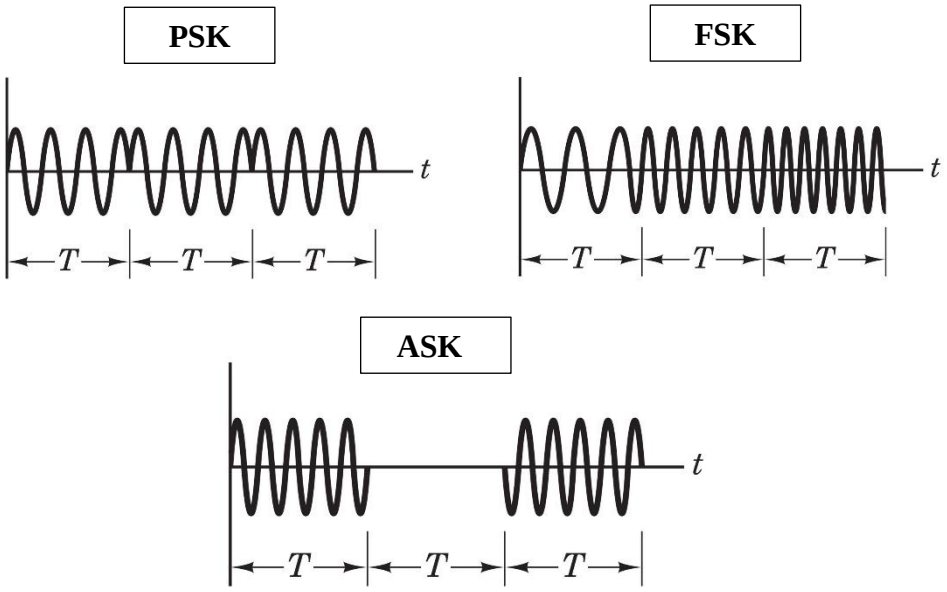


Figure 3.1 Basic Digital Modulation Methods (Sklar, 2001)

PSK itself can be further classified based on the demodulation method and the number of constellation points. Coherent receiver demodulates the received signal (Haykin, 2001) by utilizing its carrier's phase information, while non-coherent receiver does not use this information. Coherent demodulation attempts to reproduce the same carrier frequency generated on the transmitter's local oscillator. The reproduced carrier's frequency is then used by the receiver to demodulate the incoming signal, converting it down to the baseband frequency. This leads to lower error rate in expense of more complex demodulation process compared to the non-coherent demodulation. Meanwhile, higher number of constellation points results in higher spectral efficiency, i.e. Quadrature PSK (QPSK) provides twice the data rate compared to Binary PSK (BPSK) for the same bandwidth by utilizing both I and Q channel (Sklar, 2001). QPSK however also result in two times worse symbol error rate compared to BPSK. Figure 3.2 depicts the constellation diagram for QPSK modulation. It is shown that the whole IQ plane is divided into four decision regions, one for

each of the QPSK symbols. On higher MPSK modulation there are more constellation points which are spaced closer, narrowing the decision regions thus reducing the acceptable margin of error during demodulation process.

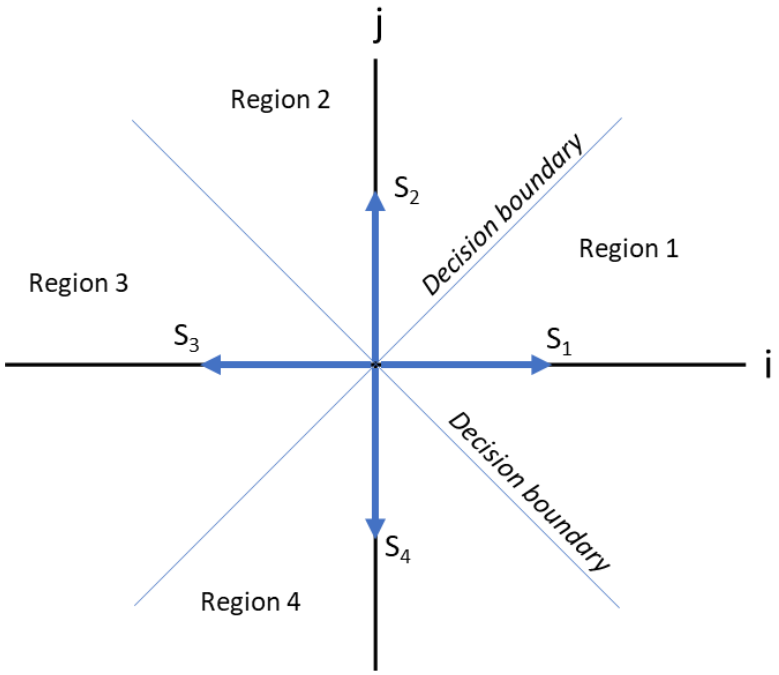


Figure 3.2 QPSK Constellation Diagram and Decision Boundary (Sklar, 2001)

3.2 Narrowband and Spread Spectrum

Spread spectrum modulation was first introduced to provide means of security for military's wireless communication (Haykin, 2001). By spreading the transmitted signal over larger frequency band, spread spectrum provides resistance to signal jamming and interception by unwanted parties. The two most widely used spread spectrum technique, Direct Sequence Spread Spectrum (DSSS) and Frequency Hopping Spread Spectrum (FHSS) provide those resistances through different methods.

In DSSS modulation, the data sequence is multiplied by a pseudorandom sequence of higher data rate, resulting in a sequence containing multiple copies of every symbol, whose data rate matches the pseudorandom sequence's. These copies of symbols are commonly referred as chips. Additionally, the resulting sequence is transmitted with lower transmission power so that the signal presence is difficult to be detected by unwanted parties. The data rate ratio between the pseudorandom sequence and original data sequence is commonly addressed as Spreading Factor (SF), which defines how many times the bandwidth expands.

As shown in Figure 3.3, the original sequence is a narrowband signal with high power, which is transformed by DSSS modulation into a low power wideband signal. The DSSS signal is spread across wider frequency band thus is harder to jam since jammer normally does not operate over wide frequency band. Moreover, as mentioned earlier the DSSS signal has low power level hence is hard to detect. On the other hand, FHSS modulation spread the original signal over wider frequency band by hopping around several narrow frequency band during a transmission. Should unwanted parties try to intercept or jam the transmission, they must know every narrow frequency bands a single transmission uses, which is not easy. FHSS provides security and robustness through this very frequency switching mechanism.

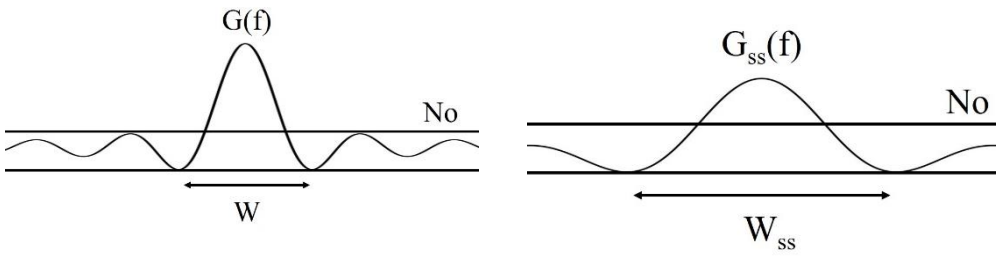


Figure 3.3 DSSS Signal Spectrum (Sklar, 2001)

There are two characteristics of DSSS which is of interest in Internet of Things (IoT) applications, that is the low power transmission and the redundancy from multiple copies of every bit in the original data sequence. In this work, the effect of DSSS modulation system on error rate will be evaluated through literature study and high-level model simulation built on Matlab platform. In practice DSSS is commonly implemented on the digital part of the data, in combination with any digital modulation technique such as PSK or FSK. Since BPSK provides the lowest error rate, a high-level model Matlab simulation of a simple BPSK-DSSS was performed and compared to a simple BPSK system. The simulation is performed to evaluate the receiver's error rate for transmission over an Additive White Gaussian Noise (AWGN) channel on varying SNR value. Figure 3.4 depicts the BPSK-DSSS simulation system model, with Pseudo-Noise (PN) algorithm generating the spreading sequence.

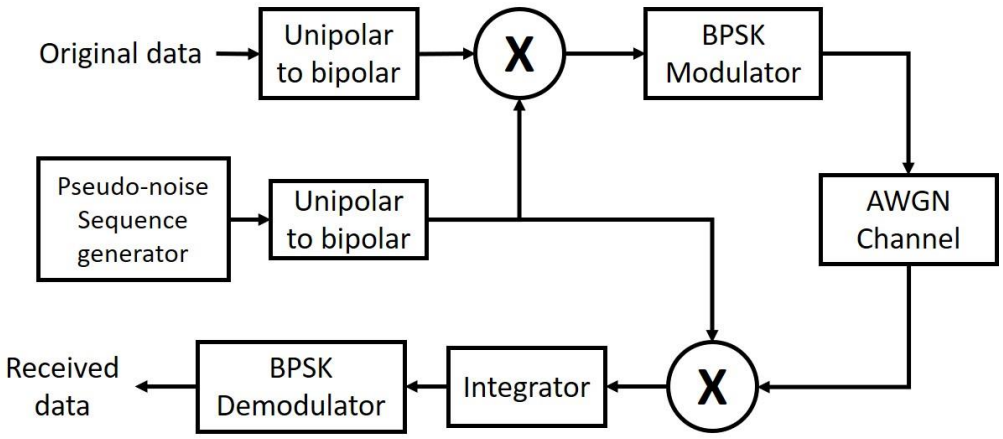


Figure 3.4 DSSS System Model

The simulation result in Figure 3.5 shows that DSSS does not provide improvement to the error rate of a BPSK system during transmission over AWGN channel, regardless of the spreading sequence's data rate. On a narrowband modulation each symbol is transmitted with power of Eb which stands for energy per bit, while on DSSS modulation each chip is transmitted with lower power of Eb/SF. During transmission over AWGN channel whose noise power is constant over whole frequency band, the receiver of narrowband system receives symbols whose individual SNR is Eb/No, where No denotes the noise power. The more positive the SNR value gets, the better is the signal's energy ratio against the noise energy, which leads to better reception and lower error rate. On DSSS modulation the receiver receives chips whose individual SNR is $Eb/SF/No$, which then accumulates chips that belong to one symbol, yielding the same individual symbol's SNR of Eb/No. It is mathematically obvious that by having the same SNR, both narrowband and DSSS system will have the same bit error rate (BER) performance (Pickholtz, Schilling, & Milstein, 1982; Wang, Yang, & Ying, 2009).

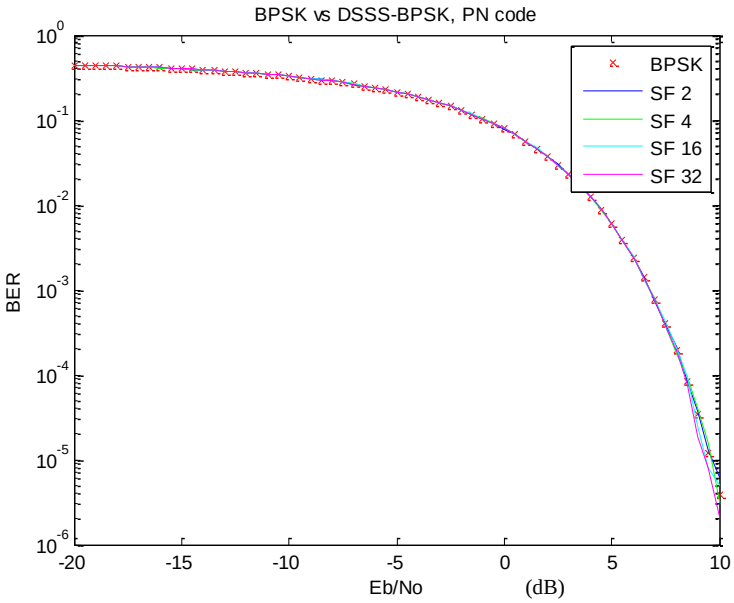


Figure 3.5 Narrowband vs DSSS Simulation over AWGN

Based on the reference study and simulation result, it can be deduced that DSSS system does not provide improvement for a transmission over AWGN channel, while adding extra complexity to the system through its spreading process. Moreover, based on receiver's sensitivity equation, larger bandwidth results in lower receiver sensitivity (Lassen, 2014). A high sensitivity receiver is very desirable in every wireless system especially IoT, to allow the system to transmit with lowest possible power over longest possible distance.

3.3 Turbo Code Structure

To provide a reasonable trade-off between error rate and complexity for the actual implementation on the RUMPS401, Turbo coding in relation to the structure of the code used, the decoding process, and the parameters trade-off, must be well understood. As in the modulation/demodulation aspects of the

wireless system, the high-level model of Turbo code's encoder and decoder is developed and simulated over various structure and parameters. The whole simulation process yields a high-level Turbo encoder and decoder model with fixed system parameters, which would be developed into an embedded software that is coded and optimized specifically for the RUMPS401. The rest of this section is arranged into four parts as follows. The first part describes the Bahl-Cocke-Jeinek-Raviv (BCJR) algorithm (Bahl, Cocke, Jeinek, & Raviv, 1974), which is the main element of Turbo decoder. The second part discusses about sliding window decoder, a memory-efficient improvement to the BCJR algorithm. The third part describes the Turbo Code's standard being adopted in this work. Finally, the last part presents in details the Turbo Code's simulation setup and results.

3.3.1 BCJR Decoding Algorithm

Named after its inventors, the BCJR algorithm was proposed as an improvement over Viterbi decoding algorithm (Viterbi, 1967). Both Viterbi and BCJR decoding algorithm are based on maximum-likelihood estimation (Haykin, 2001) that attempts to find the original transmitted data from the noisy received data, with a goal that the prediction made has the highest probability of being correct. The process of finding the original data relies on extra information provided by the parity bits generated by the encoder. While Viterbi algorithm attempts to find the most likely transmitted bits sequence, BCJR algorithm tries to find the most likely transmitted bits individually. In a more particular way, given a block of received bits, Viterbi algorithm guesses the sequence of bits inside the block as one inseparable entity, while BCJR

algorithm finds the content by guessing the value of each individual bit in the block. In terms of the BER, the BCJR algorithm can be considered to yield a more optimal decoding result as it attempts to minimize the error probability of each bit, whilst the Viterbi algorithm minimizes the block error probability. However, the BCJR algorithm is more complex compared to Viterbi algorithm.

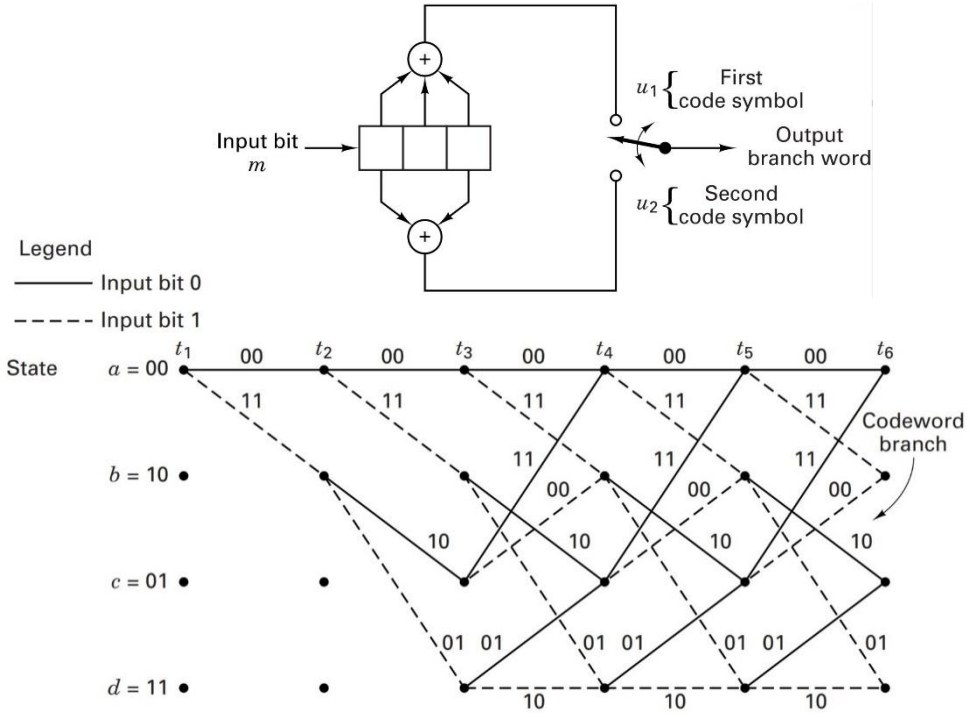


Figure 3.6 Sample Convolutional Encoder and Trellis Diagram (Sklar, 2001)

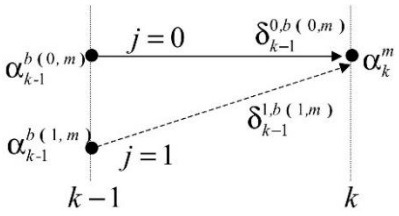
The operation of BCJR decoding algorithm on the received codeword can be best explained as a process of traversing the trellis diagram of the corresponding encoder. Figure 3.6 shows a simple convolutional encoder of rate 1/2 and depth $K = 3$. It has two memory elements which are implemented as shift registers. The encoder depth is the number of memory elements plus the input bit. The encoder takes one input bit at a time, producing two output bits through a defined modulo-two addition between the input bits and the registers

value. The registers values are bits from previous timestamp, i.e. the output codeword for third bit depends on value of the first and second bits. The operation of this encoder can be represented in its trellis diagram which provides the relation between the encoder's input, the values stored in the memory elements (commonly referred as the encoder's state), and the encoder's output. Each dot represents the encoder's state at any given time k, while the dots-connecting arrow represents the transition to the next state given a certain input bit. Solid arrow describes the transition for an input bit of 0, and dotted arrows for an input bit of 1. Number on the arrows defines the encoder's output for a transition. In practice the trellis diagram can be implemented as a simple lookup table.

During the traversal of the trellis diagram, BCJR algorithm constructs a complete trellis graph based on the encoder's basic trellis diagram and most importantly, the received codeword. It then amends three metrics to each pair of nodes and the arrow connecting them. Figure 3.7 illustrates the calculation of forward state, reverse state, and branch metrics along with the formulas, assuming an encoder of rate 1/2 and noise variance No. They are also commonly referred as α, β, and δ metrics, respectively. These metrics are essential parameters for calculating the original data bits likelihood of having value 0 or 1. Branch metric defines transition probability of each state to the next state, given the received codeword at time k. Forward state metric is the encoder's likelihood to be in a certain state at time k, knowing the forward and branch metrics of the previous states at time $k-1$ that transitioned to the current state. Reverse state metric is the opposite of forward state metric, defining the

encoder's likelihood to be in a certain state at time k, given the backward and branch metrics of the next states that the current state can transition to at time $k+1$.

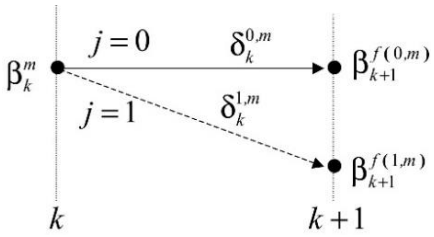
Based on the equation provided in Figure 3.7, branch metric calculation is defined as a correlation function between the received codeword at time k denoted by $x_k\, y_k$, and the expected encoder's output $u_k\, v_k$ for a certain transition between states. π_k^idenotes the a priori probability of the unencoded bit at time k for having value i. Forward state metric is the sum of all transition probabilities to the current state. α_k^m denotes the forward state metric for current state m at time k, while $\delta_{k-1}^{0,b(0,m)}$ is the branch metric corresponding to the transition by bit 0 from the previous state $\alpha_k^{b(0,m)}$ to the current state. Reverse state metric equation is expressed similarly, except that it is a summation of all transition probabilities from the current state.



$$\alpha_k^m = \alpha_{k-1}^{b(0,m)}\ \delta_{k-1}^{0,b(0,m)} + \alpha_{k-1}^{b(1,m)}\ \delta_{k-1}^{1,b(1,m)}$$

Where $b(j, m)$ is the state going backward in time corresponding to an input j

a) Forward State Metric



$$\beta_k^m = \beta_{k+1}^{f(0,m)}\,\delta_k^{0,m} + \beta_{k+1}^{f(1,m)}\,\delta_k^{1,m}$$

Where $f(j, m)$ is the next state given an input j and state m

b) Reverse State Metric

c) Branch Metric

$$\delta_k^{i,m} = \pi_k^i\ \exp\left(x_k\,u_k^i + y_k\,v_k^{i,m}\right)$$

Figure 3.7 BCJR State and Branch Metrics (Sklar, 2008)

Branch metrics are calculated individually, while state metrics are computed recursively, dependent of the branch metrics and the state metrics of adjacent timestamp $k-1$ and $k+1$. Forward state metrics of the first timestamp is commonly initialized by assuming that the encoder starts at state zero, while the reverse state metrics of the last timestamp are initialized according to the knowledge of what state the encoder's going to end at. Once every branch and state metrics is calculated for every input codeword at time $k = 1, \ldots, n$ where n is the data block length, the data bits likelihood value $L(\hat{d}_k)$ can be calculated with equation 3.1a (Sklar, 2008). Referring to the equation, the likelihood value is expressed as the logarithmic of the ratio between all the probabilities of the bit having value 1 and probabilities of having value 0. That is, the more positive the likelihood value, the higher possibility the original transmitted bit of having value 1, and vice versa. When used in the

iterative Turbo decoding, the last decoder output this likelihood value on the last iteration. However, during the iterative process where constituent decoders are still exchanging information, an extrinsic value $Le(\hat{d}_k)$ is calculated and exchanged instead of the likelihood value. The extrinsic value is defined by equation 3.1b. Since both the likelihood and extrinsic value computations serve similar purpose of defining the possibility of a bit having value 0 or 1, and for the sake of simplicity, those two will be referred as likelihood ratio from here onwards.

$$\boldsymbol{L(\hat{d}_k) = \log\left[\frac{\sum_m \alpha_k^m \delta_k^{1,m} \beta_{k+1}^{f(1,m)}}{\sum_m \alpha_k^m \delta_k^{0,m} \beta_{k+1}^{f(0,m)}}\right]} \qquad \textbf{(3.1a)}$$

$$\boldsymbol{Le(\hat{d}_k) = \log\left[\frac{\sum_m \alpha_k^m exp(\frac{y_k v_k^{1,m}}{No^2}) \beta_{k+1}^{f(1,m)}}{\sum_m \alpha_k^m exp(\frac{y_k v_k^{0,m}}{No^2}) \beta_{k+1}^{f(0,m)}}\right]} \qquad \textbf{(3.1b)}$$

Referring to the details of BCJR algorithm particularly its metrics calculation, it is obvious that in practice this algorithm would impose relatively high computational and memory requirements. Assume a convolutional encoder of rate 1/2 and two memory elements, which resulted in a trellis diagram with four states. In a case where there are received codeword for timestamp $k = 1, \ldots, 3$, a BCJR decoder would need to construct a trellis diagram with four states and three group of transitions corresponding to the received codeword such as shown in Figure 3.8.

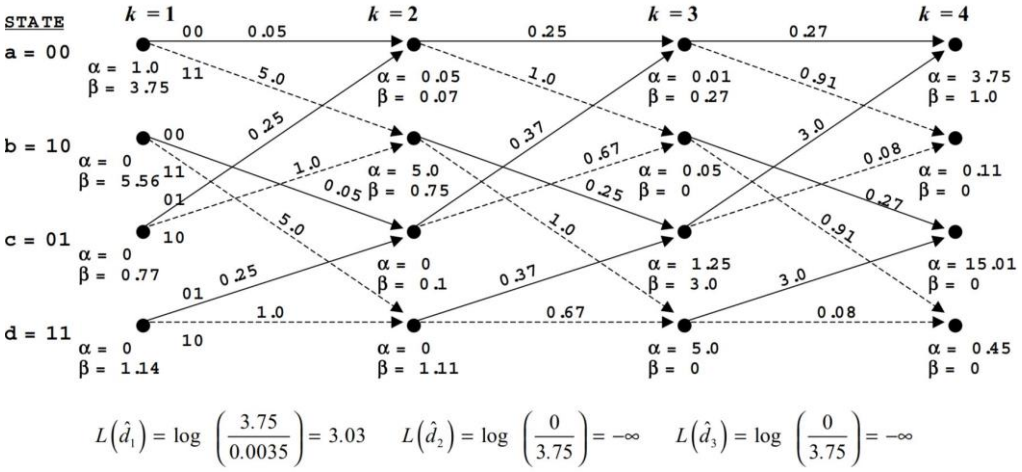


Figure 3.8 Sample of BCJR Decoder Trellis Diagram (Sklar, 2008)

It is noticeable that the trellis diagram constructed has four timestamps, whereas there are only three codeword timestamps received. The trellis diagram is initialized at timestamp $k = 1$, undergoes states transition by the first codeword at timestamp $k = 2$, and so on, resulting in the four timestamps. The encoder's state at the last timestamp is referred as the termination state. Forward state metrics is not computed at the last timestamp $k = 4$ since it is not needed by the bit likelihood calculation in equation 3.1a or the extrinsic information in equation 3.1b, and so does the reverse state metric for the first timestamp $k = 1$. The decoder needs to compute and store twelve forward and twelve reverse state metrics, plus twenty-four branch metrics totaling in forty-eight metrics just for the decoding of three data bits. Coupled with the fact that the calculation of each metric involves numerous complex operations such as logarithmic, exponential, and especially multiplications, it is straightforward that the BCJR algorithm is relatively computational and memory expensive.

$$\tilde{\delta}_k^{i,m} = \ln\left(\delta_k^{i,m}\right) = \ln\left(\pi_k^i\right) + (x_k u_k^i + y_k v_k^{i,m})/No \qquad \textbf{(3.2a)}$$

$$\tilde{\alpha}_k^m = \ln(\alpha_k^m) = \max(\alpha_{k-1}^{b(0,m)} + \delta_{k-1}^{0,b(0,m)}, \alpha_{k-1}^{b(1,m)} + \delta_{k-1}^{1,b(1,m)}) \quad \textbf{(3.2b)}$$

$$\tilde{\beta}_k^m = \ln(\beta_k^m) = \max(\beta_{k+1}^{f(0,m)} + \delta_k^{0,m}, \beta_{k+1}^{f(1,m)} + \delta_k^{1,m}) \quad \textbf{(3.2c)}$$

$$\tilde{L}(\hat{d}_k) = \ln\left(L(\hat{d}_k)\right) = max_{all\ m}\left(\alpha_k^m \delta_k^{1,m} \beta_{k+1}^{f(1,m)}\right) - max_{all\ m}\left(\alpha_k^m \delta_k^{0,m} \beta_{k+1}^{f(0,m)}\right) \quad \textbf{(3.2d)}$$

$$\widetilde{Le}(\hat{d}_k) = \ln\left(Le(\hat{d}_k)\right) = max_{all\ m}\left(\alpha_k^m \frac{y_k v_k^{1,m}}{No^2} \beta_{k+1}^{f(1,m)}\right) - max_{all\ m}\left(\alpha_k^m \frac{y_k v_k^{0,m}}{No^2} \beta_{k+1}^{f(0,m)}\right) \quad \textbf{(3.2e)}$$

Among several algorithms proposed to solve the computational complexity of the BCJR algorithm, two widely used algorithms are Max-Log-BCJR and Log-BCJR (Robertson, Hoeher, & Villebrun, 1997). Both algorithms modify the metrics and likelihood ratio calculation by taking their natural logarithm, i.e. $\tilde{\alpha}_k^m = \ln(\alpha_k^m)$ for forward state metric, and similarly for the other metrics and the likelihood ratio. Doing this reduces the number of multiplications due the logarithmic property $\ln(ab) = \ln(a) + ln(b)$, while the removal of logarithm operation itself is possible through the approximation of $\ln(a + b + c + \cdots) = \max(a, b, c, \dots)$, where max function finds the maximum among its input. These two properties yield the restatement of the branch, forward state, reverse state metric, bit likelihood equations, and extrinsic information in 3.2a, 3.2b, 3.2c, 3.2d, and 3.2e, respectively. These equations are less complex, yielding simpler and computationally inexpensive implementation in expense of the error rate. In Log-BCJR algorithm the result of the max function is added by a correction function (Sadjadpour, 2000) to compensate for the error rate.

While the Max-Log-BCJR and Log-BCJR algorithms reduce the computational complexity of the original algorithm, they do not improve the memory usage of the algorithm. Both algorithms still require the same memory resource as the original to store all the metrics required for each bit's likelihood ratio calculation. A widely used solution is sliding-window decoding which is detailed in the next section.

3.3.2 Sliding-Window BCJR Decoding Algorithm

As discussed in the previous section, BCJR algorithm requires large storage for its metrics before a final likelihood ratio can be computed. Sliding-window algorithm divides the received frame into several smaller chunks then performs the BCJR decoding on those chunks separately, greatly reducing the memory requirements. A window of fixed length w is applied to the received data frame starting from the first data, then performs BCJR decoding only on data belonging to the window. The beginning of the window then is moved as far as its size w to the right, and another round of BCJR decoding is performed on those data in the window. This operation is continued until all data in the frame has been decoded.

Figure 3.9 illustrates the concept of sliding-window algorithm. The algorithm places the first window over the data of timestamp $k = 0,1,2,\dots,w-1$, decodes them then slides the window to the next group of data of timestamp $k = w, w+1, \dots, 2w-1$, and so on. In this illustration, the first window's reverse state metric calculation (backward recursion) is performed over data range of $k = 0,1,2,\dots,w+g-1$ while the window only occupies and decodes

data up to $k = w + 1$. This few extra g data (or bits) provide extra information to the current window to improve the error rate and are referred as the guard window. This is applied similarly for every window.

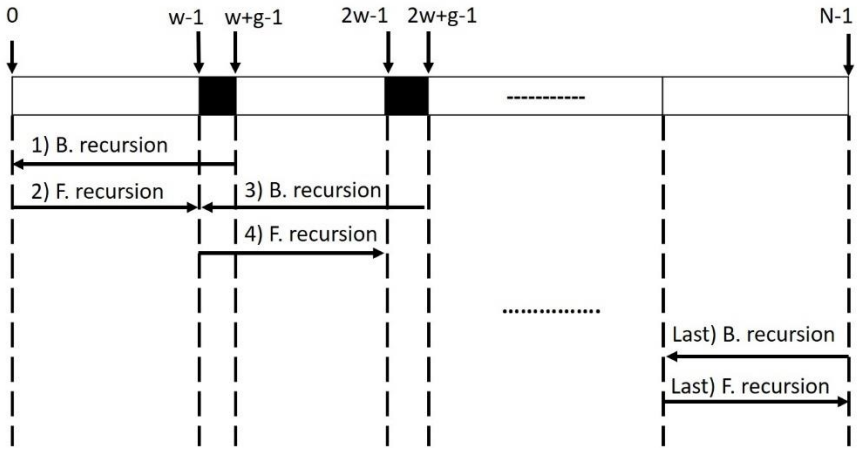


Figure 3.9 Sliding Window Algorithm (Marandian, Fridman, Zvonar, & Salehi, 2001)

Assume the same four-state encoder which produces trellis diagram in Figure 3.8. On the receiving side, the decoder requires forty-eight metrics for only three data bits, or sixteen metrics per data bit. Should the received frame contain a hundred bits, all 480 metrics must be stored before the final likelihood computation. If a sliding-window approach is used instead, the number of metrics to be stored can be reduced depending on the window (and guard window) size. Smaller window decreases the memory usage in expense of the performance loss. During implementation, the main trade-off between window size and error rate hence is the key to satisfy certain requirements of the wireless system, or to fit into the limitations imposed by the hardware. As the RUMPS401 is limited in terms of storage, the application of sliding-window

algorithm is essential. Its main trade-off will be studied through a simulation model whose details is covered on upcoming sections.

3.3.3 3GPP UMTS Turbo Code Standard

This work adapts Turbo Code structure used by the Universal Mobile Technology System (UMTS) standard for the third-generation cellular system. This standard was defined by Third-Generation Partnership Project (3GPP), an association of multiple organizations that develop and define telecommunications standard. Standards developed by 3GPP such as 3G, High Speed Downlink Packet Access (HSDPA), Long Term Evolution (LTE), are referred as 3GPP technologies, and has been widely adapted throughout the industry.

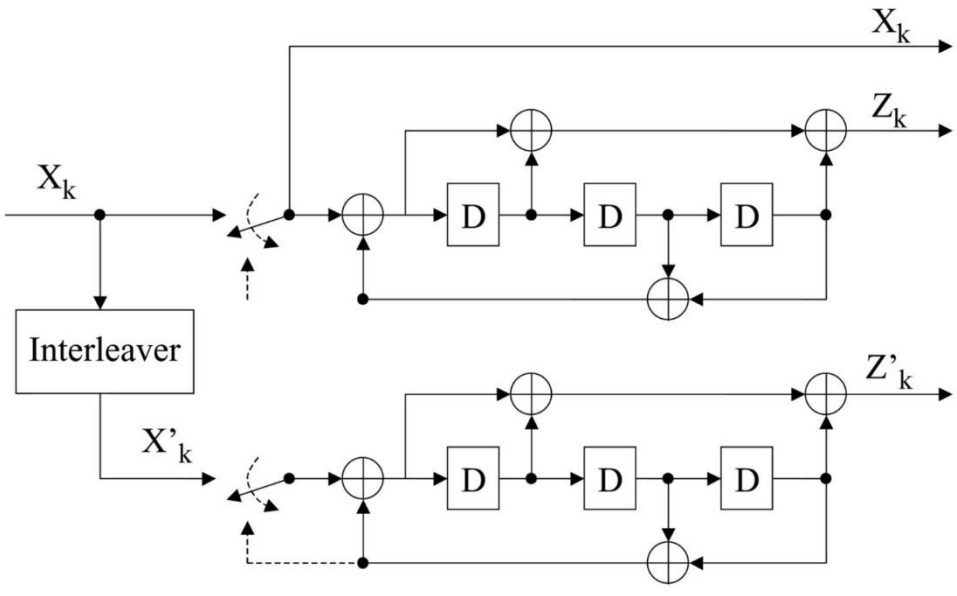


Figure 3.10 Structure of 3GPP UMTS Turbo Encoder (Valenti & Sun, 2001)

Figure 3.10 shows the UMTS standardized Turbo encoder, which consists of two identical Recursive Systematic Convolutional (RSC) encoders, a single interleaver, and trellis termination mechanism. RSC code is a specific type of convolutional code with a feedback line to the input of the first memory element. For every input bit the encoder produces two type of output bits, that is the input bit itself (systematic bit) and the parity bit. RSC code was first introduced in the Turbo Code and has been widely adapted due to its superior error rate performance compared to the non-systematic and non-recursive convolutional code (Berrou, Glavieux, & Thitimajshima, 1993). Combining the two encoders, the UMTS standardized Turbo encoder has a total rate of 1/3, whose output are one systematic bit and two parity bits generated by each encoder.

As in other types of convolutional code the operation of the RSC code can be described in a trellis diagram. In practice, polynomial description of a convolutional encoder is sometimes preferred over the trellis diagram for describing the encoder's structure. It describes the connection among the memory elements (or shift registers in practical implementation) and the modulo-two adders as sequence of bits, represented in octal number format. For example, the rate 1/2 RSC encoder in Figure 3.10 has two lines of connection, one parity-generating line and one feedback line, each represented by a modulo-two addition. The encoder has a depth of $K = 4$, hence there are four elements with possible connection to the adder on each line. On the parity-generating line, the input bit, the first register, and the third register are connected to the adder. This connection can be represented as a binary number 1101 or its octal

format 15. Similarly, the feedback line is represented as binary 1011 or octal 13. These numbers are referred as the generator polynomial of a convolutional encoder. Coupled with the encoder's rate and depth, they can be used as a compact description of a convolutional encoder.

The UMTS standard Turbo Code uses matrix interleaver to scatter data bits going into the second encoder (Valenti & Sun, 2001). It arranges data bits by writing the bits row by row in a matrix, then reads them back per column. The matrix size is adjusted according to the data frame size. While this interleaver is easy to implement, it requires an index mapping table whose size grows in linear to the frame size. Matrix interleaver also suffers from worse error rate performance compared to other type of pseudorandom interleaver, which will be shown by Matlab simulation in the next section.

In the UMTS standard, Turbo Code's trellis diagram must be zero-terminated, which means that at the end of the encoding process every memory element must be reset to zero value. By always terminating the encoder to a known state, the reverse state metric can be initialized properly thus improving the decoding result (Bahl, Cocke, Jeinek, & Raviv, 1974). This is done by inserting few extra bits to the encoder after the last data bit. These bits are commonly referred as the termination bits, whose number equals the number of memory elements. The termination bits sequence is a function of the last state the encoder was at, which is depicted in Figure 3.10 by the connection formed when the switch in each encoder is tapped to the feedback line instead of input bit.

Decoding of the UMTS standardized Turbo encoder is performed by a common Turbo decoder shown in Figure 2.6, given that the decoder possesses the same trellis diagram. The high-level modeling and simulation of both encoder and decoder shall be covered in the following section.

3.4 Turbo Code Modeling and Simulation in Matlab

The high-level model of the Turbo encoder and decoder along with the BPSK modulator and demodulator will be built in Matlab. It is a matured numerical computing environment developed by Mathworks. Matlab provides numerous ready-to-use libraries for various applications such as complex mathematical computation, data analysis and representation, signal and image processing, communication system (Mathworks, n.d.). The enormous libraries coupled with Matlab's high-level scripting language accelerates the model development and simulation process, allowing proper understanding of the system's structure and parameters. Furthermore, the scripting language is procedural thus the model developed with it can be used the base for the practical implementation in other programming language such as C.

This subchapter discusses the modeling and simulation of Turbo-coded system to study its characteristic. The Turbo code simulation is split into two steps. The first step employs full-frame decoder on the receive side to verify that the model works properly. Simulation is run by varying few basic parameters of Turbo code, that is the iteration number, the data frame size, the decoding algorithm. The second step takes the simulation model further for practical implementation by employing the sliding window decoder, with

window size and interleaving pattern as the study parameters. This two steps simulation resulted in a high-level model of sliding-window Turbo decoder with fixed parameters, upon which the RUMPS401's Turbo Code implementation is based on.

The rest of the subchapter is arranged into three sections as follow. The first section describes the model setup. The scenario and results for the first and second simulation steps are presented in section two and three, respectively.

3.4.1 Simulation Setup

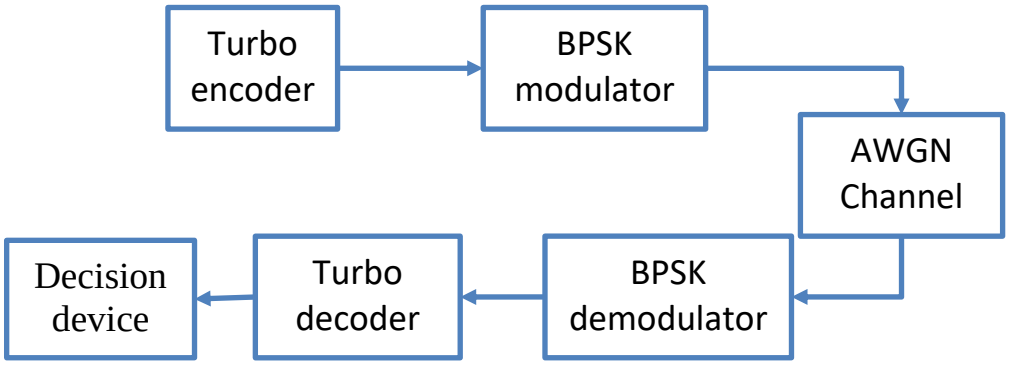


Figure 3.11 System Model for Matlab Simulation

Depicted in Figure 3.11 is the complete Turbo-coded system's model consisting of a Turbo encoder and a BPSK modulator on the transmit side, as well as a Turbo decoder and a BPSK demodulator on the receive side. The model performs baseband half-duplex communication. Simulation is then carried out with the modelled system by transmitting random data and measuring the receiver error rate over AWGN channel with varying SNR. Every

simulation is performed on two hundred thousand bits per SNR value. The simulation is performed over various scenario by varying several key parameters of Turbo code to study their effect on the receiver's error rate. A simple BPSK transmit-receive model is simulated too as the baseline for the error rate performance of non-coded system. This non-coded system model is identical to the Turbo-coded system model except it does not have the Turbo encoder and decoder block.

As shown in Figure 3.11, the Matlab's simulation model comprises of six key components: the BPSK modulator and demodulator, the Turbo encoder and decoder, the noisy AWGN channel, and the decision device. Matlab's AWGN function is used to implement the noisy channel model. The function adds Gaussian noise to the input signal according to the SNR parameter as its output, as shown in Figure 3.12. As shown in Figure 3.13, the modulator is implemented with Matlab's `pskmod` function, which accept the data signal, the modulation order M, and the constellation offset. In this simulation, a BPSK with zero-offset constellation is chosen, hence the `pskmod` function is run with the randomized data signal, the M of two, and the constellation offset of zero. The function outputs the modulated data for each bit which takes form of complex number.

In practice, the coherent BPSK demodulator should perform various synchronization on the received signal to correct the timing and frequency experienced by the received signal. Since the simulation focuses on studying the Turbo Code characteristic, it simulates on an ideal baseband transmitter and receiver system. There is no frequency offset in baseband transmission, and the

demodulator is set to sample the signal at the exact same time as the transmitter. The demodulator then passes the sampled signal to the Turbo decoder, whose result is then used by the decision device to determine the bit value.

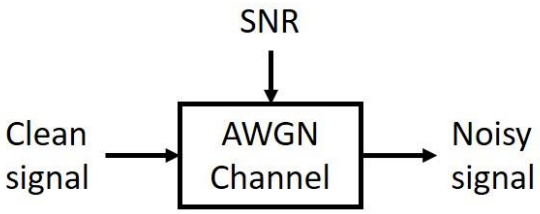


Figure 3.12 Matlab's AWGN Function

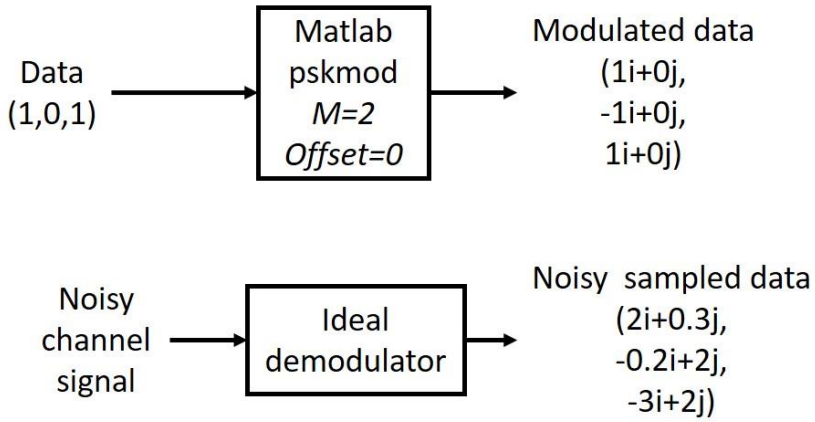


Figure 3.13 BPSK Modulator/Demodulator Model

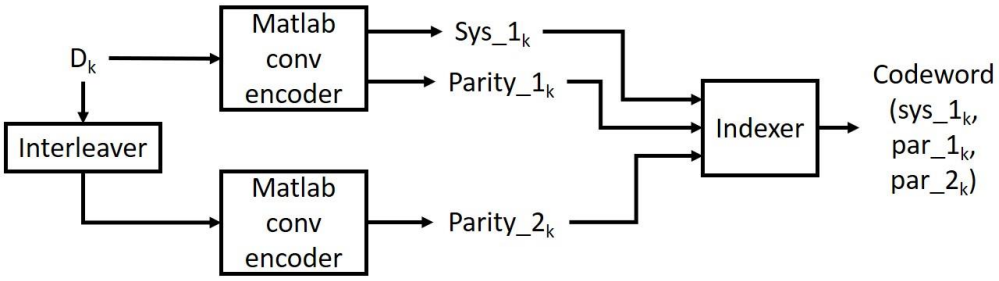


Figure 3.14 Turbo Encoder Model

Figure 3.14 shows the high-level model of the Turbo encoder. The encoder model takes the input sequence and provides this sequence along with its interleaved version as the input for the two parallel RSC encoders. Matlab's convolutional encoder function is used to model the RSC encoder. The function encodes input bits into codeword, provided the encoder's trellis description as the input parameter. As discussed in previous section, polynomial description is much more compact and can also be used to describe an encoder's structure. This can be leveraged by using Matlab's `poly2trel` function to convert an encoder's polynomial description into the corresponding trellis description. Outputs of both encoders are then indexed accordingly.

As described earlier, the encoder structure referred in this work belongs to the UMTS standard which uses trellis termination mechanism to ensure that the encoder is terminated at a known state. However, on the receiver that uses sliding-window decoding, each window's metrics are initialized separately, thus the improvement provided by the trellis termination at the end of the full frame is less significant as it only applies to the last chunk of a frame. As the real system implementation on the RUMPS401 uses sliding-window decoding, the trellis termination bits generation is removed from both simulation model and real system implementation.

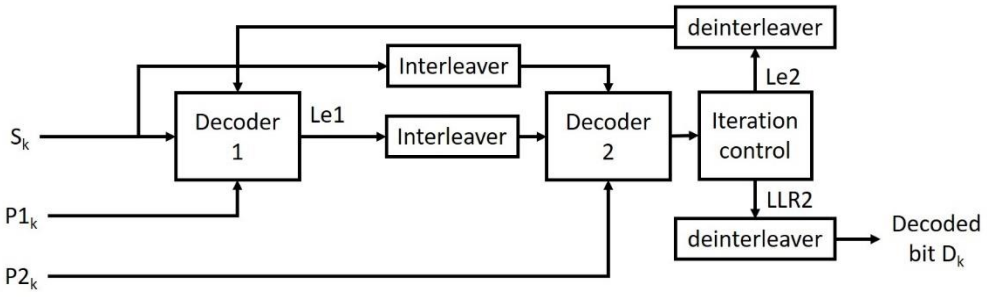


Figure 3.15 Turbo Decoder Model

The decoder model shown in Figure 3.15 comprises of two serially concatenated decoders, interleavers and an iteration control. Every component in the decoder model is self-coded, except for some interleaving algorithm. Both constituent decoders are identical and can be configured to run either BCJR algorithm or its suboptimal Max-BCJR algorithm. The interleavers and deinterleavers are black boxes which can be replaced with any specific interleaving pattern according to the simulation and parameter study needs. Each simulation scenario in both the first and second step uses Matlab's pseudorandom interleaving pattern, except for the last scenario of the second step where interleaving patterns are compared. The iteration control block tracks the number of decoding loops performed. It outputs the extrinsic information of the received bits as feedback to the first decoder for next loop, and outputs the likelihood ratio of the received bits at the end of the last loop.

3.4.2 Result of Full-window BCJR Decoding Simulation

Figure 3.16 through Figure 3.18 shows the results of the first simulation step for which iteration number and data frame size are varied. These simulations were run with Matlab's pseudorandom interleaving algorithm, and

BCJR algorithm on the decoder side. It is evident from these graphs that higher iteration number yields lower error rate. However, the error rate improvement provided per iteration number is deminishing as the iteration increases. For example, the amount of error rate improvement provided by increasing the iteration number from 3 to 4 is lower compared to increasing from 1 to 2. It is even less when increasing from 5 to 18 iterations. This is a common behavior of Turbo code where much higher iteration number provides very little or almost no error rate improvement compared to the lower iteration number (Berrou, Glavieux, & Thitimajshima, 1993), known as the saturation point of the Turbo code. The saturation point defines the optimal number of iterations needed by certain Turbo-coded system to provide the lowest possible error rate. More intelligent Turbo-coded system tracks the saturation behavior by monitoring the error rate improvement and decides when the iterative decoding should end. This allows the system to decode more efficiently by suppressing the number of decoding iterations performed whilst maintaining an optimal error rate improvement.

Another observable behavior from Figure 3.16 through Figure 3.18 is the lower error rate provided by larger data frame size. The larger frame size provides more information to the decoder, allowing the BCJR algorithm to traverse over longer trellis graph (Bahl, Cocke, Jeinek, & Raviv, 1974) yielding a more confident decoding result. This is achieved at the expense of larger processing and memory resources to process more data bits. It should be noticed that the frame size starts to significantly affect the error rate on higher iteration number. The first iteration decoding for systems with frame sizes of 40, 256,

and 1024 bits provide similar error rate, while on the second and higher iteration the system with 1024 bits performs better than the other two. This behavior is more evident on Figure 3.19 which presents the simulation result when the system is set to run four decoding iterations and compared for varying frame size.

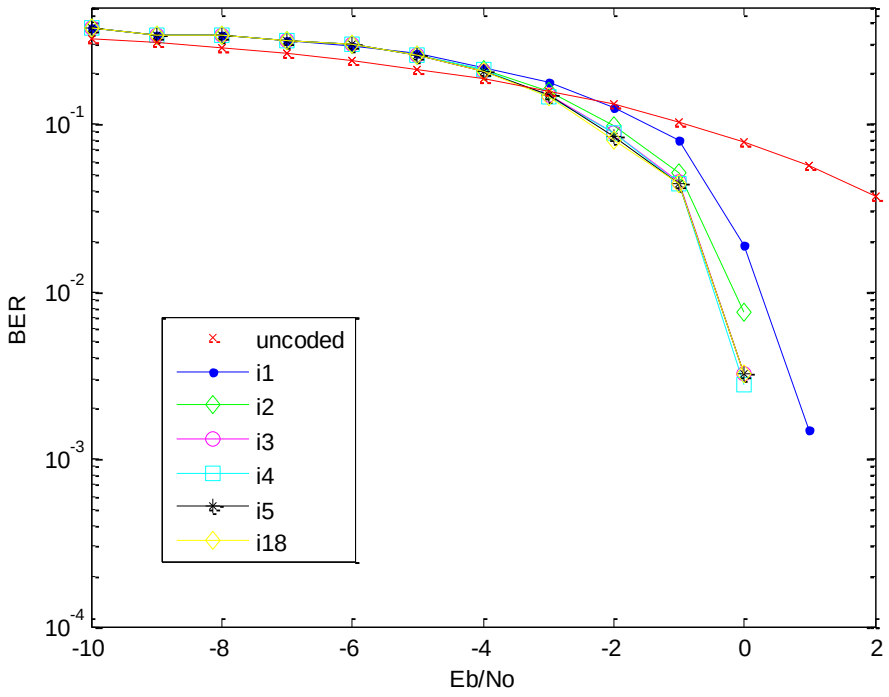


Figure 3.16 Turbo Model Simulation – 40 bits Frame, Varying Iteration

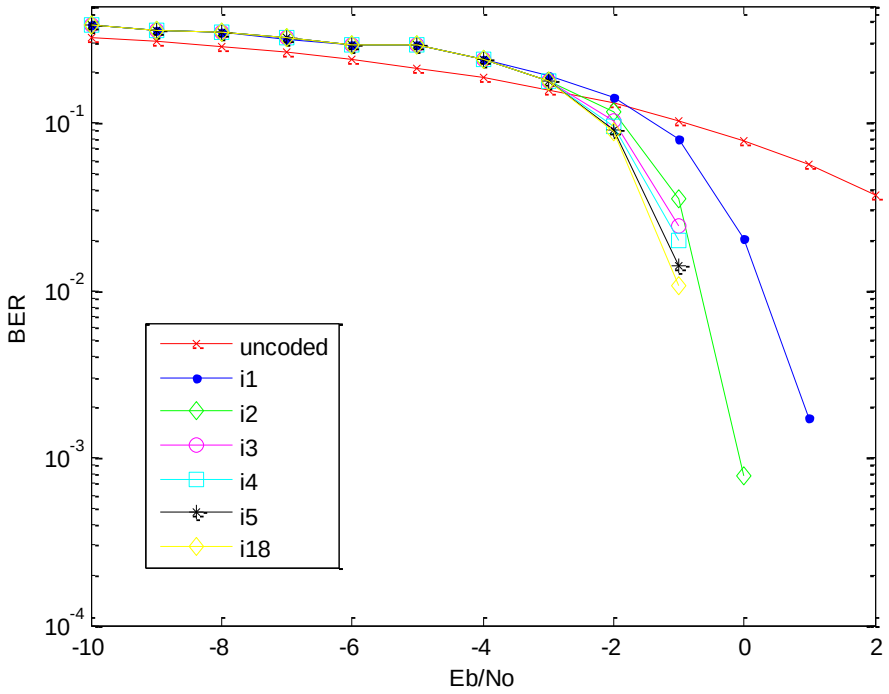


Figure 3.17 Turbo Model Simulation – 256 bits Frame, Varying Iteration

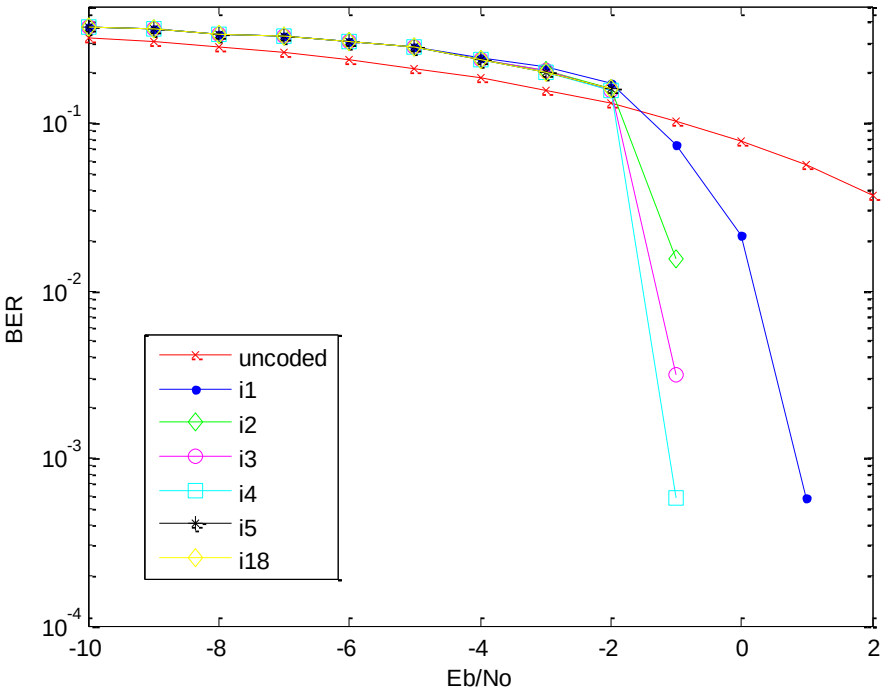


Figure 3.18 Turbo Model Simulation – 1024 bits Frame, Varying Iteration

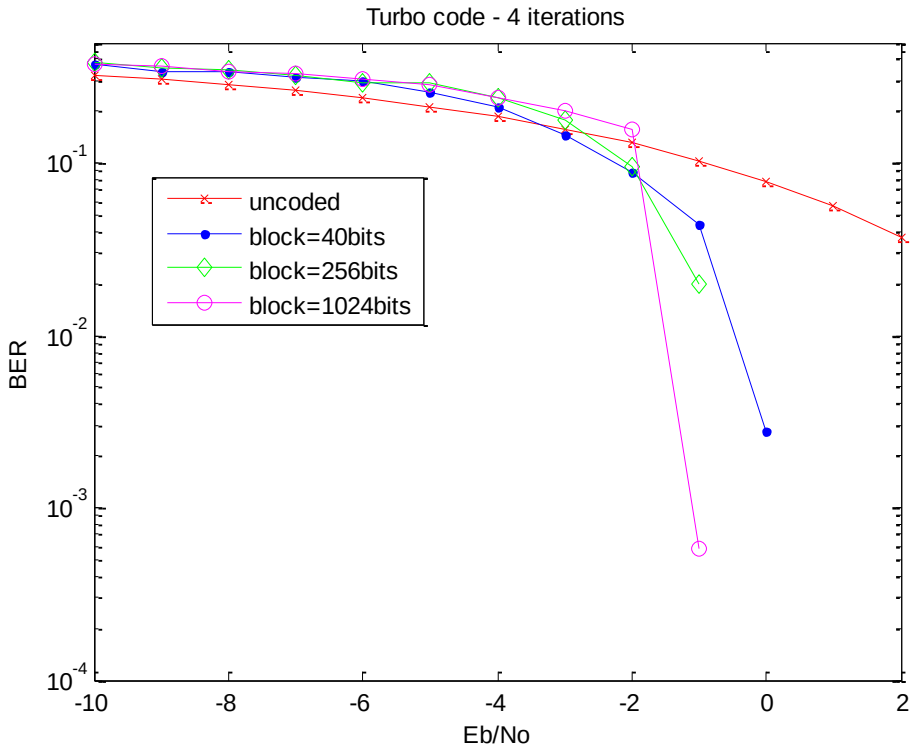
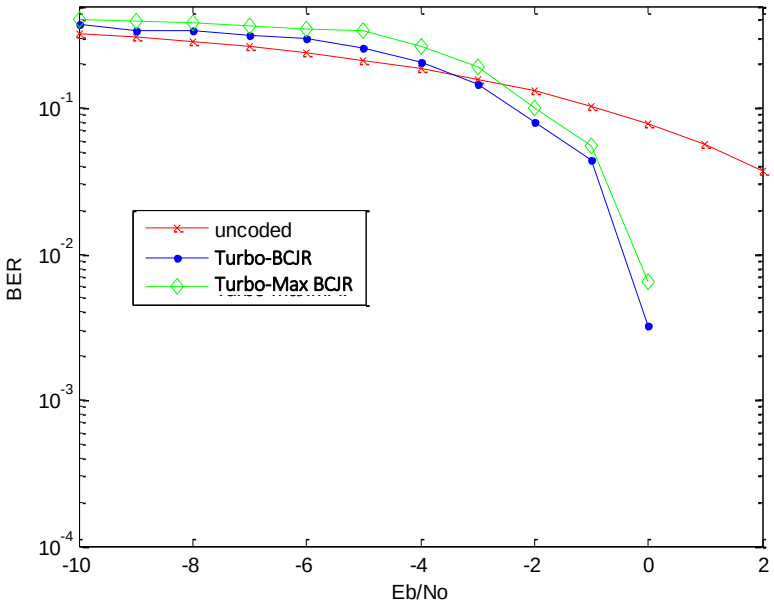


Figure 3.19 Turbo Model Simulation - Frame Size Comparison

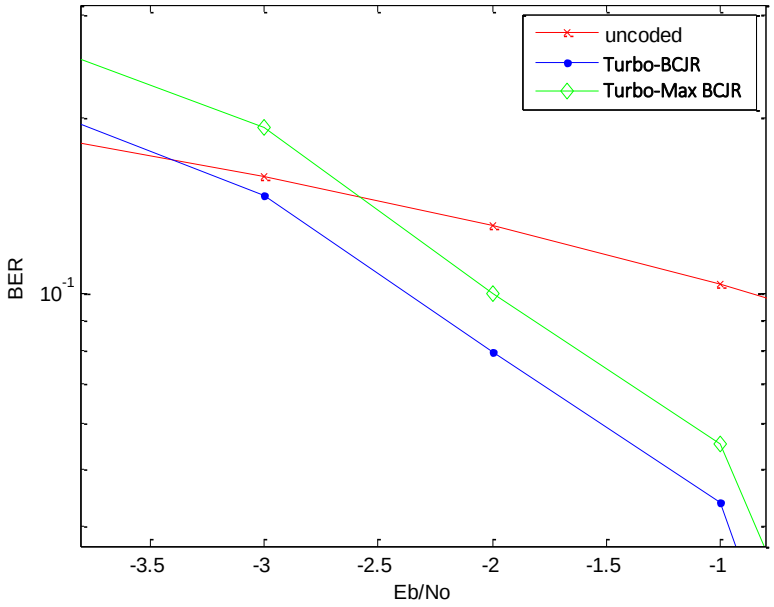
As discussed in Chapter Two, one of main reasons the Turbo decoding process being relatively computational expensive is the use of BCJR algorithm in both of its constituent decoder. It was also reviewed that one of fundamental approaches for simplifying the Turbo decoding is by utilizing the suboptimal version of the BCJR algorithm instead of the original version. The BCJR algorithm performs better than its suboptimal version Max-Log-BCJR algorithm (Robertson, Hoeher, & Villebrun, 1997). This is illustrated in the simulation results shown in Figure 3.20 and Figure 3.21. It is evident from the graph that the model with original BCJR algorithm provides around 0.5 – 1 dB improvement over the suboptimal Max-Log-BCJR algorithm, and that the improvement is more significant on the model with frame size of 256 bits. Despite the worse error rate performance provided, the Max-Log-BCJR

algorithm is used in most real application due to its modest complexity (Li, Maunder, Al-Hashimi, & Hanzo, 2013; Huang, et al., 2011). This last scenario completes the first of two steps simulation described earlier, and verifies that the system model functions in a proper way. The model then can be taken into the second simulation step, which requires modification on its decoder part. As the second part of the simulation intends to bring the model closer to practical implementation, Max-Log-BCJR algorithm is used for each simulation scenario in the second part.

Change on the decoder model is made by modifying its Max-Log-BCJR algorithm which decodes a full frame at once into sliding-window version (Marandian, Fridman, Zvonar, & Salehi, 2001). A sliding-window decoder is commonly characterized by its window and guard-window size. Window size defines the number of bits being decoded at a time, while guard-window size describes the number of extra bits from the next window which is added to the current window to provide extra information to the decoder hence improving the decoding performance. These additional bits are not decoded when a specific window is being decoded, they are decoded when their own window is being decoded. The resulting sliding-window decoder model is then simulated for varying window and guard-window size, with the full frame size fixed at 256 bits. Comparison of interleaving patterns will be performed on the same sliding-window decoder model as well.

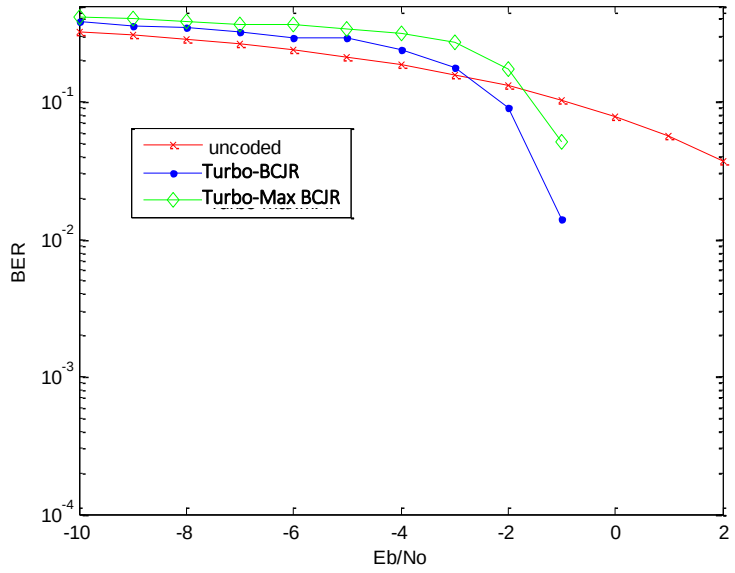


(a) Frame Size 40 bits, 18 iterations

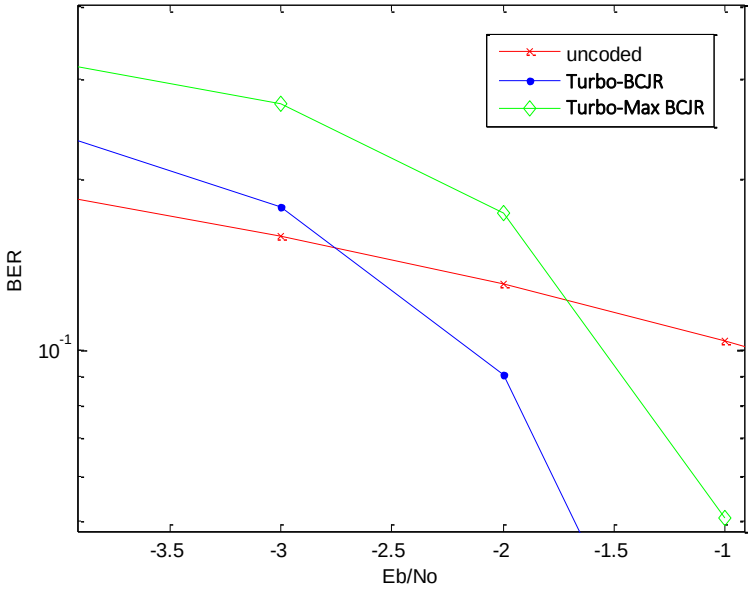


(b) Frame Size 40 bits, zoomed

Figure 3.20 Turbo Model Simulation - BCJR Compared, 40 bits Frame



(a) Frame Size 256 bits, 5 iterations



(b) Frame Size 256 bits, zoomed

Figure 3.21 Turbo Model Simulation - BCJR Compared, 256 bits Frame

3.4.3 Result of Sliding-window BCJR Decoding Simulation

Figure 3.22 through Figure 3.25 presents comparison of the sliding-window model's simulation results over various window and guard-window size. It is clear from the graph that larger window and guard-window size leads

to better error rate in expense of more bits to process, like the effect of frame size on the full frame decoding. Another observable behavior from the four graphs is the Figure 3.23 where the error rate declines in a flat manner on high SNR, as opposed to the expected curve manner. This flat slope behavior is commonly referred as the error floor of an error correction code (Garello, Chiaraluce, Pierleoni, Scaloni, & Benedetto, 2001). The error floor phenomenon is not acceptable for applications that requires low error rate, which can be mitigated by increasing minimum distance of the error correction code. In Turbo Code, this can be achieved by employing an interleaving pattern with good spreading property (Crozier & Guinand, 2001).

The last part of the two steps high-level model verification and simulation works around another main component of a Turbo encoder and decoder, that is, the interleaver and deinterleaver. Interleaving scatters the data frame according to a specific pattern, the more scattered the data frame, the lower the error rate. Additionally, it has impact on diminishing the error floor phenomenon. On Turbo Code, the interleaving is performed on the data frame going into the second encoder or decoder. It should be recalled that the very reason Turbo Code utilizes two parallel encoders is to introduce redundancy into the transmitted frame. Should a burst error occur in a transmission, a block of consecutive data might get corrupted. While this kind of error may affect the first decoder, the second encoder is unaffected due to the interleaving process.

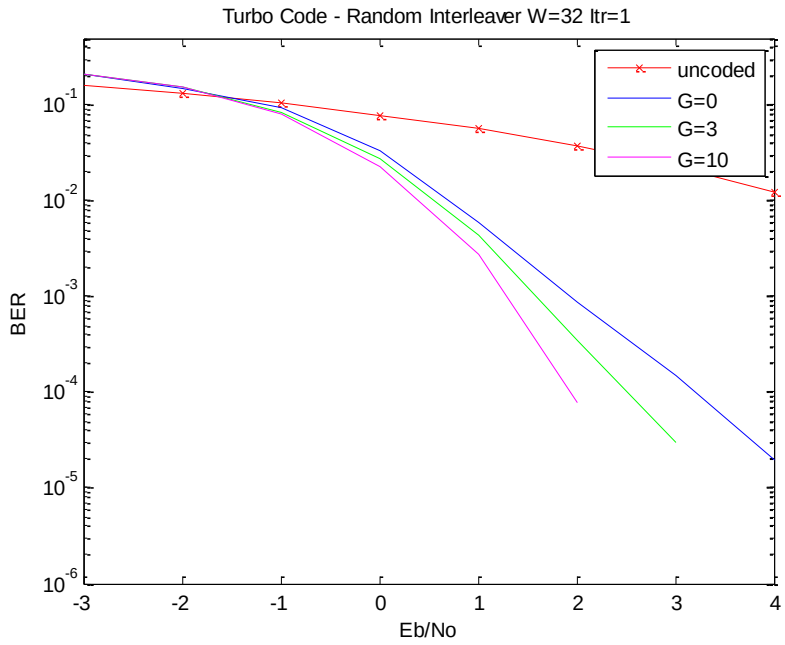


Figure 3.22 Turbo Model - Sliding Window Comparison, W=32 Itr=1

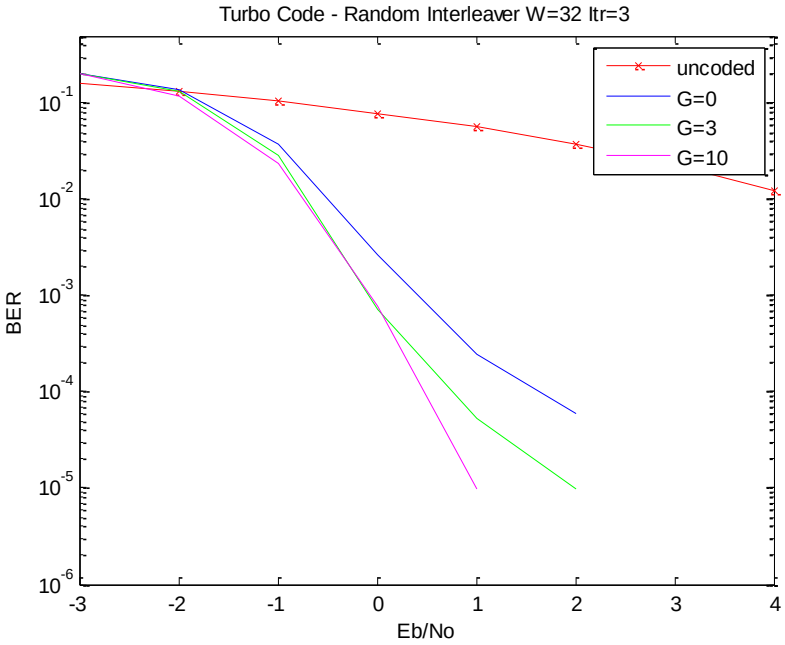


Figure 3.23 Turbo Model - Sliding Window Comparison, W=32 Itr=3

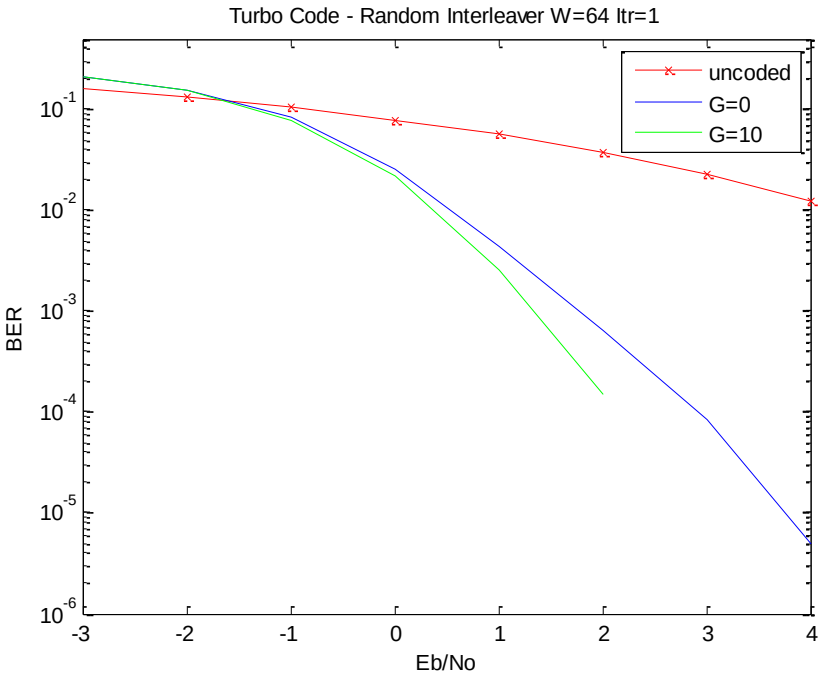


Figure 3.24 Turbo Model - Sliding Window Comparison, W=64 Itr=1

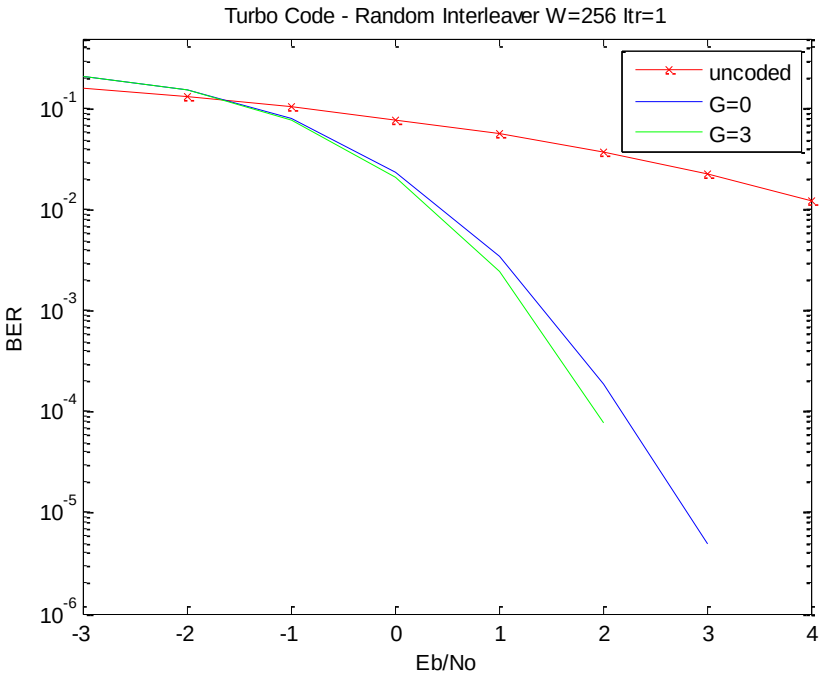


Figure 3.25 Turbo Model - Sliding Window Comparison, W=256 Itr=1

A good interleaver should have good data scattering pattern and should require as less computational and memory resources as possible. A Dithered

Relative Prime (DRP) interleaver satisfies both requirements (Crozier & Guinand, 2001). Its interleaving algorithm requires very little memory and performs simple calculations during the iterative interleaving process, while providing good data scattering pattern. Figure 3.26 and Figure 3.27 show the simulation result for comparison of three interleaving pattern, that is DRP interleaving, Matlab pseudorandom interleaving which runs Mersenne Twister algorithm (Matsumoto & Nishimura, 1998), and matrix interleaving. These simulations were performed on the system model with a frame size of 256 bits and runs sliding-window decoder with Max-Log-BCJR algorithm. It is evident that matrix interleaver is inferior to the other two, and that Matlab pseudorandom interleaver still exhibit error floor phenomenon. DRP interleaver provides lower error rate compared to the other two and no error floor at this range of error rate and SNR, thus is chosen for the real system implementation on the RUMPS401.

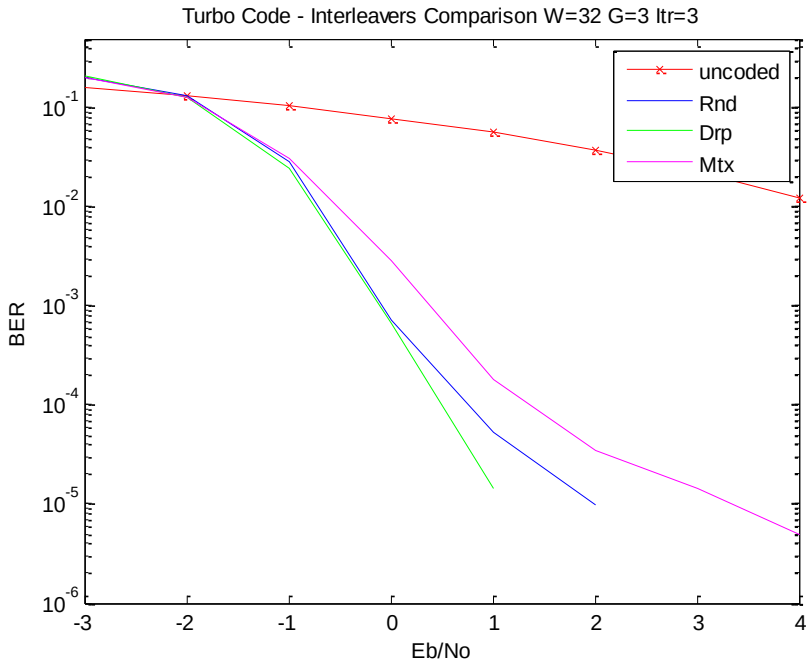


Figure 3.26 Turbo Model - Interleaver Comparison W=32 Itr=3

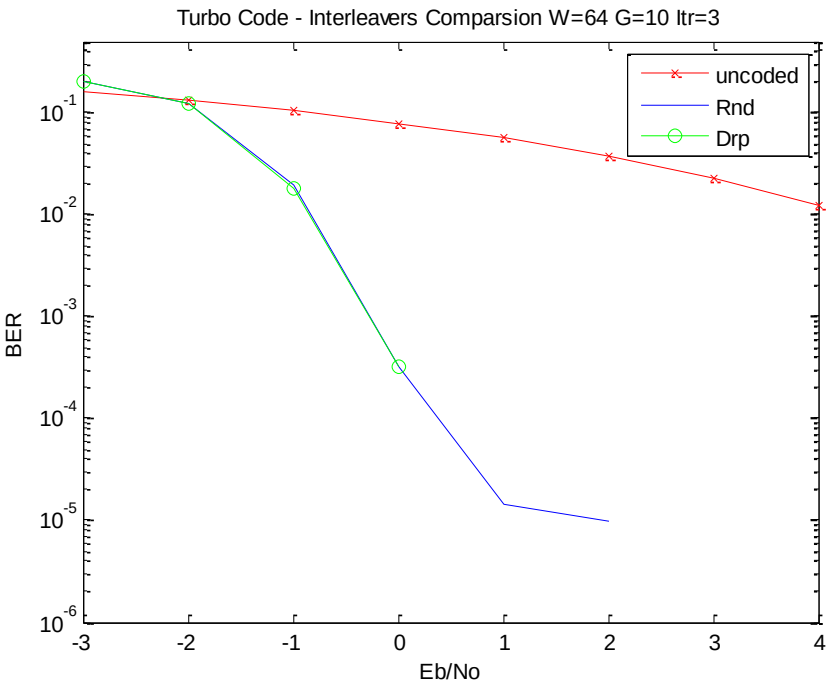


Figure 3.27 Turbo Model - Interleaver Comparison W=64 Itr=3

3.5 Summary

In this chapter, extensive reference literature study and simulation have been performed, with the goal of deciding which wireless modulation scheme and the Turbo Coding structure to be implemented on the RUMPS401. As inferred from the study on digital modulation scheme, coherent BPSK is the suitable choice for a system where data rate and spectral efficiency is less of a concern compared to the system's error rate. After a thorough study and simulation of the spread spectrum technique particularly DSSS, the narrowband system is chosen due to the fact that it exhibits the same error rate as the spread spectrum system and it is less demanding for implementation in the RUMPS401. This leads to the final decision to use a coherent BPSK as the digital modulation for the practical system implementation on the RUMPS401.

The coherent receiver requires proper synchronization. The implementation details will be covered in Chapter five.

As described earlier, the Turbo Code simulations are intended to produce a high-level model upon which the practical system implementation will be based on. At the end of the two steps simulation, a high-level encoder and decoder system model has been built and verified. The decoder adopts the more practical Max-Log-BCJR implementation together with the sliding window decoding algorithm. In addition to providing a high-level model of a Turbo encoder and decoder, the simulation serves as the base for deciding on the necessary Turbo code parameters. The data frame size is fixed at 256 bits with window size of 32 bits. According to the simulation results, these numbers provide a good error rate performance without imposing memory and computational issues for the RUMPS401, which will be further detailed in Chapter four. Lastly, the DRP interleaving pattern will be used for its superior error rate performance and low complexity.

CHAPTER 4

TURBO DECODER IMPLEMENTATION AND OPTIMIZATION ON THE RUMPS401

The previous chapter covers in details reference literature studies and simulation of both the modulation scheme and the Turbo Code, which resulted in a high-level model with specific parameters. The high-level model of Turbo encoder and decoder which were written in Matlab's language are ported to C-based code, specifically optimized for the RUMPS401 with proper understanding of the chip's architecture. It is obvious that the decoder's operation is much more complex compared to the encoder which can be implemented with a simple lookup table. This chapter discusses the Turbo encoder and decoder software development on the RUMPS401, with focus on the decoder's optimization. The rest of this chapter is arranged into three parts as follows. The first part describes the primary concept of the software design and the details of software development environment. In the second part, step-by-step optimization process of the decoder software is presented in detail, along with execution time of each task involved in the decoding process. Finally, the last part summarizes the chapter.

4.1 Turbo Decoder Software Implementation on the RUMPS401

The extensive initial study and high-level simulation performed in the third chapter has resulted in a specific Turbo Code system that will be implemented in this project. The encoder adopts the Third Generation

Partnership Project (3GPP) Universal Mobile Telecommunication System (UMTS) standardized Turbo encoder structure, with no termination bits as it provides insignificant error rate improvement to the sliding window decoder. For practical deployment, the decoder runs suboptimal Max-Log-BCJR algorithm coupled with sliding window decoding. The size of the original data frame is fixed at 256 bits and decoded in chunks with sliding window of 32 bits. It was mentioned in Chapter Three that these numbers were chosen based on the error rate performance which has been established through the simulations, and the resource requirements which will be discussed in the rest of this chapter. Dithered Relative Prime (DRP) interleaving pattern is used due to its low computation and memory requirement, and its superior error rate improvement displayed in the simulation results.

The rest of the chapter is arranged into three sections as follows. The first section specifies the toolchains used for the RUMPS401 software development and deployment. Section Two discusses the Turbo encoder and decoder software structure, particularly the parallelism employed on the decoder. Details regarding the variables used in the software along with their memory usage are presented in the third section.

4.1.1 Implementation Setup

As discussed in previous chapters, the Turbo encoder and decoder is fully implemented via software on the RUMPS401. Those software were fully developed from scratch without the use of any external signal processing libraries, except for a few complex mathematical functions and fixed-point data

type which shall be detailed as this chapter progresses. Close-to-metal coding approach is applied as the software intends on maximizing the RUMPS401's resources to compensate for the low power and low-performance nature of the ARM Cortex-M0. The resulting program tends to directly access the hardware resources such as registers or specialized hardware to minimize the time overhead introduced by any function call, be it a self-written function or a function from external library.

Since this work implements a simplex wireless system with exactly one transmitter and one receiver, there are two sets of software code, one each for the encoder and decoder. Each set contains four software codes. These codes are compiled separately into the four individual binary files which then is programmed into each core's flash memory. As the RUMPS401 is fully custom designed by the UTAR VLSI's Research Center, there is no ready-to-use software Integrated Development Environment (IDE) from other sources that provides a complete toolchain for developing the RUMPS401's software.

The software was written in the C programming language. The code can be written with any text editor software, but Sublime Text 3 was used due to personal preference (Sublime, n.d.). It provides essential features for the programmer such as syntax highlighting, code folding, multiple files view. The code is then cross-compiled with GNU Embedded Toolchain for ARM (ARM, GNU ARM Embedded Toolchain, n.d.), and targeted for the ARM Cortex-M0 processor. The resulting binaries are then flashed into the RUMPS401 using a self-written Python-based program which reads the binary files and transfer them byte by byte into the RUMPS401 via the Universal Asynchronous

Receiver Transmitter (UART) interface. These bytes are received by a bootloader program running in the IO Control Core, which then distributes the bytes to each corresponding core through the internal Network-on-Chip (NoC). This bootloader program resides on a special region of the IO Control Core's flash memory, while the rest of the memory are available for the program being flashed, hence they are not interfering. As for the other cores, the flash memory is fully available for the program being flashed. This process is performed based on the RUMPS401 software-bootloader protocol (Hartono, 2014). Figure 4.1 depicts the RUMPS401 software development process.

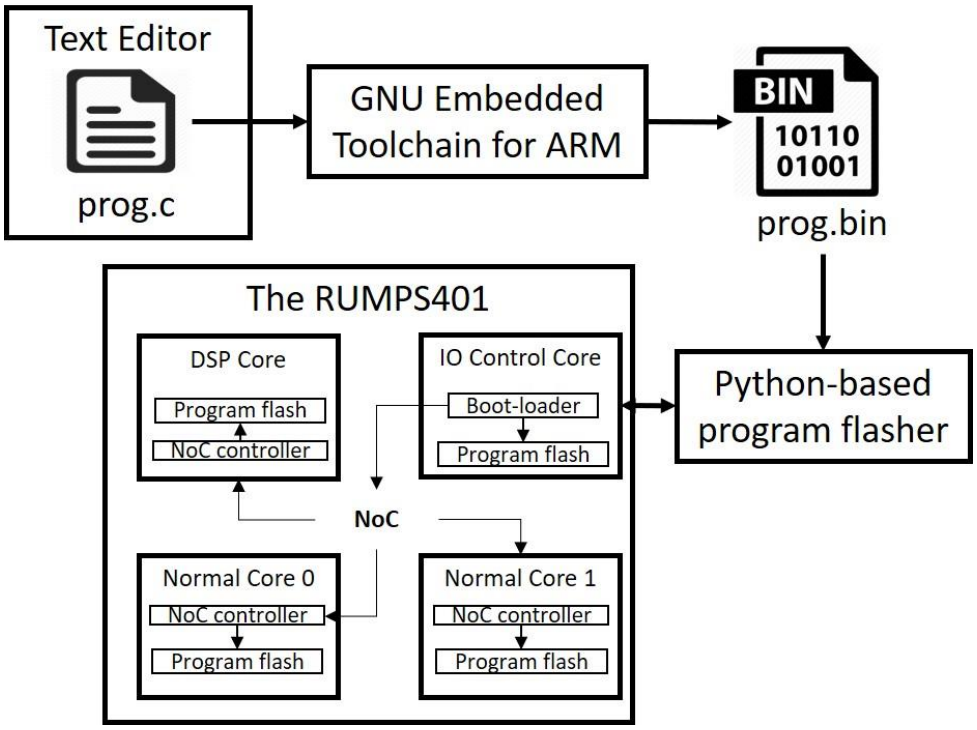


Figure 4.1 The RUMPS401 Program Development Setup

During the development process, two Python-based programs were written to test the encoder and decoder software functionalities separately. The programs run on a PC and communicate with the RUMPS401 via the UART

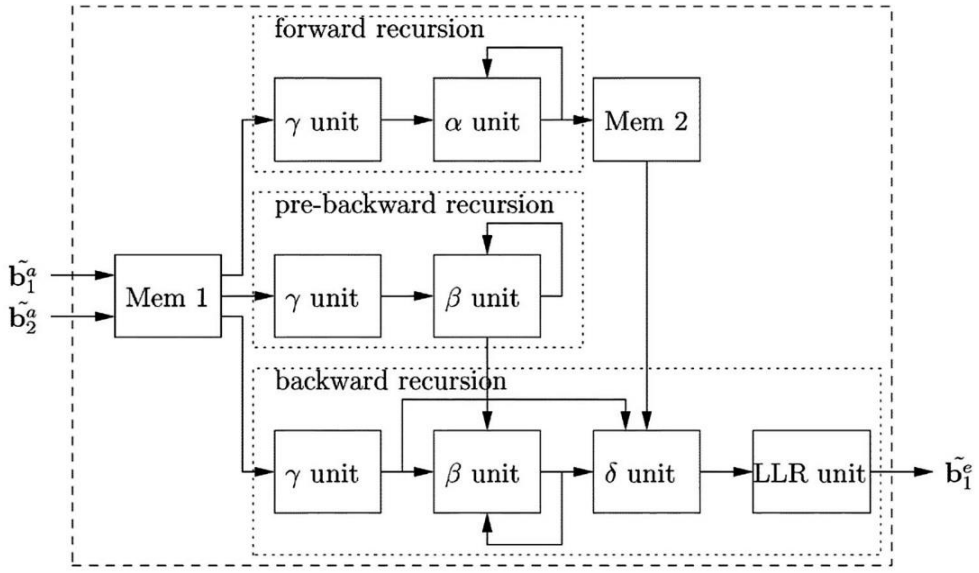
interface. The encoder-testing program randomizes several pairs of unencoded frame and its corresponding codeword, drives the unencoded frames into the RUMPS401 which then encodes those frames and send the resulting codeword back to the Python program. The codeword sent back by the RUMPS401 is checked against the pre-computed codeword, where they should match. Inversely, the decoder is tested by driving codewords into the RUMPS401 which decodes and returns the results back to the Python program. The decoding results is then compared against the unencoded frames. These functionality tests are performed without noisy data sets, i.e. codewords driven into the decoder software still contains its original value of either 1 or 0. The decoder will be tested against noisy data later in the full system test, after integration with the coherent-BPSK modulation and demodulation.

4.1.2 Turbo Decoder Parallelism and Task Allocation

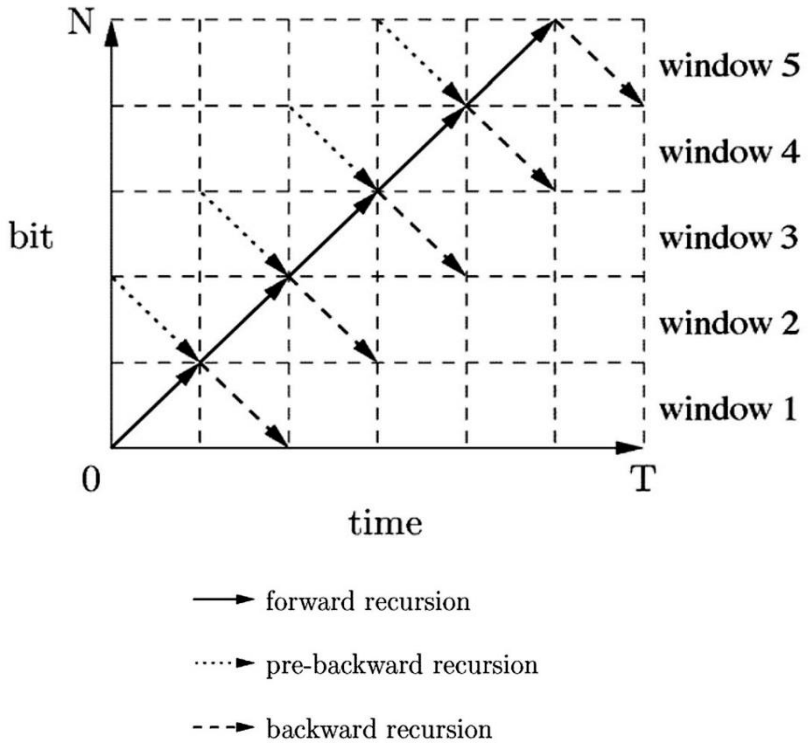
It was shown in Chapter Two that previous works achieved practical implementation of Turbo Code especially the decoder through the design of highly parallel system and simplification of the algorithm. A previous work (Li, Maunder, Al-Hashimi, & Hanzo, 2013) particularly highlighted two interesting parallelism schemes for an Application Specific Integrated Circuit (ASIC)-based Turbo decoder hardware. Both schemes decompose the Bahl-Cocke-Jeinek-Raviv (BCJR) algorithm into several operations to be performed in parallel. The first scheme parallelized the algorithm based on the type of operation, i.e. metrics corresponding to a specific state and timestamp k are computed in parallel. Meanwhile, the second scheme employs a data-centric parallelization where only one type of operation is performed at a time for

multiple data, i.e. forward state metric for one timestamp k is performed for multiple states. For example, referring to the decoder trellis diagram in Figure 3.8 and assuming the decoding for data at timestamp $k = 2$, the first scheme may compute several metrics (branch, forward state, or reverse state) corresponding to a single state in parallel at once, while the second scheme computes one metric (branch metric for example) for four states at once.

Figure 4.2 depicts the first parallelization scheme's decoder architecture and the operation timeline. There are three separated processing lines, one for forward state metric computation of the current window, one for backward state metric computation of the next window, and the last one for backward state metric along with likelihood ratio (LLR) of the current window. The term 'recursion' is often used to indicate the metrics recursive computation. The pre-backward state metric is computed to assist with the initialization of the current window's backward metric. It has the exact same function as the guard window in the sliding window algorithm - that is to bring up extra information from the next window to the current window, hence lowering the error rate. By using this decoder, three windows can be processed at once in a pipelined fashion assuming there are sufficient data arriving at the receiver. The pipelined process starts from the pre-backward recursion, followed by the forward recursion, then the backward recursion, and ends by computing the likelihood ratio. It is shown in Figure 4.2 that at most three windows can be process in parallel by the three recursions, except for the first timestamp where the backward recursion is not performed since the preceding recursions for the first window is yet to finish.



(a) Hardware Structure



(b) Window-based parallelization

Figure 4.2 Task-centric BCJR Parallelization (Li, Maunder, Al-Hashimi, & Hanzo, 2013)

While the first scheme improves the decoding throughput by processing multiple windows at once, it is evident that such architecture imposes a lot of

hardware redundancy. There is a dedicated branch metric γ computation unit for each recursion line, and similarly for the pre-backward and backward recursion line with their dedicated backward state β computation unit. This leads to the need for more hardware resources due to the multiple instances of the same unit, which ultimately increases the power consumption.

Furthermore, it is evident from Figure 4.2 that each processing line does not possess the same number of computation units. While this number does not directly reflect the computational load of each processing line, from the BCJR algorithm breakdown discussed in Chapter Three it is obvious that the backward recursion line requires the longest time to complete. Although the three processing lines can carry parallel computation, still outputs of each line must be chained serially to ultimately produce the likelihood ratio. This means that the forward and pre-backward recursion lines must wait for the backward recursion line to finish before moving on to the next window. Provided that such decoder can be clocked at a high enough frequency, the waiting interval might be negligible. However, that is not good for power consumption. The faster processing lines stay idle waiting for the longer process to finish while still consuming power. An intelligent power management controller may be employed but it imposes additional complexity which may reduce the decoding throughput.

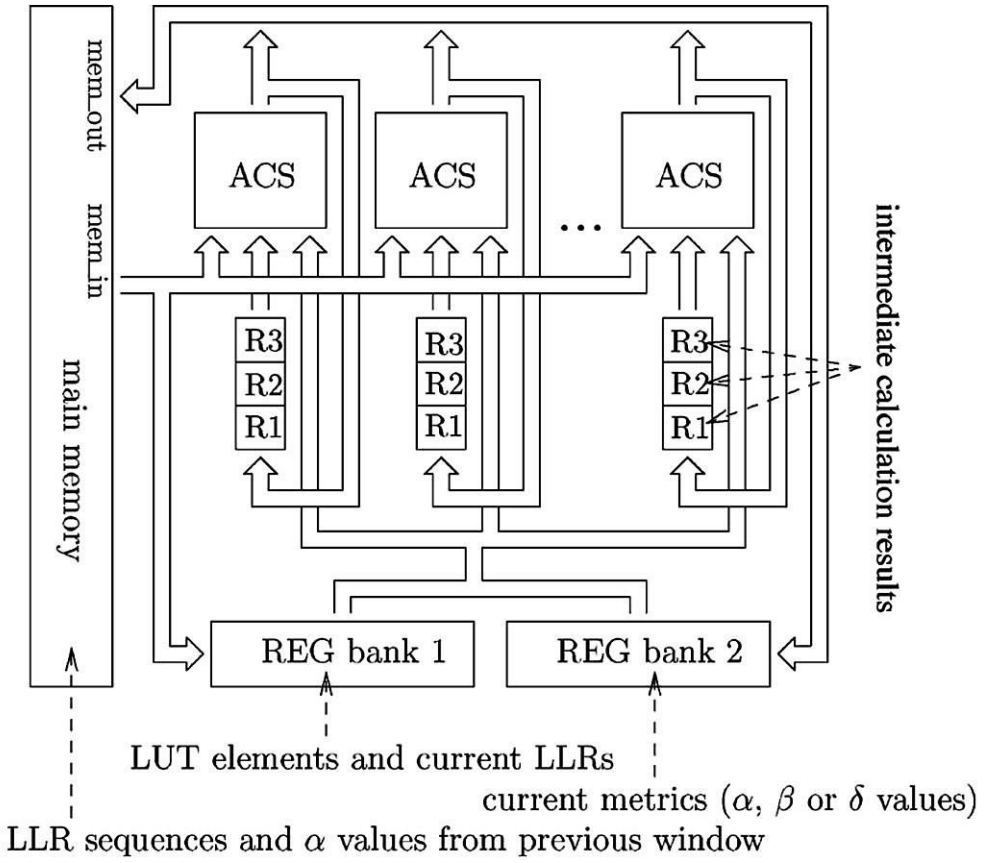


Figure 4.3 Data-centric BCJR Parallelization (Li, Maunder, Al-Hashimi, & Hanzo, 2013)

Li et al. (2013) proposes a solution by introducing the second parallelism scheme, which was briefly explained earlier as a data-centric parallelization. It tackles both the duplicate units and uneven processing time problems by employing a group of m identical Add-Compare-Select (ACS) units where m is the number of the encoder states, as shown in Figure 4.3. It was discussed that the BCJR algorithm can be represented by three metrics and likelihood ratio computations, which is performed for every bit received, and for every encoder trellis state and transition associated with the bit. This second scheme decomposes each of those computations further into sequence of simple operations that can be performed by a single ACS unit. The group of these units thus would be able to perform the same type of computation over m number of

states or transitions. Since every BCJR computation can be performed with the same unit, the second scheme does not require duplicate computation units as in the first scheme which increases area and power consumption. By performing only one type of operation at one time over multiple identical units, the execution time of each unit will be similar with slight deviation due to the electrical properties, which is negligible. Hence, there is no time wasted on waiting on other processing lines to finish.

Although the two schemes discussed were hardware-based implementation of Turbo Code, the parallelism concept can be adopted into software-based implementation. While the original paper (Li, Maunder, Al-Hashimi, & Hanzo, 2013) clearly favors the second scheme due to better power usage, its adoption into software must be reevaluated properly. The second parallelism scheme stresses on the use of multiple identical units whose number scales accordingly to the number of encoder states. This kind of implementation may pose a problem to software-based Turbo Code. Software-based implementation typically employs ready-to-use Multi-Processor System-on-Chip (MPSoC) whose number of processing elements is fixed and may not match the number of encoder states. Additionally, the MPSoC may employ heterogeneous architecture where each processing elements are different in terms of resources and processing speed. In such a system, the first parallelism scheme is more suitable. However, there are also homogeneous MPSoC systems that employ a large number of uniform processing units which is fit for the second scheme. Hence, both parallelism scheme may find their own adoption in a software-based Turbo Code system depending on the MPSoC being used.

In terms of computing resources, the RUMPS401 can be considered as a heterogeneous system due to the different hardware accelerator possessed by each core. The DSP Core clearly excels in complex mathematical computations due to its Multiply-Accumulate (MAC) hardware. Moreover, this work implements the UMTS Turbo Code standard whose number of states is eight, while the RUMPS401 has only four cores. This obviously limits the adoption of the data-centric parallelization scheme. Hence, the first scheme which is an operation-centric parallelization is adopted in this work. The Turbo Code implementation thus revolves on the proper task breakdown, simplification, and assignment to each of the RUMPS401 cores.

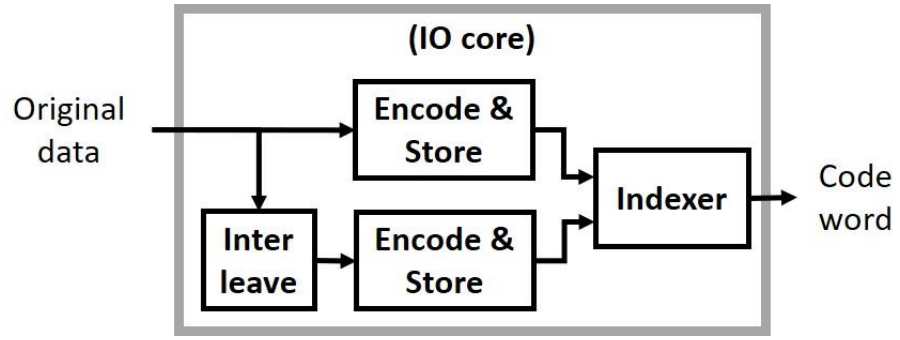
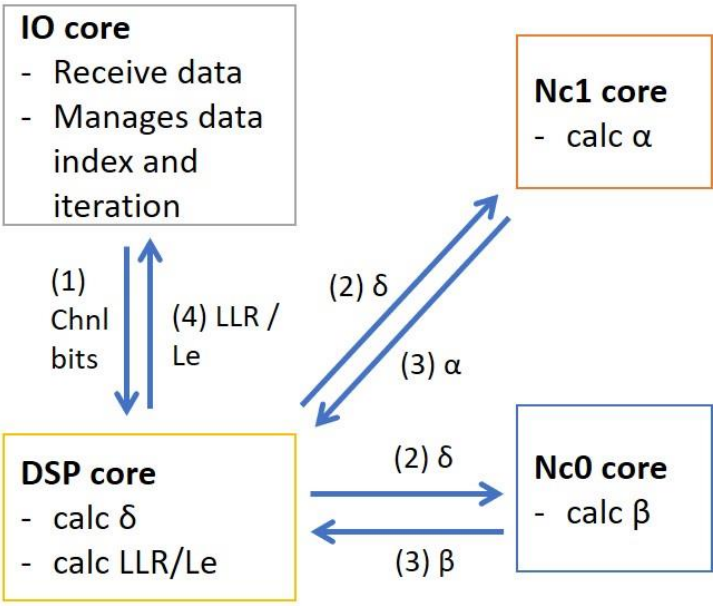


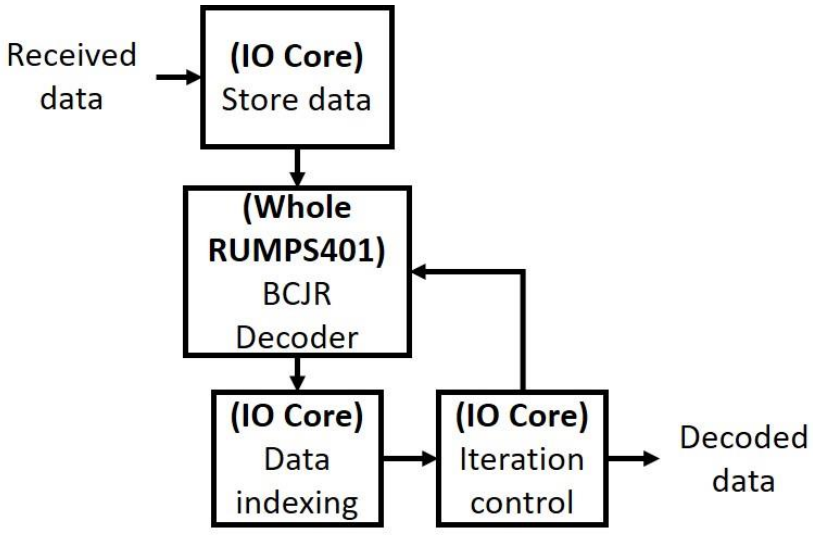
Figure 4.4 The RUMPS401 Encoder Implementation

Figure 4.4 depicts the Turbo encoder software implementation on the RUMPS401, which is entirely deployed in the IO Control Core. As discussed in the previous chapter, the Turbo encoder consists of two identical convolutional encoders whose operation can be represented by the trellis diagram. The encoder thus can be implemented by a simple lookup table consisting of every combination of current encoder state and input bit, along with the corresponding output and state transition. During the encoding process,

the software can simply use the table to produce the output codeword and track the state transition.



(a) BCJR Decoder Implementation



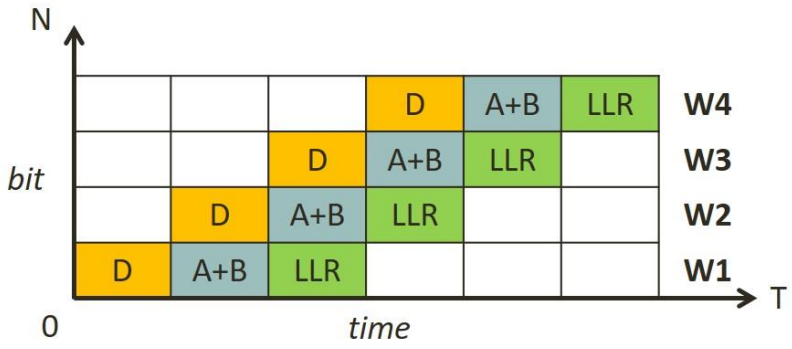
(b) Complete Turbo Decoder Implementation

Figure 4.5 The RUMPS401 Decoder Implementation

Since the two convolutional encoders are parallel and operates independently, they can be implemented either in one or two cores. A single

core implementation is chosen based on two reasons. First, the dual core implementation requires additional time for exchanging the data frame and the resulting codeword via the NoC, while on single core implementation those data are retained in a local memory and can be accessed rapidly. Encoding is a very simple lookup function that can be performed quickly, thus the overhead introduced by NoC access may negate the improvement made by the parallel encoding.

Similarly, the DRP interleaving function is also a simple operation whose parallel deployment will not provide significant improvement. Secondly, the Turbo Code complexity lies on its decoding process which as discussed in previous chapters, is the target of the optimization process for practical implementation. The IO Control Core first operates as the first convolutional encoder with the original data frame, save the output codeword, then operates as the second encoder with the interleaved data frame and again save the resulting codeword. Codeword from both encoders are then arranged into the final Turbo encoder's codeword in a similar manner to the Matlab model shown in Figure 3.14.



(a) General Sliding-Window Parallelization Idea

IO	Recv from chnl			Store LLR 32b(1)		Store LLR 32b(2)		...
		Send 32b(1)	Send 32b(2)		Send 32b(3)		Send 32b(4)	...
DSP		Calc d 32b(1)	Calc d 32b(2)		Calc d 32b(3)		Calc d 32b(4)	...
		Send d 32b(1)	Send d 32b(2)	Calc LLR 32b(1)	Send d 32b(3)	Calc LLR 32b(2)	Send d 32b(4)	
NC0			Calc & send b 32b(1)		Calc & send b 32b(2)		Calc & send b 32b(3)	...
NC1			Calc & send a 32b(1)		Calc & send a 32b(2)		Calc & send a 32b(3)	...

(b) The RUMPS401 Sliding-Window Parallelization

Figure 4.6 Sliding Window-based Parallelism

Since the Turbo decoder consists of two identical decoders that are serially chained, it can be implemented with only one actual BCJR decoder along with a control system for managing data interleaving and transfer between the two decoders. As shown in Figure 4.5, the BCJR decoder software is implemented in all of the RUMPS401 cores. The BCJR decoding is divided into four tasks, that is the three metrics and the likelihood ratio computations. Along

with the control system, these four computations are distributed into the four cores which then perform the tasks in parallel. The IO Control Core stores the received frame, splits it into smaller windows, and send them to the DSP Core. It also keeps track on the number of iterative loops performed along with the likelihood ratio computed by each decoder. The DSP Core then cooperates with the Normal Cores to perform the BCJR decoding algorithm and returns the results back to the IO Control Core. The branch metric and likelihood ratio calculation are performed by the DSP Core, while the forward and reverse state metrics are calculated by the Normal Cores. All data transfers among cores are performed via the NoC. Note that the task distribution shown in Figure 4.5 is at the initial stage, and is refined through a detailed optimization process covered in the following sub-chapter.

Figure 4.6a illustrates the idea of parallelism for these computations. The windowed data are fed into a pipeline. Its ordered computation stages are the branch metric, the forward and reverse state metric, and the likelihood ratio. The actual implementation of the sliding window-based parallelism on the RUMPS401 is depicted in Figure 4.6b. Each group of rows labelled with the RUMPS401 core names represents tasks performed by that specific core, whilst each column represents the timestamp. Notice that in the actual implementation there are only two windows being processed in parallel, while the original idea could process three windows at a time. This is evident in Figure 4.6b where the third window data bits are not sent to the DSP Core before it returns the likelihood ratio values. Similar thing happens with the other windows, such as the fourth window which is not sent until the second window is fully decoded.

This limitation is imposed by the RUMPS401 memory resources which shall be discussed in detail in the next section.

4.1.3 Decoder Memory Usage

Regardless of the target hardware, PC, mobile phones, or embedded controller, any proper software development would carefully consider the software performance and the resources it takes up. Every real system has limited amount of memory, and the software should be developed with this in mind. This is especially the case for low power MPSoC such as the RUMPS401 where the Random-Access Memory (RAM) is integrated and very limited in size. As described in Chapter two, the RUMPS401 has four cores with their own set of Static RAM (SRAM) and Flash memory, sized at 8KB and 32KB respectively. The Flash memory stores the software binary and necessary large-sized lookup tables, while the SRAM stores the software variables.

The Turbo Code being implemented in this work employs two identical convolutional encoders each with three memory elements, which means that it has eight states. The Turbo encoder's total rate is 1/3, encoding the original frame of 256 bits into codeword of 768 bits. Sliding window BCJR decoder is employed with a window size of 32 bits and no guard window, and processes at most two windows in parallel. Based on these configurations and tasks distribution in Figure 4.5, the numbers of variables required by the BCJR decoder are listed in Table 4.1.

Table 4.1 RUMPS401 BCJR Decoder Variables List

The RUMPS401 Core	Variable	Required number	Description
IO Core	> Received codeword	768	3 times the original frame
	> LLR value	256	Likelihood ratio for each bit in received frame
DSP Core	> Branch metrics	2 x 16 x 32	2 windows, each state has two possible transitions
	> Forward state metrics	1 x 8 x 32	1 window, each timestamp has eight states
	> Reverse state metrics	1 x 8 x 32	Same as above
Normal Core 0	> Branch metrics	1 x 16 x 32	1 window
	> Reverse state metrics	1 x 8 x 32	1 window
Normal Core 1	> Branch metrics	1 x 16 x 32	1 window
	> Forward state metrics	1 x 8 x 32	1 window

For the decoder functionality, the IO Control Core must allocate two one-dimensional arrays for the received codeword and the likelihood ratio values from each decoding iteration. It separates and deinterleave the received codeword of rate 1/3 into two codewords of rate 1/2 for the two BCJR decoders. The properly sequenced data then is split per window size and passed on to the DSP Core, which computes the branch metric. Note that there are redundancies in the arrays of metric spanned across the DSP and Normal Cores. The Normal Cores require the branch metric for state metrics calculation, while the DSP Core requires the state metrics for likelihood ratio computation. Each core receives the necessary metrics from the other cores and keeps those as a local copy in their own SRAM. This method allows faster decoding speed since accessing local SRAM is much faster compared to querying through NoC, at the expense of more memory resources. However, this redundancy does not apply to the received codeword. The DSP Cores does not keep a local copy of the windowed codeword, since it is only needed for branch metric computation.

The whole decoding process on the RUMPS401 is performed using fixed-point arithmetic with S16.15 number format - a 32-bit signed fixed-point data type, 16 integer bits and 15 fractional bits. It supports an integer value from -65536 to +65536 and decimal precision of 3×10^{-5}. This fixed-point data type is defined as part of an extension added to the GNU Compiler Collection (GCC) for supporting fixed-point operation on embedded software. The documentation for this fixed-point extension is described in the International Organization for Standardization (ISO) document with reference number ISO/IEC TR 18037 (Standardization, 2006).

```
64bit_result = MAC_MUL(32bit_opA, 32bit_opB)
Bit right-shift 64bit_result by 15 bits
```

Figure 4.7 Fixed-point Multiplication

The extension is ready-to-use, providing necessary functionalities of fixed-point data type such as floating-point to fixed-point conversion, addition, multiplication, division. These functionalities are utilized by the decoder software, except for the multiplication. Since the software intends to maximize the use of the RUMPS401 hardware accelerator, the fixed-point multiplication function is implemented using the single cycle multiplication of the MAC hardware. The addition does not require hardware acceleration, and there is no hardware accelerator for division in the RUMPS401. Figure 4.7 shows the multiplication function's pseudocode. It treats the 32-bit signed fixed-point operands as 32-bit integers and performs integer multiplication using the MAC hardware. Bit-shifting is applied to the multiplication result to adjust its decimal point, which then is treated as a signed fixed-point data type again.

Table 4.2 RUMPS401 BCJR Decoder Memory Requirement

The RUMPS401 Core	Variable	Required number	Actual size in bytes
IO Core	> Received codeword	768	3072 bytes
	> LLR value	256	1024 bytes
DSP Core	> Branch metrics	2 x 16 x 32	4096 bytes
	> Forward state metrics	1 x 8 x 32	1024 bytes
	> Reverse state metrics	1 x 8 x 32	1024 bytes
Normal Core 0	> Branch metrics	1 x 16 x 32	2048 bytes
	> Reverse state metrics	1 x 8 x 32	1024 bytes
Normal Core 1	> Branch metrics	1 x 16 x 32	2048 bytes
	> Forward state metrics	1 x 8 x 32	1024 bytes

Since the whole decoding operation is performed with 32-bits data type, the Table 4.1 can be amended with the exact value of required memory in bytes, shown in Table 4.2. It is evident that the BCJR algorithm consumes relatively huge amount of memory considering that the RUMPS401 only has 8KB of SRAM per core. The IO Control Core has half of its SRAM consumed for storing the received codeword and decoding result, while the DSP Core has 6KB consumed for parallel processing of only two windows. Note that while the DSP Core stores branch metrics of two windows, it only stores state metrics for one window. Once the DSP Core receives the state metrics, it sends the next window's branch metrics to the Normal Cores then computes the likelihood ratio immediately. Therefore, the current window's state metrics are only necessary until the likelihood ratio computation is finished.

Should the software arrange parallel computation of three windows, for the branch metric alone the DSP Core requires extra $1 \times 16 \times 32$ variables for

the additional window, which is an extra 2048 bytes of memory. This added to the 6KB usage would lead to memory overflow in the DSP Core, rendering the software unable to run. A possible alternative is storing these variables in the flash memory, but this would slow down the overall processing speed due to the much slower access to flash memory.

The same applies to the alternative of storing some of the DSP Core arrays in the Normal Cores SRAMs, due to the additional overhead for accessing the NoC coupled with the control complexity of managing separated variables database. Moreover, the variables retrieval process limits the parallelism, i.e. DSP Core will not be able to compute before the Normal Cores send the required variables. The Normal Cores may be performing other computation. Hence, the decoder can only process two windows in parallel while still leaving memory space for another essential part of the complete wireless system, the coherent-BPSK demodulation.

4.2 Turbo Decoder Software Optimization for the RUMPS401

Based on the initial structure described in Figure 4.5, the decoder software is speed-optimized by analyzing each of its essential tasks execution time. A Register-Transfer-Level (RTL) model of the RUMPS401 (Hartono, 2014; Lim, 2015) runs the decoder software in a cycle-accurate simulation, allowing detailed logging of each tasks execution time. As discussed in previous chapters, improvements are made by adhering to the idea of parallel processing and simplification of the tasks. The parallel processing stresses on distributing the tasks evenly among the cores, i.e. if a certain core is assigned with a task

that takes long time to complete, the task should be split and distributed among the cores. Simplification takes place on tasks with complex mathematical operation which is either not supported by the RUMPS401 or inefficient when performed on the RUMPS401.

The tasks being observed are the computation of the branch metric, the forward state metric, the reverse state metric, and the likelihood ratio. The NoC transfer of branch and state metrics are also analyzed, while the transfer of the likelihood ratio values is not analyzed as its volume is insignificant compared to the three metrics. Recall from the BCJR decoder trellis diagram in Figure 3.8, for a single timestamp k there are $2n$ branch metrics, n forward state metrics, and n reverse state metrics, where n is the number of states. In comparison, there is only one likelihood ratio for a single timestamp. Being specific to the code structure used in this work, there are only 256 NoC transfers of the likelihood ratio values for a single frame, compared to 256 forward state metric transfers for a single window, or 512 transfers of the branch metric.

Figure 4.8 illustrates the initial task distribution of BCJR decoding and communication among the RUMPS401 cores. For convenience sake, changes made to the software during the optimization are tracked and referred by a version number in the format of 1.X. The first working software is versioned as 1.0, any significant change to the software is marked as version 1.1, 1.2, and so on. Since the Turbo decoder software development and optimization process is performed on data sets without noise, the noise variance estimation is not performed either. Instead, the BCJR decoding is performed with noise variance value set to $No = 1$. Recall from the BCJR algorithm equations in Chapter

Three, the noise variance value is necessary to the branch metric and likelihood ratio computations thus cannot be omitted. The noise variance estimation is implemented during the integration with the coherent-BPSK demodulation which works on real signals and shall be covered in Chapter Five.

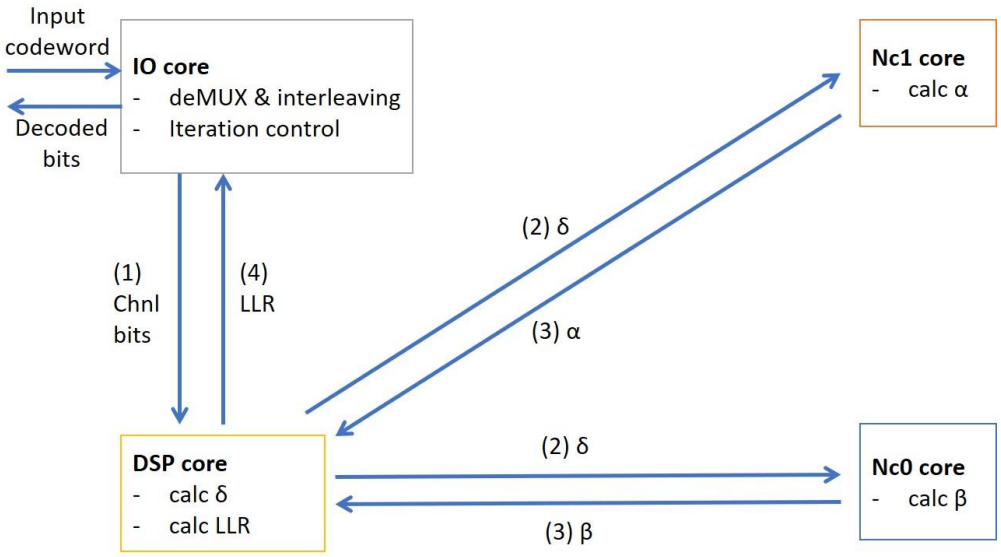


Figure 4.8 Decoder Software 1.0's Structure

This initial software structure in software version 1.0 is similar to the general idea shown in Figure 4.5. The IO Control Core performs demultiplexing to divide the incoming codeword of rate 1/3 into two codewords of rate 1/2 for each decoder, and tracks the number of decoding result. Since the branch metric and likelihood ratio computations include multiplications, they are loaded to the DSP Core. The forward and backward state metrics are computed by the two Normal Cores. The metrics and likelihood ratio computations implemented in the RUMPS401 are based on the Max-Log-BCJR algorithm formula presented in equations (3.2a) to (3.2d). Every communication in and out of the RUMPS401 is performed by the IO Control Core.

Presented in Figure 4.9 is the detailed graph of the tasks execution time in software version 1.0 in comparison to version 1.1. The execution time is measured in milliseconds, and represents the time taken by the respective core to perform certain task per timestamp k. Task `calc_d` represents the sixteen branch metric calculations, two for each of the eight decoder states. Task `calc_a` represents the eight forward state metric calculations, one for each of the eight decoder states. Task `calc_b` is similar to task `calc_a` except that it performs the reverse state metric calculations. Task `calc_llr` represents the likelihood ratio calculation for a single timestamp. Task `trans_a`, `trans_b`, and `trans_d` each represents the NoC transfer of forward state, reverse state, and branch metrics, respectively. Consequent task load figures are presented in the same way.

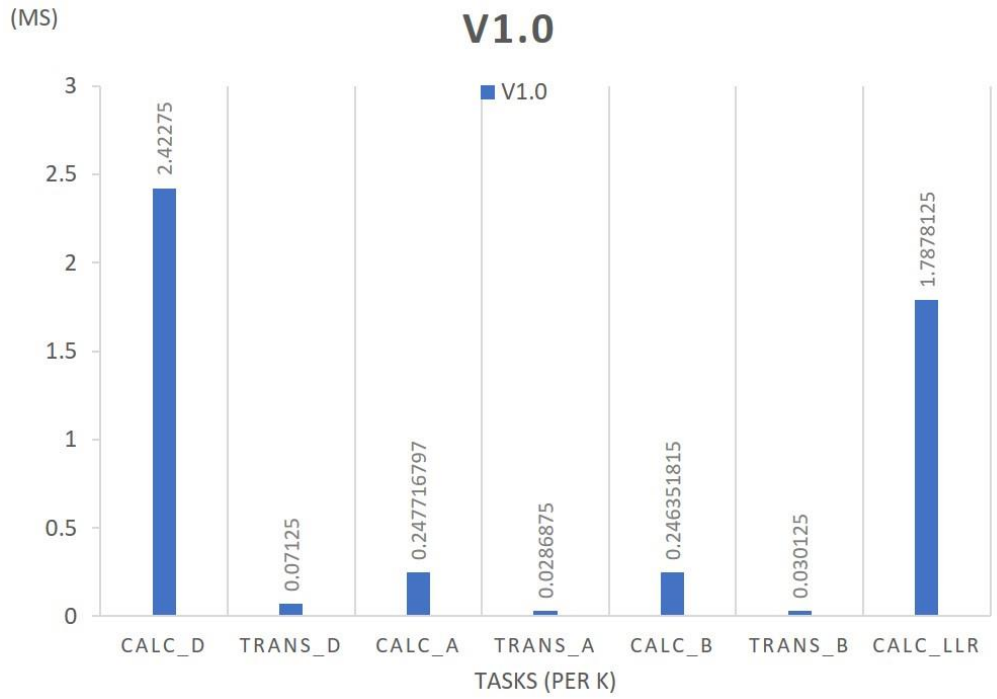


Figure 4.9 Task Load - Software Version 1.0

Analyzing the tasks load graph of software version 1.0, it is straightforward that the NoC transfers of each metric only consumes tiny portion of time compared to the calculation part. Execution time of the NoC transfer tasks are consistent, shown by the sixteen transfers of branch metric taking twice as long as the eight transfers of forward state metric or reverse state metric. It is evident that the computations of the branch metric and the likelihood ratio are much slower compared to the forward and reverse state metrics computations. Each of the state metrics takes around 0.24ms to finish, while the branch metric and likelihood ratio computation finishes in around 2.42ms and 1.78ms, respectively. That is tenfold and sevenfold of the state metrics computation time. Coupled with the fact that both computations are performed by the DSP Core, it can be directly inferred that a performance bottleneck happens in the DSP Core. The total time taken by the Turbo decoder to process one frame for one iteration on this version is 1.88 seconds.

It was discussed in previous sections that the branch metric and likelihood ratio calculation tasks are loaded to the DSP Core due to both tasks involving multiplications. The ARM Cortex-M0 processor used in the RUMPS401 cores possesses hardware multiplier which only able to perform multiplication over 32 cycles, and this is mitigated by utilizing the single-cycle MAC hardware in the DSP Core. However, both the branch metric and likelihood ratio calculations also involve division operation, which is not supported by either the ARM Cortex-M0 native hardware or the RUMPS401 hardware accelerator. Moreover, the branch metric calculation also involves logarithm operations which has no hardware support. The lack of hardware

implementation for these time-critical functions forces implementation via software, which naturally is not as efficient as specialized hardware block. Mitigations for the division and logarithm operations are discussed separately, started by the division.

The division operation is implemented by using the standard routine provided by the ARM C and C++ Libraries (ARM, GNU ARM Embedded Toolchain, n.d.), whose execution cycle count depends on the input values and the division quotient. The longer the quotient, the longer it takes to finish a single division routine. For example, a division with 32-bit quotient might get as long as 96 cycles to complete, while a lower 4-bit quotient may require only 12 cycles to complete. However, considering the computation of the 32-bit fixed-point number in the BCJR algorithm, the quotient will tend to have more bits to preserve its accuracy, hence yielding more cycle count.

Mathematically, a division can be performed as a multiplication by simply taking the reciprocal of the divider as the multiplier, i.e. $\frac{a}{5}$ is computed as $a \times 0.2$. On most embedded processor, multiplication is usually faster than division, and especially so for the RUMPS401 due to the single-cycle multiplication by the MAC hardware. Depending on the scenario, this property can either reduces the number of division or actually making it worse. Assume a case where there are n number of divisions required. If every division is divided by a single constant, the constant's reciprocal can be precomputed and hardcoded into the software as a multiplication by another constant. This scenario is the most efficient since there is no division performed at all.

Similarly, if all the divisions are divided by a number which is variable but holds its value through all the divisions, the number's reciprocal can be calculated with a single division operation while the rest of the n divisions are replaced by multiplications. Should the variable only hold its value for $n/2$ number of divisions, while on the other half the variable holds different value, there are two divisions required in addition to the n multiplications. When the variable values are random over n numbers of division, there is no point in applying the reciprocal property since it would need n reciprocals and n multiplication, which is even worse than directly performing the n divisions.

The Turbo decoder software version 1.1 applies the simple mathematical property above to its branch metric and likelihood ratio calculation. It can be observed from the Max-Log-BCJR equations (3.2a) and (3.2e) that both calculations involved a division by the noise variance value. Due to the RUMPS401 low processing speed, the incoming frame must be completely received and stored before further processing. While this may seem like a disadvantage, the store-then-process mechanism offers a benefit that the division reciprocal property can be utilized.

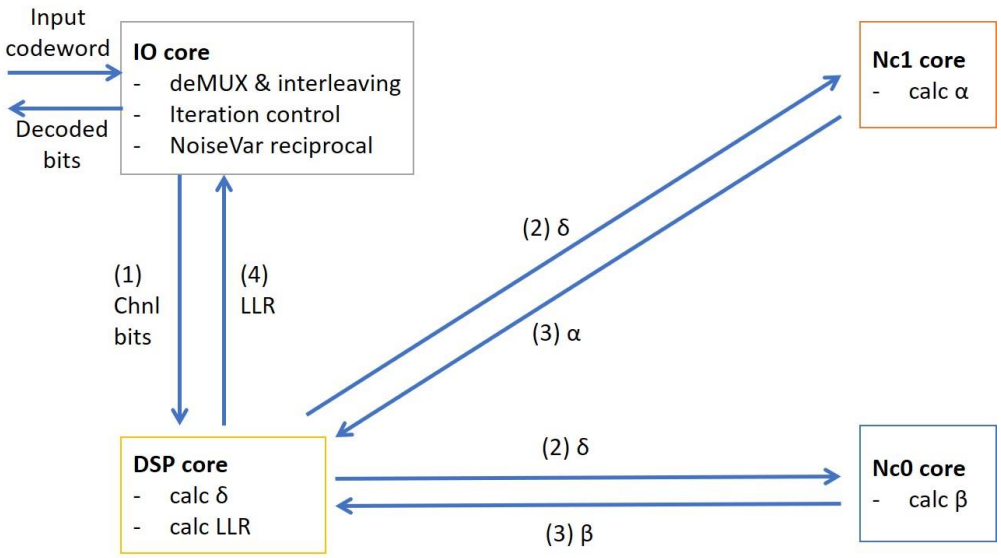


Figure 4.10 Decoder Software 1.1's Structure

The noise variance value is indispensable from the BCJR algorithm calculation and is obtained by finding variance of the incoming data in a frame. Should the software implementation choose to perform BCJR decoding while receiving data of a frame, the noise variance value would have variable values across the frame because the variance estimations are performed over different data sets. Thus, the branch metric computations at different timestamps are divided by different noise variance value. Assuming that the varying noise variance values still deviates in an acceptable margin without causing any loss in calculation accuracy, it still results in varying divider values which reduces the effectiveness of the division reciprocal property.

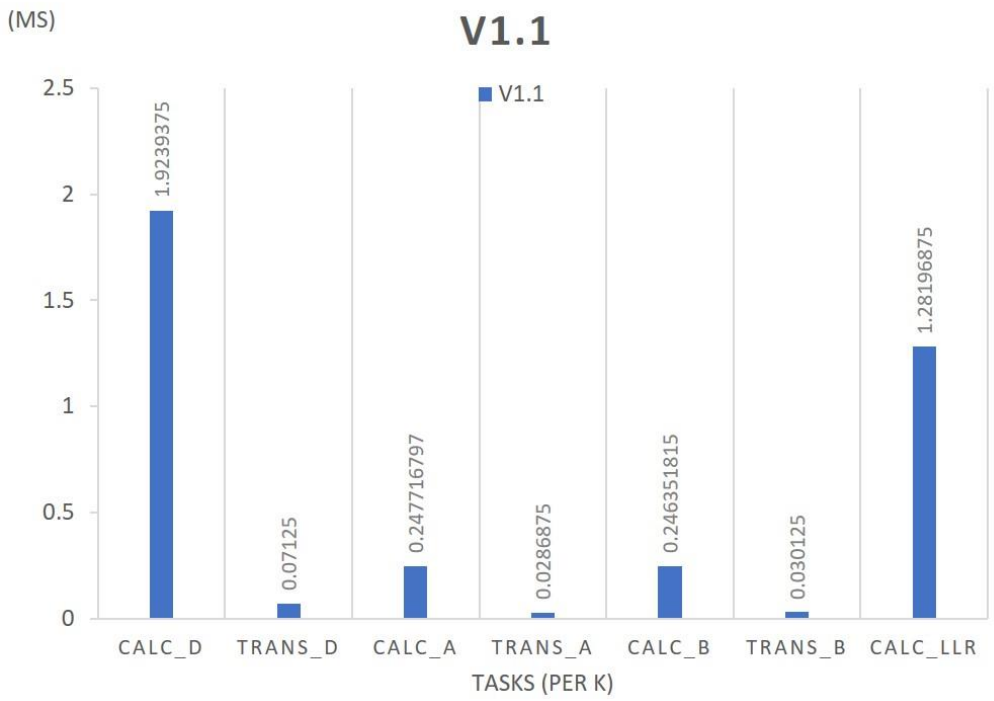


Figure 4.11 Task Load - Software Version 1.1

Hence, the RUMPS401 can utilize the store-then-process mechanism by estimating the noise variance over the whole received frame for just one time, then computes and stores the reciprocal of the noise variance. Later during the whole BCJR decoding process, every division by the noise variance is replaced by a single-cycle multiplication performed by the MAC hardware. This new task assignment is shown in Figure 4.10. Improvement made by applying this simple property is observable through the tasks execution time of software version 1.1 in Figure 4.11. The branch metric computation improves by twenty percent from 2.42ms to 1.92ms, while the likelihood ratio computation went twenty-eight percent faster from 1.78ms to 1.28ms. In total, the software version 1.1 cuts down 0.29 seconds from version 1.0, finishing the single iteration Turbo decoding in 1.59 seconds.

Despite the twenty and twenty-eight percent faster computations of the branch metric and likelihood ratio on software version 1.1, both computations are still significantly slower compared to the state metrics computation. It can be naturally inferred through equations (3.2b) and (3.2c) that state metrics fast computation stems from the absence of any complex operation beside addition and comparison. As discussed in the beginning of this chapter, the optimization focuses on balancing the task load assigned to each core. Further optimization on the branch metric and likelihood ratio computation hence is required.

As mentioned above, the branch metric calculation still involves logarithm function which must be implemented via software on the RUMPS401. On the other hand, the likelihood ratio still involves division by arbitrary number which is performed on the likelihood values sent to the other decoder. The likelihood ratio calculation resulted in a value π_k which is a ratio between the probability of the bit being 1 or 0. For this value to be utilized by the other decoder as a priori probability value, it must be split into two probability values instead of a ratio, that is the probability of the bit being 1, π_k^1, and probability of being 0, π_k^0. The conversion can be performed by equation (4.1), where the division by arbitrary number takes place.

$$\pi_k^1 = \frac{\pi_k}{1+\pi_k}; \quad \pi_k^0 = 1 - \pi_k^1 \qquad \textbf{(4.1)}$$

The logarithm function is approximated by an algorithm which only involves addition, bit shift, and lookup table (Owen). The key to this algorithm is the multiplication of ‘good’ numbers which are power of two such as 4, 16,

256, 3/2, 5/4. Any arbitrary number can be multiplied by these 'good' numbers via simple bit shifting operation, i.e. `(a<<2)` multiplies a by `2`, while `a+(a>>1)` multiplies `a` by `3/2`. Before finding the logarithm of an arbitrary number, the algorithm first prepares a precomputed lookup table which consists of pairs of a 'good' number and its logarithm value, as shown in Figure 4.12. The algorithm then prepares two variables, say `x` and `y`, initialized with the arbitrary number value and zero, respectively. There are only two operations performed in the algorithm. The first operation multiplies `x` by any number `k`, while the second operation substract `log(k)` from `y`. These two operations are performed repetitively to get `x` value as close as possible to 1. The pair of `k` and `log(k)` can be obtained from the precomputed lookup table prepared earlier. Since the lookup table consists of only 'good' numbers, the multiplication can be replaced with bit shift operation.

k	log(k)
16	2.7726
4	1.3863
2	0.6931
3/2	0.4055
5/4	0.2231

Figure 4.12 Logarithm Approximation Table

Since the logarithm approximation function does not contain any multiplication and division, it can be executed by other cores without any performance different compared to execution by the DSP Core. In software version 1.2 whose structure is shown in Figure 4.13, the logarithm computation is loaded into the IO Control Core. Referring to the branch metric computation

in equation (3.2a), the logarithm computation is performed on the a priori probabilities of the bit. These probabilities values are the likelihood result output by the other decoder, which is stored and indexed by the IO Control Core. Hence, it is best to let the IO Control Core computes the logarithm and send it to the DSP Core, along with the received data. Assigning the logarithm computation to the Normal Cores would be inefficient since the a priori probabilities must be sent to the Normal Cores first, introducing additional overhead by the NoC access and control complexity. Moreover, in both software version 1.0 and 1.1, after sending over received data to the DSP Core, the IO Control Core idles while waiting for the likelihood ratio value. This idle time can be utilized by computing the logarithm.

Figure 4.14 shows the tasks execution time of software version 1.2. It is evident that the logarithm function consumes a lot of time compared to the rest of branch metrics computation. The logarithm function itself consumes 1.51277ms to perform and cannot be split due to the repetitive and continuous process. Software version 1.2 improves the total decoding time significantly, by being able to decode a full frame in just 0.88 seconds, almost twice as fast as software version 1.1.

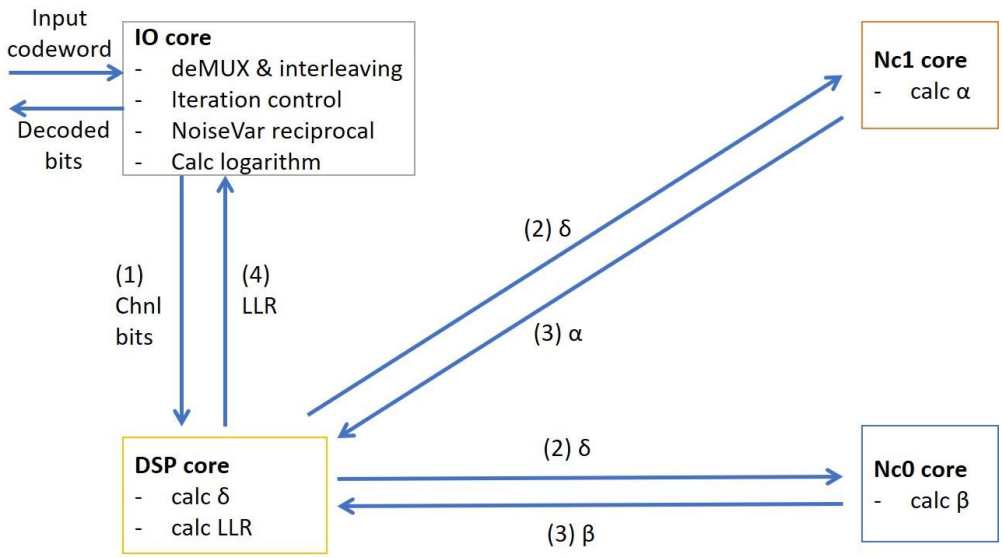


Figure 4.13 Decoder Software 1.2's Structure

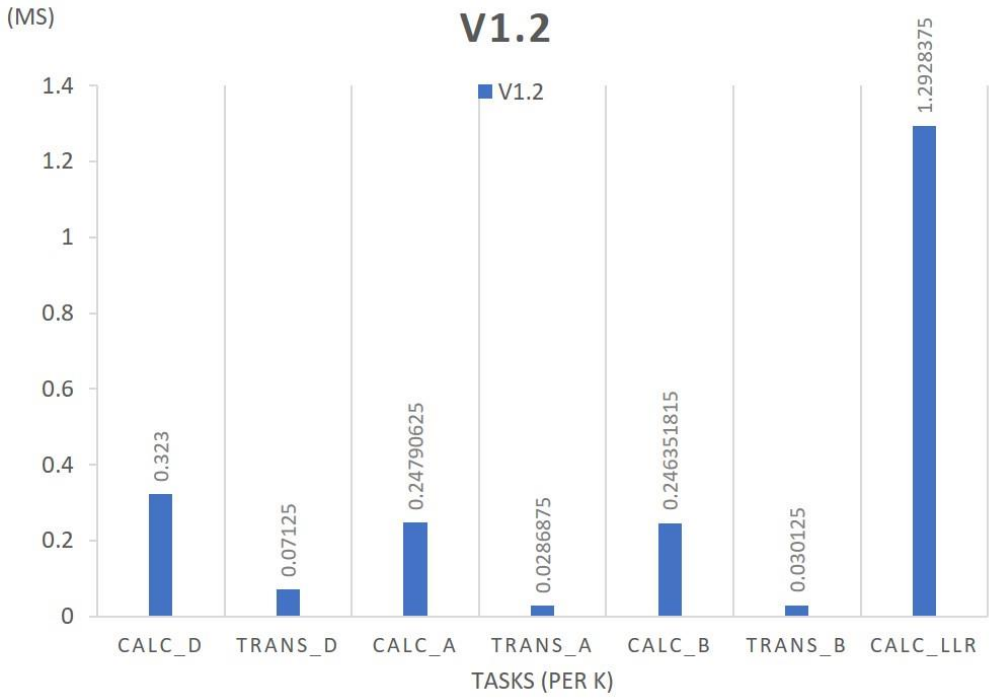


Figure 4.14 Task Load - Software Version 1.2

It was shown by the software version 1.2 that the basic idea of simple task splitting and distribution leads to a significant improvement to the total decoding speed. The same concept is applied to the division by arbitrary number

in the likelihood ratio conversion performed by the DSP Core. This change is applied in software version 1.3 whose structure shown in Figure 4.15, where the likelihood ratio conversion process is split from the likelihood ratio calculation and assigned to one of the Normal Cores. Earlier in software version 1.0 through 1.2, the decoding flow starts by the IO Control Core sending the received data and the corresponding a priori probabilities to the DSP Core. Branch metrics are computed and sent by the DSP Core to both Normal Cores, which then compute the state metrics and send the results back to the DSP Core. The decoding flow ends by the DSP Core computing and returning the likelihood ratio to the IO Control Core.

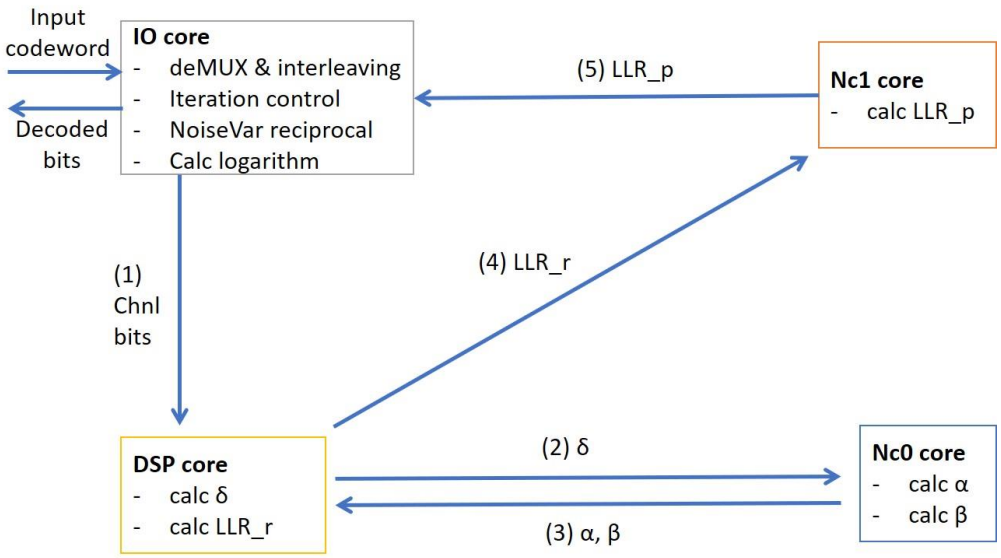


Figure 4.15 Decoder Software 1.3's Structure

In software version 1.3, there are changes involving the main tasks distribution among the cores and its data flow. The IO Control Core's and the DSP Core's tasks are not changed except that the DSP Core does not perform the likelihood ratio conversion. That task instead is assigned to the Normal Core

1. Both forward and reverse state metrics are now computed by the Normal Core 0 since they are simple and is still fast enough to be computed in serial. The decoding flow still starts with the IO Control Core sending the received data and the a priori probabilities to the DSP Core, which then computes and sends the branch metric to Normal Core 0. The Normal Core 0 computes both state metrics and send them back to the DSP Core, which then calculates the likelihood ratio. The Normal Core 1 receives the likelihood ratio from the DSP Core, converts the value, and ends the decoding process by sending the results to the IO Control Core.

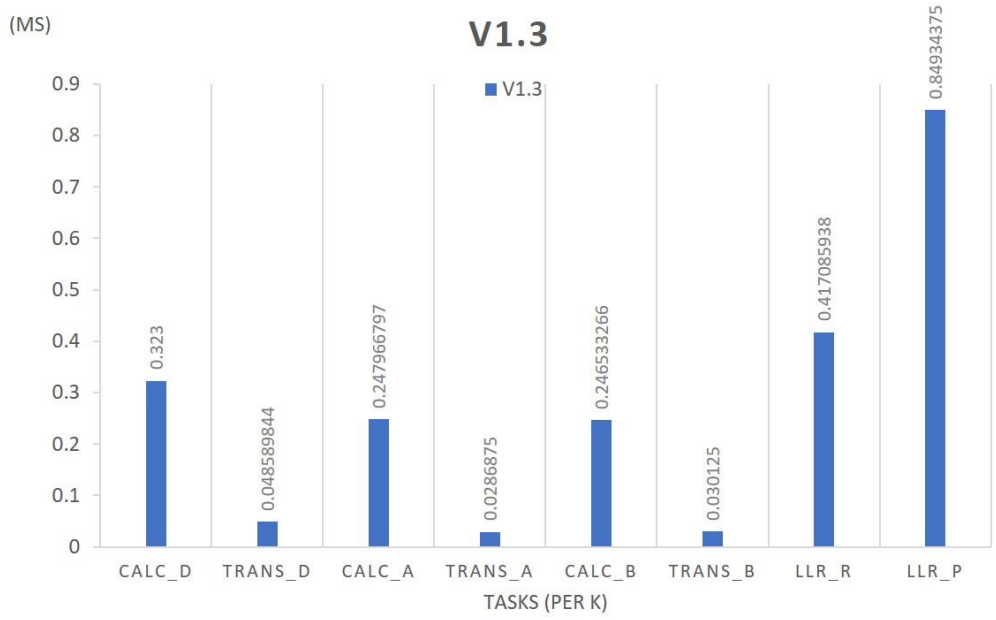


Figure 4.16 Task Load - Software Version 1.3

Figure 4.16 shows the tasks execution time of software version 1.3. The task `calc_llr` is split into two parts, the first part denoted `llr_r` is the likelihood ratio computation task, and the second part which solely perform the conversion from ratio to probability value denoted as task `llr_p`. It is evident

that the likelihood ratio conversion task llr_p imposes longer execution time of 0.84 seconds than the actual likelihood ratio computation which takes 0.41 seconds to perform. Software version 1.3 improves the total decoding time by 0.15 seconds, completing the decoding of single frame in 0.73 seconds.

It was clear from previous task load graphs that complex mathematical operations such as logarithm and division are relatively expensive to perform on such low power processor without specialized hardware accelerators. Approximation and simplification are applicable to these operations, but only to a certain degree, especially on software implementation. Similarly, reducing the number of those operations are possible only to a certain extent, as shown in the replacement of division operation by multiplication which is only effective if the dividers are the same. An alternative is to process those multiple operations in parallel.

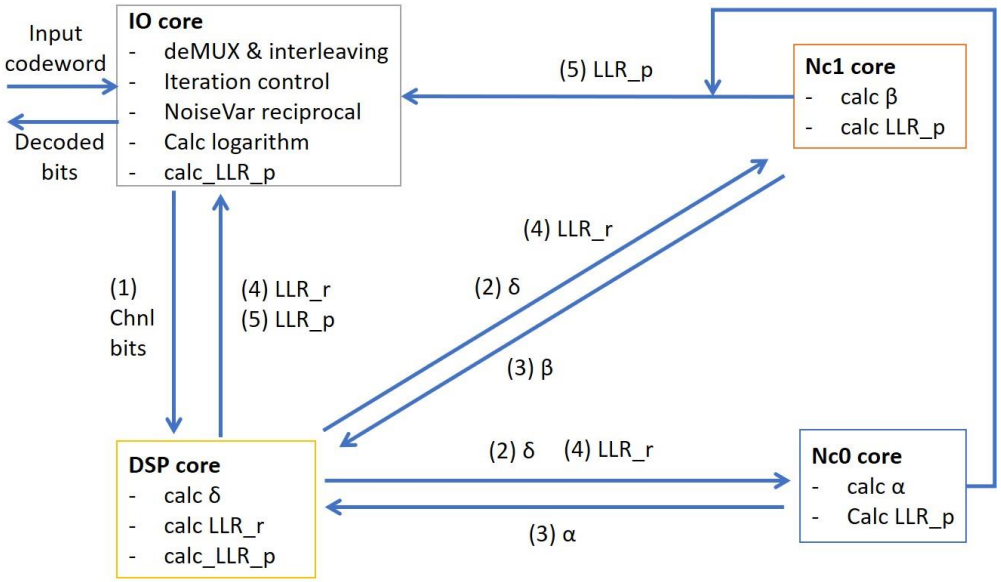


Figure 4.17 Decoder Software 1.4's Structure

Software version 1.4 is built based on the idea of distributing multiple complex operations to multiple cores for parallel processing. As shown by the software structure in Figure 4.17, the main tasks distribution is similar to version 1.2. The IO Control Core manages the data sequence and decoding iteration, along with the logarithm calculation to relief the calculation burden from the DSP Core. It also performs one divide operation to find the noise variance reciprocal. The DSP Core calculates the branch metric and likelihood ratio, while both Normal Cores calculate the forward and reverse state metrics. In this version, instead of dedicating one core solely for converting the likelihood ratio, the task is evenly distributed to each core.

The decoding flow in software version 1.4 is a mix between version 1.2 and 1.3. Decoding starts from the IO Control Core that sends out received data and a priori probability to the DSP Core, which then computes and sends the branch metric to the Normal Cores. Both Normal Cores compute the forward and reverse state metric and return the result to the DSP Core. The DSP Core computes the likelihood ratio, and evenly sends out the result to other cores and itself for the value conversion. Since the frame size is fixed at 256 bits, for a single BCJR decoding, in software version 1.4 each core only needs to perform sixty-four conversions, while in software version 1.3 all 256 conversions are performed by only one core.

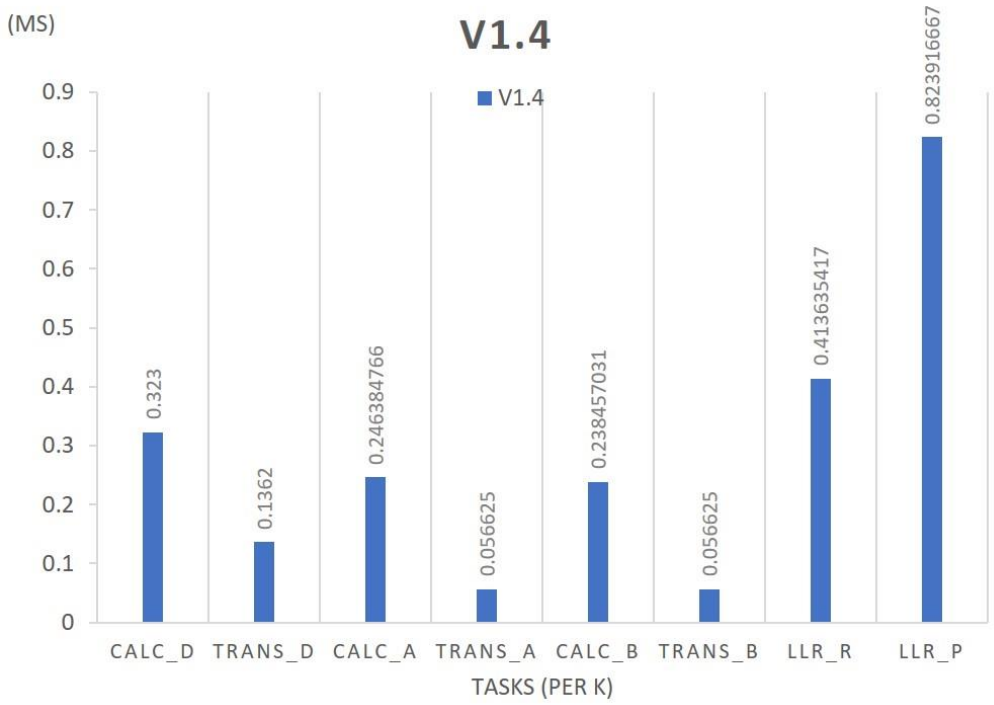


Figure 4.18 Task Load - Software Version 1.4

Software version 1.4 completes a full frame decoding in 0.98 seconds, which is 0.25 seconds slower than previous version. The individual tasks execution time of software version 1.4 is shown in Figure 4.18, which shows almost no difference compared to version 1.3. Since software version 1.4 only distributes the multiple tasks into each core without changing how the individual tasks are performed, it is straightforward that the execution time of each task will not be affected. However, the transfers of each metric are noticeably slower than previous versions, which is caused by the additional complexity of message passing among the cores.

Referring to the message passing of software version 1.0 in Figure 4.8, there is only one type of message sent from one core to another core. Data flow originating from the IO Control Core to the DSP Core only consists of the

received data, while the reverse flow only consists of the likelihood ratio. Similarly, the DSP Core sends only the branch metric to the Normal Cores, each then replies with either forward or reverse state metric. Similar message passing behavior can also be observed from software version 1.1 to 1.3, while in version 1.4 there are some communication links that consists of more than one type of message. For example, the DSP Core sends the IO Control Core both the likelihood ratio that has and has not been converted into its probability values. The DSP Core is also required to send the likelihood ratio to the Normal Cores in addition to the branch metric.

In software version 1.0 through 1.3, the message can be sent in a batch since each core knows what type of data to receive from another core, while software version 1.4 requires each message to be encapsulated with a packet identifier. Consider a situation where the IO Control Core receives data from the DSP Core. Without a packet identifier, the IO Control Core would not be able to differentiate whether the data is the likelihood ratio that has or has not been converted, hence disrupting its operation. Consider another situation of multiple windows processing. The first window is still being decoded and has reached the likelihood ratio calculation and conversion value, and at the same time the DSP Core has computed the second window's branch metric and need to send them to the Normal Cores for state metrics computation. The Normal Cores would need a packet identifier to differentiate whether the data being received from the DSP Core is the likelihood value or the branch metric.

The need for packet header thus introduces additional data overhead and additional control complexity for the message identifying process. In the

RUMPS401, every core has separated NoC buffers for receiving packet from other cores, whose size is limited to 8 packets, i.e. the IO Control Core has three separated NoC receive buffers for receiving packets from the DSP Core and the Normal Corers, and so does the others. In every software version built during this optimization process, acknowledgement scheme is implemented to avoid receive buffer overflow. A maximum of eight packets can be sent to a certain core before waiting for acknowledgement.

Recall that branch metric and state metric for a single timestamp consists of sixteen and eight values, respectively. In software versions 1.0 to 1.3 the branch metric can be simply split into two batches of transfers, while each state metric only requires one batch of transfer. Each batch consists of eight values of the corresponding metric. In software version 1.4, the branch metric must be split into four batches of transfer, each consisting of a packet identifier and four branch metric values. Similarly, the state metric must be split into two batches of transfer of the same size. Compared to previous versions, for the same number of values being sent, software version 1.4 requires four and two extra packets for the branch metric and each state metric, respectively. That is twenty-five percent data overhead which is quite significant.

Despite the arbitrary division being performed in parallel by all cores, its improvement is offset by the complex message passing scheme. In software version 1.0 and 1.3, since every core is assigned with a specific task, the metrics or data required by a certain core for computation can be sent by other cores consecutively, i.e. the DSP Core sends sixteen branch metric values in two consecutive batches with only a single wait for the acknowledgement in

between. On the other hand, in software version 1.4 these batches are not sent consecutively which would cause performance drop. Consider a case where the DSP Core sends out a part of branch metric for the second window, while the likelihood ratio conversion of the first window is still being performed by the Normal Cores. Since the scheduler in each core performs the tasks in round-robin fashion, the transfer of branch metric values would be held up by the likelihood ratio conversion, thus the second window cannot be decoded in parallel. More complex scheduler may solve the problem but would again impose additional logic complexity.

4.3 Summary

The Turbo Code implementation on the RUMPS401 was discussed in great details, with focus on the decoder counterpart. It was shown during the optimization process that the basic concept of proper task distribution and simplification has significantly increased the decoding performance.

Table 4.3 summarizes the total time required by each software version to perform the Turbo decoding with single iteration on a single frame. The first software version completed the decoding in 1.88 seconds and was optimized down to 0.73 seconds decoding time on software version 1.3. Further attempt to distribute a single task to all cores in software version 1.4 resulted in a worse performance of 0.98 seconds compared to the version 1.3 due to the additional complexity of data transfer and identification. The comparison of each tasks execution time between software versions is summarized in Figure 4.19.

Table 4.3 Full-frame Decoding Time for Each Software Version

Ver	Time (s)
1.0	1.88
1.1	1.59
1.2	0.88
1.3	0.73
1.4	0.97

During the software optimization process it was observed that, reducing or simplifying complex mathematical operations, together with the use of available hardware accelerator for some operations significantly improves the execution speed. It was also observed that in the RUMPS401, splitting a bigger task into smaller parallel tasks provides better improvement than distributing a same type of task across multiple cores. Splitting the algorithm from branch metric calculation and splitting likelihood ratio conversion from its computation improves the decoding speed. In contrary, distributing the same conversion task across all cores degrades the decoding speed. This confirms that task-centric parallelism scheme suits the RUMPS401 better than the data-centric parallelism scheme. This chapter thus concludes the adoption of software version 1.3 as the decoder implementation on the RUMPS401, together with the coherent-BPSK modulation scheme, the implementation of which is detailed in the next chapter. Pseudocode of Turbo decoder software version 1.3 for can be found in Appendix A.

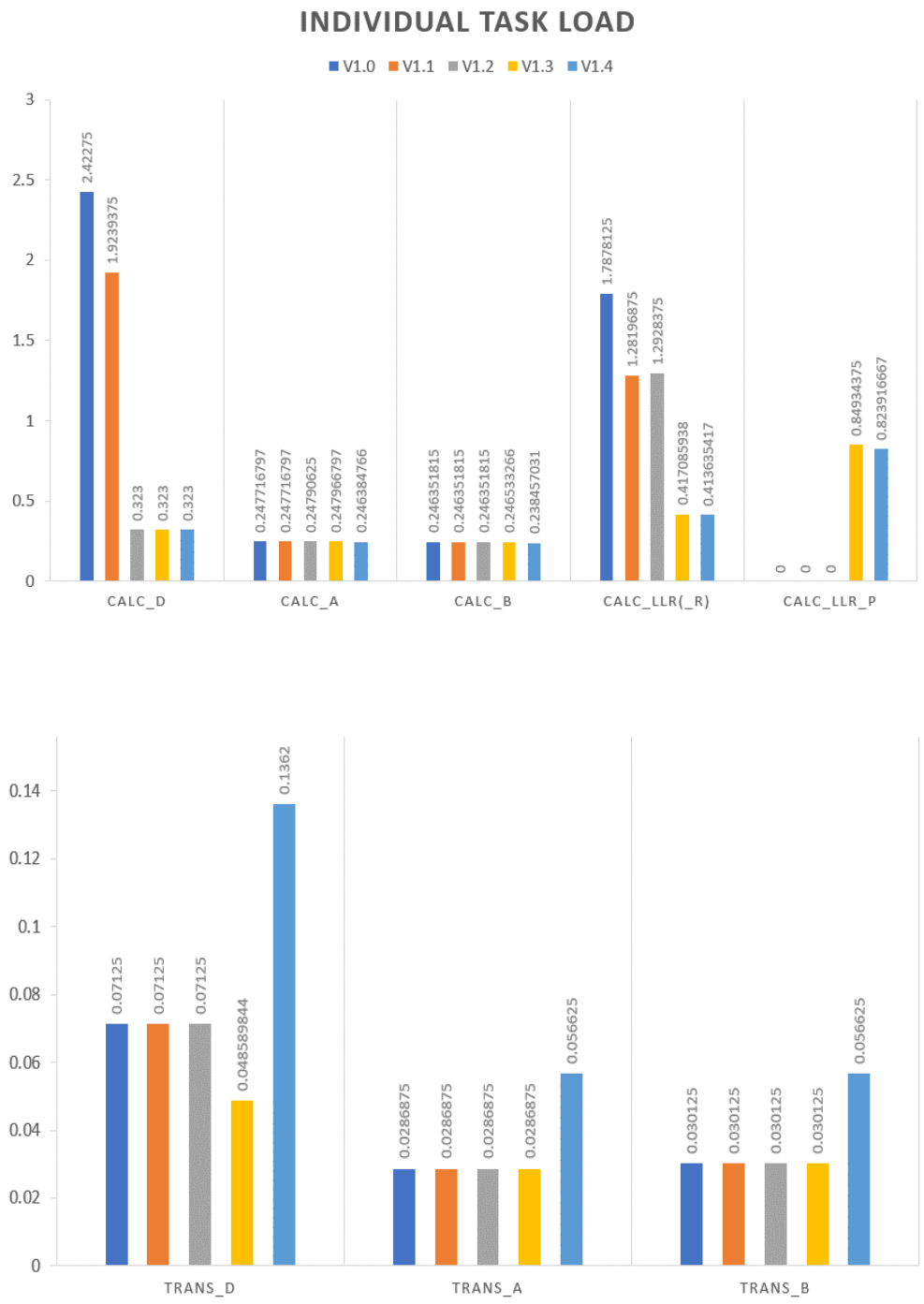


Figure 4.19 Individual Task Load Comparison Between Versions

CHAPTER 5

SOFWARE-BASED IMPLEMENTATION OF COHERENT BPSK TRANSCEIVER ON THE RUMPS401

From the study of digital modulation schemes in Chapter Three, coherent-BPSK is chosen as the digital modulation in this work, due to its simplicity and low error rate. As with the Turbo Code, both the BPSK modulator and coherent demodulator are implemented in the RUMPS401 in software. The implementation faces the same principal challenge as the Turbo Code, which is the low-power hardware architecture with no hardware support for fast and complex mathematical operations except multiplication. This chapter discusses the software implementation of the BPSK modulator and demodulator in the RUMPS401, divided into four parts.

Paired together with the RUMPS401 is a programmable radio frontend from Lime Microsystem, the Lime LMS6002D (Microsystem, 2012) that is required for the actual wireless transmission and reception. The first section describes in detail the usage of the Lime LMS6002D for this work. The second section describes the BPSK modulator implementation, while the third section details the BPSK demodulator implementation. Lastly, the fourth section summarizes this chapter.

5.1 Programmable Radio Frontend Lime LMS6002D

As described in the first chapter, the wireless system in this work consists of the one transmitter and one receiver, each composed of the RUMPS401 as the digital processor and the Lime LMS6002D as the radio front end. Both the RUMPS401 and the Lime LMS6002D are packed into their own development board, which shall be connected to form a complete wireless module. This subchapter breaks down the usage of the Lime LMS6002D in this work into three sections. Section one discusses the wireless system's structure along with the functionalities partition between the RUMPS401 and the Lime LMS6002D. Section two presents the hardware and software setup for integrating the Lime LMS6002D and the RUMPS401. The third section represents how the RUMPS401 interfaces with the Lime LMS6002D for the data transmission and reception. This includes the details of both chips development boards and their hardware connection along with the signaling protocol.

5.1.1 Complete Picture of the Wireless System

Figure 5.1 depicts the wireless system functional diagram. The RUMPS401 works fully on digital baseband domain, while the Lime LMS6002D works in the analog domain. Conversion between digital and analog domain is also performed by the Lime LMS6002D. On the transmit side, the RUMPS401 processes the digital data received from application processor and feeds the processed data to the Lime LMS6002D, which converts the signal

to analog domain then mixes it with carrier signal before wirelessly transmitting it.

The reverse happens on the receiving side where the Lime LMS6002D separates the information signal from the carrier, then feeds the received data to the RUMPS401 after converting the signal back to digital domain. The RUMPS401 processes the digital data prior to passing it to application processor. The application processor could be any digital processor that utilizes the wireless transmission service. In this work, the Central Processing Unit (CPU) of a laptop or personal computer is the application processor that communicates with the RUMPS401 via a Universal Asynchronous Receive Transmit (UART) interface.

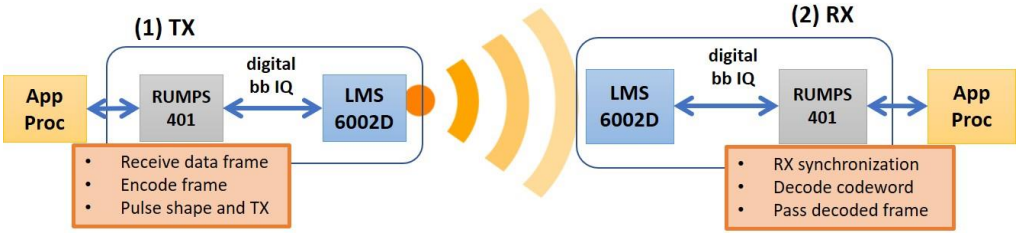


Figure 5.1 Complete Wireless System Diagram

The application processor passes original data to the RUMPS401 for wireless transmission. Before the actual transmission process, every wireless standard encapsulates the data into a fix-sized frame consisting of the data and preambles. If the original data size is larger than the frame size, it is split into several smaller packets. On the other hand, the wireless standard pads extra bits to the original data if the size is smaller than the defined frame size. Both the splitting and padding process is straightforward and shall not be further

discussed. It is assumed that the application processor on the transmitting side generates a frame of exactly 256 bits to be sent over wirelessly, which is identical to the frame size derived through the Turbo Code simulation in Chapter Three. The RUMPS401 encodes the original data frame into a codeword of 768 bits with the procedure defined in Chapter Four. The RUMPS401 constructs a transmit frame by adding preambles to the codeword, then perform BPSK modulation and pulse shaping to produce the baseband signal, which shall be detailed in the next subchapter about the BPSK modulator implementation. The RUMPS401 then drives the baseband signal into the Lime LMS6002D for the wireless transmission.

On the receiving end, the Lime LMS6002D receives the signal and passes the down-converted digital data to the RUMPS401. As in the transmit side, the received frame consists of a 768-bit codeword and preambles. The RUMPS401 then performs receiver synchronization by utilizing the preambles, to correct any timing or frequency offset experienced by the frame during the reception process. This synchronization is the most essential part of the coherent-BPSK demodulator and is detailed in the third subchapter. The Turbo decoder then takes the corrected data frame as an input, decodes and outputs a 256-bit data frame to the application processor.

5.1.2 Lime LMS6002D Implementation Setup

As discussed in the previous section, the Lime LMS6002D performs two essential operations of a wireless transceiver. The first one is the conversion of signal between the analog and digital domain. The second one is the up/down-

conversion of the analog signal along with the signal wireless transmission and reception. Figure 5.2 depicts the Lime LMS6002D block diagram. The chip has separated lanes for the transmit and receive path. Each path composed of four main components, the Analog-to-Digital Converter (ADC) / Digital-to-Analog Converter (DAC), the lowpass filter, the Phase-Locked-Loop (PLL) frequency synthesizer along with the signal mixer, and the Voltage Gain Amplifier (VGA) (Microsystem, 2012). These main components collectively provide the two wireless transceiver operations.

The 12-bit ADC/DAC performs signal conversion between the analog and digital domain, providing 4096 signal levels ranging from -2047 to +2048. Its sampling rate is based on the clock signal driven by the RUMPS401, which is detailed in the next section. The lowpass filter removes any out of band components from the analog baseband signal. The PLL frequency generator produces local carrier signal whose frequency ranges from 0.3 to 3.8GHz. On the transmitter side this generated carrier signal is mixed with the analog baseband signal for up-conversion, while on the receive side it is mixed with the received signal for down-conversion to baseband signal. Lastly, the VGA adjust the signal strength to a desired level.

Combinations of every Lime LMS6002D component's parameters define the functionality performed by the wireless transceiver. The Lime LMS6002D allows user to fully configure the parameters of these components by modifying the Lime LMS6002D internal registers values through the SPI interface, as shown on the bottom right part of the chip in Figure 5.2. These registers are volatile, thus must be reconfigured on every power cycle. The

configurable parameters include, but not limited to ADC/DAC sampling rate, lowpass filter bandwidth, carrier signal's frequency, and VGA's amplifying factor. SPI communication between the Lime LMS6002D and the RUMPS401 is performed by the IO Control Core, based on the signaling rule defined in the Lime LMS6002D datasheet (Microsystem, 2012).

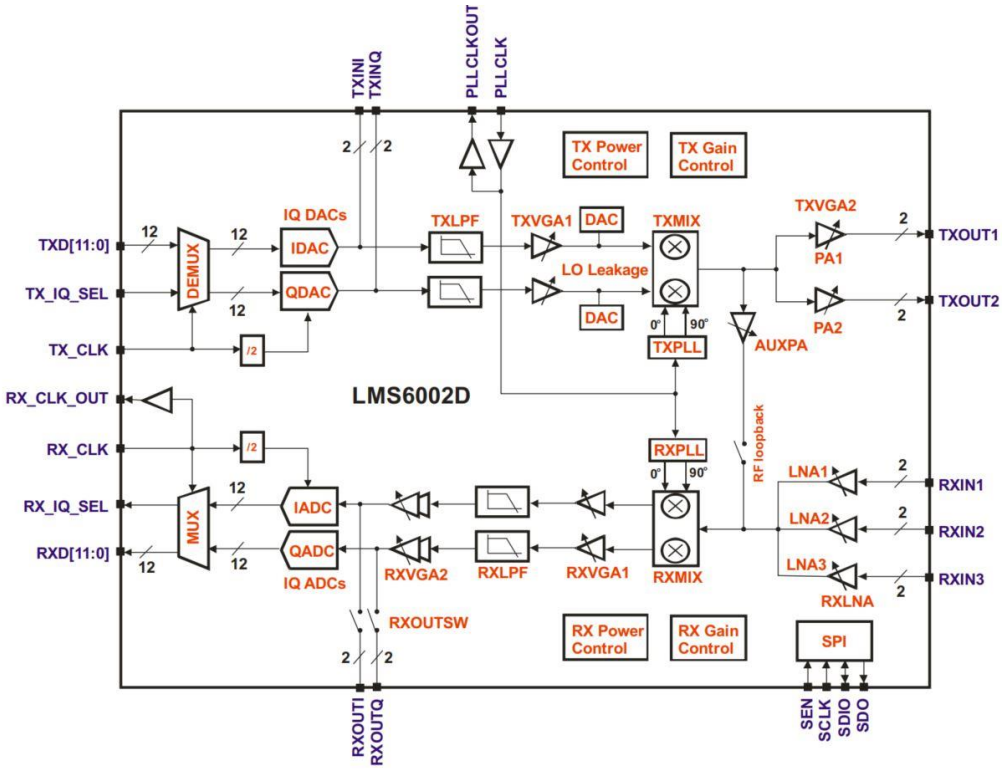


Figure 5.2 Lime LMS6002D Functional Block Diagram (Microsystem, 2012)

Digital baseband data transfer into and out of the Lime LMS6002D is provided through the digital data interface shown on the left part of the chip in Figure 5.2. As in the analog signal path, the digital data interface also consists of two separated groups for transmit and receive. Each group composed of a 12-bit data line and a control line. The data line is a 12-bit parallel bus carrying the multiplexed In-phase & Quadrature (IQ) data samples. The control line consists

of two signals, the IQ_SEL which controls the multiplexing of IQ components, and the CLK signal which dictates the ADC/DAC sampling speed. On transmit side the RUMPS401 writes into both the data line and the control line, while on receive side the RUMPS401 drives the CLK signal and reads from both the data line and the IQ_SEL signal. For both cases, the write and read operation must be done with proper signaling protocol and timing defined in the datasheet, which shall be detailed in the next section.

Referring to Figure 5.2, shown on right side of the Lime LMS6002D are the Radio Frequency (RF) signal interfaces. There are two output lines and three input lines which are prepared for multi-band operation. Since this work operates on one frequency band, only one of each input and output line is used. The Lime LMS6002D also provides access to analog baseband signals via the TXINI, TXINQ, RXINI, and RXINQ interfaces. TXINI and TXINQ are the in-phase and quadrature components of the transmit signal, respectively. These signals can either be driven by the DAC output or an external source. Similarly, RXINI and RXINQ are the in-phase and quadrature components of the receive signal, which are passed to the ADC. Like the transmit counterpart, these signals can either be driven by external source or the output of the down-conversion mixer.

In this work, both transmit and receive signals must be processed digitally by a software running in the RUMPS401. The RUMPS401 drives digital signal into the DAC, producing the TXINI and TXINQ signals. On the receiving side, the RXINI and RXINQ signals are sourced from the mixer output, which are then converted into digital signal by the ADC for further

processing by the RUMPS401. The TXINI, TXINQ, RXINI, and RXINQ signals are not meddled by any external source. Instead, they are used during the development process for monitoring the transmit and receive signal in analog baseband region.

The two PLL frequency generators allow the Lime LMS6002D to transmit and receive at different frequency. Both generators require a reference frequency as the PLLs input, supplied by external source through the PLLCLK interface. The reference frequency can be set to any value between 23MHz and 41MHz. However, 40MHz is recommended for easier calibration process (Microsystem, 2012). In this work, a programmable clock generator from the Silicon Labs Si5356-EVB clock generator is used for producing the reference frequency for the Lime LMS6002D's PLLs. The board can easily be programmed via USB by a Windows-based program supplied by Silicon Labs. It is capable of producing clock signal whose frequency ranges from 1MHz to 200MHz (Labs).

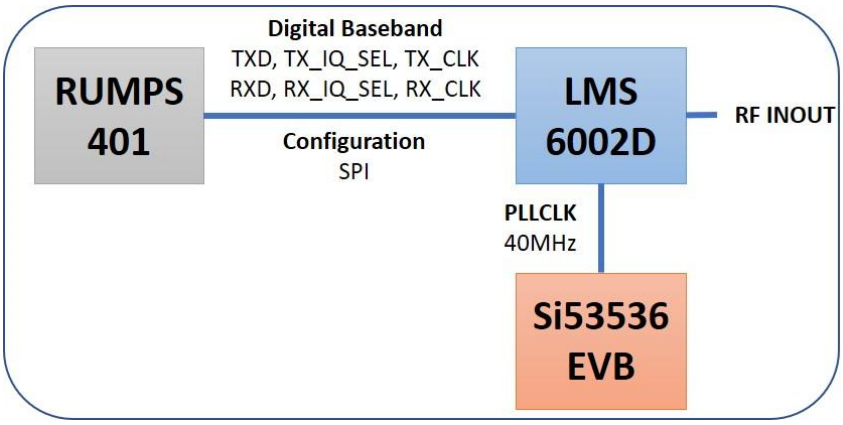


Figure 5.3 Complete Wireless Module Diagram

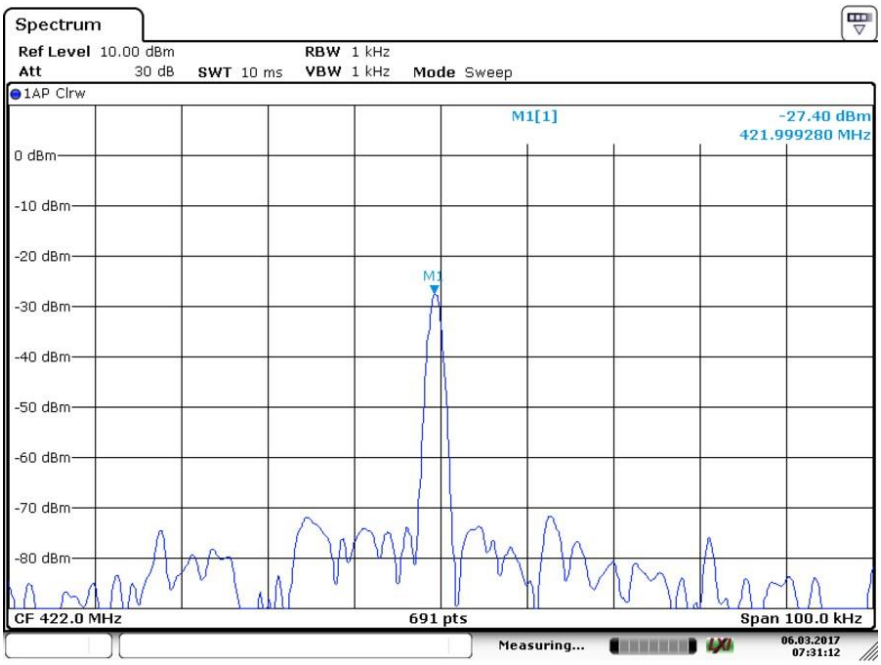
Figure 5.3 depicts the block diagram of a complete wireless module which consists of the RUMPS401, the Lime LMS6002D, and the Si5356-EVB. Connections between the RUMPS401 and the Lime LMS6002D are classified into two groups, the digital baseband connection consisting of signal values and ADC/DAC control signals, and the SPI connection for the Lime LMS6002D parameters configuration. The Si5356-EVB supplies a 40MHz clock signal as the Lime LMS6002D's PLL reference clock.

As mentioned earlier, the Lime LMS6002D configurations are stored in a set of registers that must be set on every power cycle by the RUMPS401. After the registers configuration, the Lime LMS6002D will operate in the desired setting, e.g. 2.4GHz center frequency and filter bandwidth of 1.5MHz. Following the register configurations, the Lime LMS6002D must be calibrated to produce optimum settings. Like the operation parameters configuration, the Lime LMS6002D calibrations are also performed by the RUMPS401 via the SPI interface by accessing certain registers dedicated for calibration purpose. The calibration sequence is split into three steps, the PLL frequency generator calibration, the Direct-Current (DC) offset cancellation of various processing blocks, and the transmit path Local Oscillator (LO) leakage calibration.

The calibration on the PLL frequency generator is required to improve the accuracy of the carrier's signal frequency. Prior to the calibration, the produced carrier signal normally drifts from the desired frequency. The Lime LMS6002D allows fine tuning of the resulting frequency by choosing the PLL's internal capacitor's value which ranges over preset numbers. This value can be changed by simply modifying a register value. The correct capacitor value

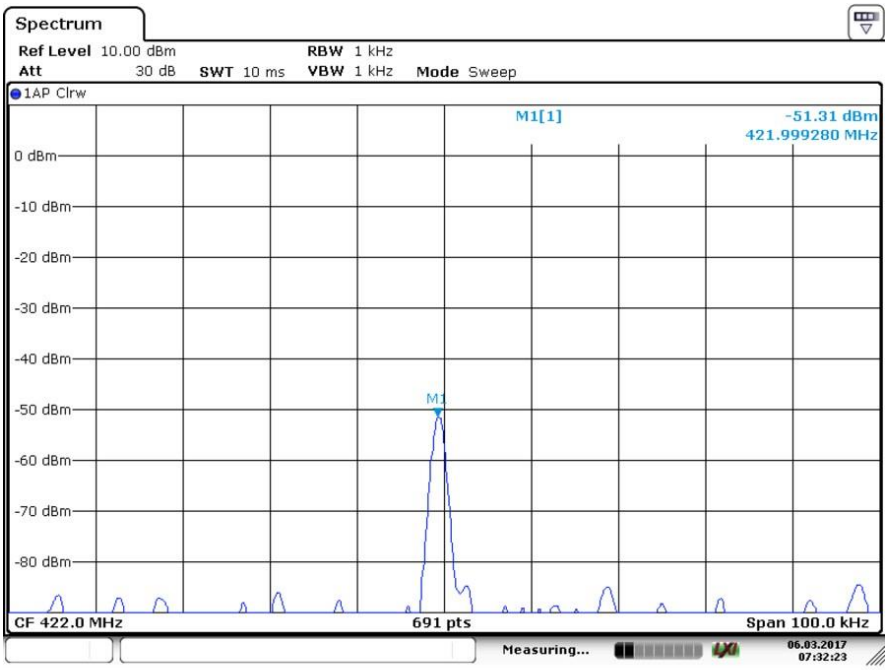
results in a carrier signal with least frequency drift. The calibration objective thus is to find the suitable value by cycling over those preset values and checking the outcome via another register, which tells if the capacitor value is too high, too low, or just fine. The RUMPS401 thus needs to find two capacitor values which are started to be considered too high, or too low. The suitable value is the middle of those two values. The DC offset calibration is performed on various processing blocks of the Lime LMS6002D chip shown in Figure 5.2 to cancel the DC effect on the analog components. This calibration is performed by accessing certain registers to start the autonomous calibration process and waiting for a certain time before the calibration finishes.

Prior to the calibration, the transmit path exhibits a power leakage sourcing from the local oscillator, or the PLL frequency generator itself. The Lime LMS6002D allows the cancellation of this power leakage by applying DC offset on the DAC output, whose offset amount can be chosen from a set of predefined values. There are two separate offset values for the I and Q signal components. As in the configurations and other calibrations, these values are set by modifying certain registers through the SPI interface. The calibration process thus is performed by cycling over the predefined DC offset values and checking the RF output signal in the frequency domain. For this purpose, the RF transmit output must be connected to a spectrum analyzer.



Date: 6.MAR.2017 07:31:12

(a) Pre-calibration RF output signal spectrum



Date: 6.MAR.2017 07:32:23

(b) Post-calibration RF output signal spectrum

Figure 5.4 Transmit Path LO Leakage Calibration

Shown in Figure 5.4a is the power spectrum of the signal coming out from the RF transmit output before the LO leakage calibration, with the DAC

turned off and no analog signal going into the TXINI and TXINQ. The DAC is turned off to avoid any arbitrary data signal affecting the output signal spectrum. The sanitization of TXINI and TXINQ from any input serves the same purpose, which is to analyze only the carrier's signal spectrum. The calibration process looks for DC offset values that result in the signal spectrum with lowest peak power. Figure 5.4b shows the post-calibration signal spectrum, where the peak power is noticeably lower compared to the pre-calibration measurement.

After the configuration and calibration processes, the two devices can function as transmitter and receiver as described in an earlier section. If the same configuration is used on every power cycle, the calibration process can be speeded up by logging down the exhaustively searched calibration values such as the PLL's capacitor and DC offset values. These values can be set directly on every power cycle instead of performing the calibration again.

5.1.3 The RUMPS401 and Lime LMS6002D Interfacing

As described in earlier section, connections between the RUMPS401 and the Lime LMS6002D are split into two parts, the digital baseband and the SPI connection. Figure 5.5 shows the exact pin-to-pin connection between the RUMPS401 and the Lime LMS6002D, where most connections on the RUMPS401 side are handled by the IO Control Core. As described in Chapter Two, the IO Control Core is equipped with thirty-two input-output (IO) pins, while the other cores each is equipped with only eight.

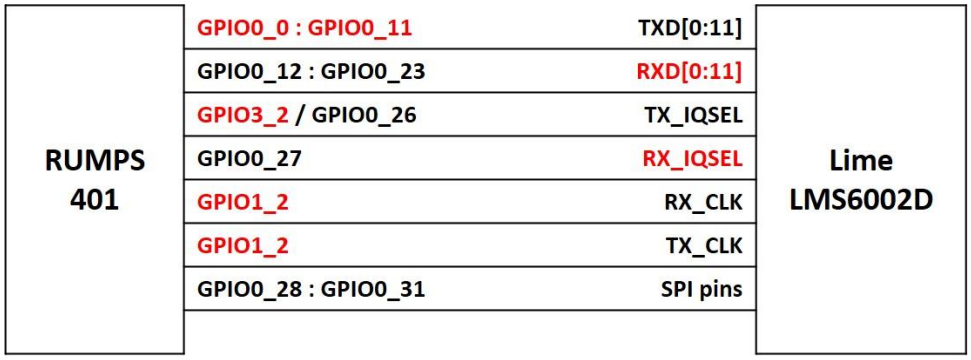


Figure 5.5 RUMPS401 and Lime LMS6002D Pins Diagram

Thirty of the IO Control Core pins are used for the connection as shown in Figure 5.5, denoted by GPIO0_x. It indicates the x^{th} IO pin of the IO Control Core, the first core of the RUMPS401. IO pins of Normal Core 0, Normal Core 1, and DSP Core are indexed as GPIO1_x, GPIO2_x, and GPIO3_x, respectively. This notation format will be carried for referencing the RUMPS401's IO pins throughout this writing. The first twenty-four pins of IO Control Core denoted by GPIO0_0 to GPIO0_23 are used to connect with the Lime LMS6002D's 12-bit transmit and receive data buses, denoted by TXD and RXD, respectively. As described before, the RUMPS401 drives the digital data to be transmitted through the transmit data bus, which will be converted by the Lime LMS6002D internal DAC for further processing in the transmit chain. Conversely, the Lime LMS6002D receives signal from the air, down-converts and passes to the internal ADC which then drives the converted digital signal to the RUMPS401 via the receive data bus.

While the transmit and receive data buses are completely separated and works individually, each bus does not have distinguished line between I and Q signal components. Instead, the I and Q components are multiplexed into the

data bus that can only carry one component at a time. TX_IQSEL and RX_IQSEL are the signals controlling the multiplexer for the transmit and receive path, indicating whether the bus is carrying I or Q component at a certain time.

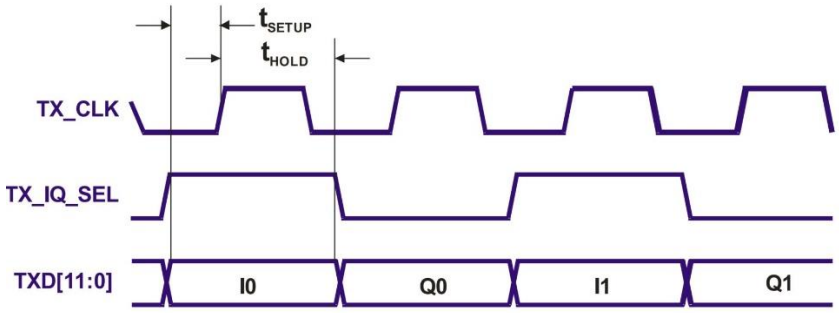


Figure 5.6 Lime LMS6002D TX Timing Diagram

Figure 5.6 shows the signals required by the Lime LMS6002D for transmission purpose, along with their timing. All three signals are driven by the RUMPS401 via its IO pins. TX_CLK is a clock signal supplied into the Lime LMS6002D internal DAC, dictating the DAC sampling rate which is half of the clock frequency. The halving of TX_CLK into the DAC sampling clock is performed internally by the Lime LMS6002D. TX_IQSEL takes the form of a clock signal whose frequency is half of the TX_CLK. As depicted by the timing diagram, data must already be written to the TXD bus along with the TX_IQSEL signal at least 1ns before TX_CLK clock edge arrives, which then latches the data into the DAC. In this timing diagram, TX_IQSEL flags I component with high signal level and Q component with low signal level. It should be noted that the flagging is interchangeable, i.e. I marked with low signal level and Q with high, as well as the choice of latching the data on positive or negative clock edge.

Signals timing diagram for the receive line is depicted in Figure 5.7. As in the transmit line, RX_CLK is a clock signal driven by the RUMPS401 into the Lime LMS6002D internal ADC to control its sampling frequency. The Lime LMS6002D outputs the received data into the RXD bus, along with the RX_IQSEL signal to indicate whether the data on RXD bus is the I or the Q component. In this diagram, the data is read on every RX_IQSEL clock edge while the RX_CLK level is held high. Positive clock edge indicates I component on the RXD bus, while negative clock edge indicates Q component. Identical to the transmit line, the trigger level and clock edge are fully configurable.

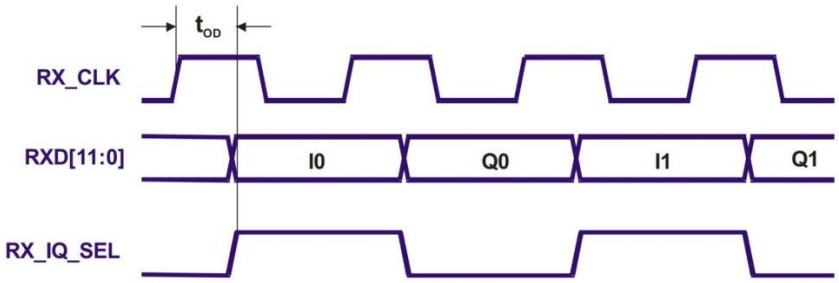


Figure 5.7 Lime LMS6002D RX Timing Diagram

Adhering to the signaling diagrams above, to properly transmit and receive with the Lime LMS6002D, the RUMPS401 is required to produces two clock signals, the T/RX_CLK signal and the TX_IQSEL signal. These clock signals are provided by the RUMPS401 via its IO pins, and produced by utilizing the RUMPS401's internal timer to periodically toggle the pins value. The timer supports hardware toggling that automatically toggles the pin value each time the timer reaches desired ticks. providing a more stable clock compared to software toggling. The timer provides two hardware-toggled pins with separated timer comparator hence the two pins can be toggled at different

time. However, those comparators share the same counter which can only be reset or stopped upon reaching a certain value. Hence, despite the two hardware-toggled pins, a single timer would not be able to produce two different clock signals.

Since each of the RUMPS401 cores is equipped with individual timer, two cores can be used instead. The IO Control Core does not participate in producing these clock signals because its internal timer is responsible for timing the write and read into and from the data buses. Other cores can be used without any concern, and the choice of which cores generating the clock signals poses no issue. In this work, the Normal Core 0 produces T/RX_CLK via its GPIO1_2 pin, while TX_IQSEL is produced by the DSP Core via its GPIO3_2 pin. As described earlier, during the read operation the RUMPS401 requires RX_IQSEL signal to distinguish between I and Q data as well as timing the operation properly, which is received by the IO Control Core via GPIO0_27 pin. Likewise, the RUMPS401 must align the data write operation properly with the TX_IQSEL, hence the IO Control Core also receives the TX_IQSEL signal via the GPIO0_26 pin. There is no need for the IO Control Core to align with the T/RX_CLK signals directly since the T/RX_IQSEL signals is already aligned to those.

For registers access purpose, the RUMPS401 interfaces with the Lime LMS6002D via the IO Control Core's SPI hardware, whose pins are provided through GPIO0_28 to GPIO0_31. The SPI pins connection is straightforward with the RUMPS401 as master, and the Lime LMS6002D as slave. The use of timer and SPI hardware greatly relieves processing burden from the RUMPS401

software. Only a tiny amount of the software time is spent for initiating the timer or SPI transfer process, and for checking the SPI transfer result.

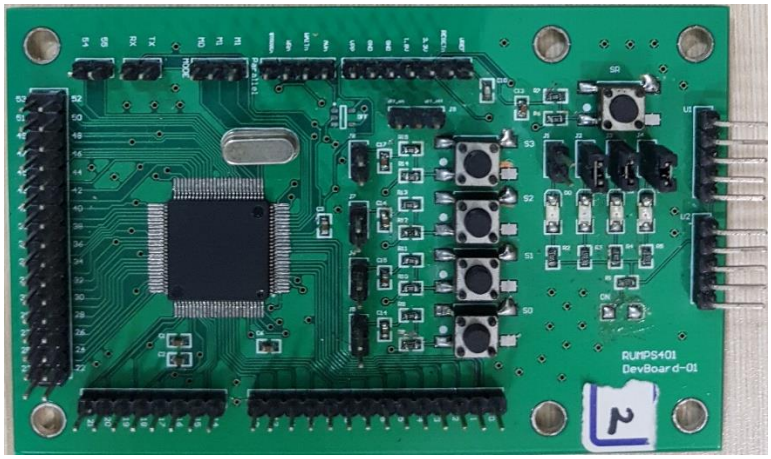


Figure 5.8 RUMPS401 Development Board

The physical connection between the two chips is straightforward. Both the RUMPS401 and the Lime LMS6002D are packed in their own development board which gives access to its IO and allows them to be used out of the box. Design of the RUMPS401 development board which is also a part of this work is shown in Figure 5.8. This board basically provides the RUMPS401 connections to an on-board external crystal oscillator, several push-buttons and LEDs, and routes the RUMPS401 IO pins out to group of male pin headers to allow convenient access to other external devices.

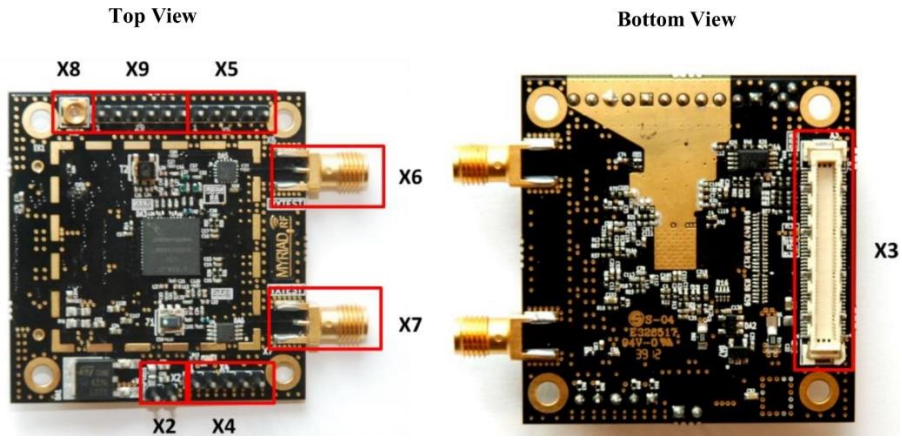


Figure 5.9 Lime LMS6002D Reference Development Kit

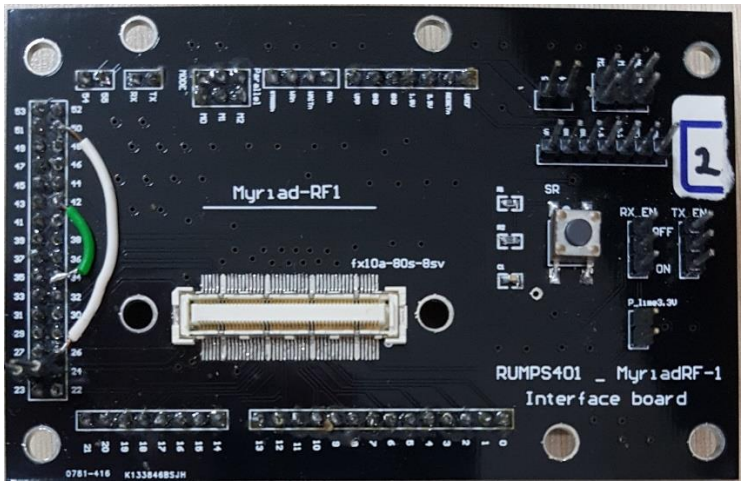


Figure 5.10 RUMPS401-Lime LMS6002D Interface Board

The Lime LMS6002D is packed in the Myriad RF Reference Development Kit, shown in Figure 5.9. It provides access to the RF signal interfaces via SubMiniature version A (SMA) coaxial connectors and access to PLL reference clock via Micro Miniature Coaxial (MMCX) connector, while access to the digital baseband signals is provided through a compact and high-speed Hirose FX10A-80P0 connector (MyriadRF, 2013). An antenna is connected to each of the transmit and receive RF signal connector, while input to the PLL reference clock is directly connected to the Si5356-EVB clock

generator. Figure 5.10 shows a simple interface board designed to connect the RUMPS401 IO pins to the Lime LMS6002D digital baseband interface provided via the Hirose FX10A-80P0 connector.

5.2 Software-based BPSK Transmitter on the RUMPS401

As described in the previous subchapter, the Lime LMS6002D is a programmable RF transceiver which only works on raw baseband signal. It does not provide any means of modulation or demodulation, hence the baseband processor must perform the signal processing itself on both transmit and receive ends. On the transmit side, the baseband signal supplied by the RUMPS401 into the TXD bus is directly converted to analog signal and mixed with the carrier frequency for transmission. Thus, for a proper transmission the RUMPS401 needs to transform the data frame which is in binary bits into a baseband signal, which is then passed to the Lime LMS6002D.

The transmission involves three steps, preambles augmentation, IQ modulation, and pulse shaping which are all performed by the IO Control Core. Since this work implements coherent-BPSK, the receiver is required to perform synchronization on the received data before further processing. The synchronization utilizes sequence of bits whose values are known by the receiver. These bits are augmented to the beginning of the data frame and sent along with the data. For convenience, this augmented frame will be referred as the transmit frame throughout the chapter. BPSK modulation is then carried on the transmit frame by simply deriving I and Q value for every bit in the frame based on the defined IQ modulation scheme. This modulation process yields

two baseband signals, the I and the Q signals which collectively represents the transmit data frame. Lastly, pulse shaping filter is applied to both the I and Q signals to limit the signal bandwidth and to reduce inter symbol interference (ISI) on the receiving end.

This subchapter is arranged into three sections as follow. The first section presents the IQ modulation scheme, while section two describes the pulse shaping function. Lastly, third section discusses the transmit frame format.

5.2.1 BPSK IQ Modulation Scheme

To perform BPSK IQ modulation, the RUMPS401 is required to feed the I and Q values corresponding to each bit in the transmit frame in a defined rate. This means that RUMPS401 must be able to provide the necessary signals on TX_CLK, TX_IQSEL, and TXD according to the timing diagrams in Figure 5.6 and Figure 5.7 for every bit in the transmit frame and every transmission. Generating stable clock signals for TX_CLK and TX_IQSEL is completely feasible for the RUMPS401 due to the timer-based hardware toggling described in the previous subchapter.

The RUMPS401 runs on 16MHz oscillator clock, thus the internal timer ticks on every clock cycle which is 62.5ns. This limits the maximum frequency of clock signal the RUMPS401 can generate at 8MHz. While it is possible to generate 8MHz clock signal for TX_CLK and 4MHz clock signal for TX_IQSEL, the RUMPS401 could not drive the transmit data in such a short

time. Due to the use of pulse shaping, for each bit in the transmit frame the RUMPS401 needs to choose proper IQ value, calculate the pulse response for that value then drives it to the TXD bus in time. Likewise, the data rate is also limited on the receive side as the RUMPS401 must read the value in RXD bus synchronization in time then performs the timing for each arriving bit. Coupled with the fact that these are performed via software, it is reasonable to lower the data rate to match the RUMPS401 processing capability.

To decide on a suitable data rate, adjustments are carried with the RUMPS401 performing the calculation and data bus access required for the transmit and receive as mentioned above at various rate. The T/RX_CLK, T/RX_IQSEL, and T/RXD outputs from the RUMPS401 are monitored with a logic analyzer. During the adjustments, instead of doing the real read or write operation, the data bus value is toggled. This toggle represents at which time the data bus access operation occurs and compared to the write or read with real data, the toggling edges are easier to track with the logic analyzer. Adjustments for the transmit and receive processes are performed separately, each to find the fastest data rate the RUMPS401 operate at while keeping the signals timing correct. The suitable data rate is the lowest data rate between those two.

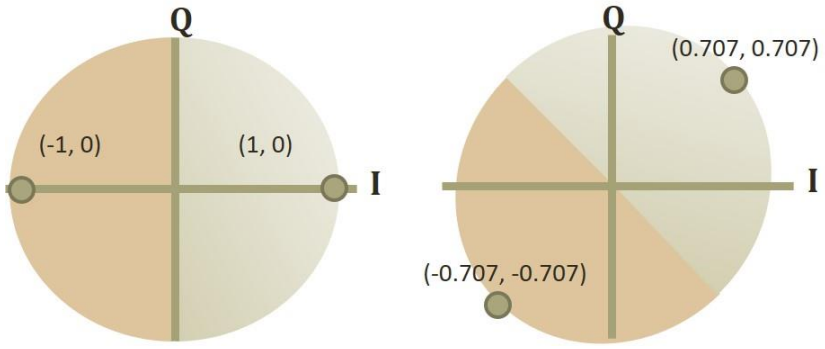
The simple adjustment process derived that the RUMPS401 can stably toggles the data bus, producing up to 8 KHz clock signal while performing the necessary calculation for either transmit or receive. Any attempt on setting the RUMPS401 to toggle the data bus at frequency higher than 8 kHz resulted in incorrect signals timing, i.e. the RUMPS401 could not access the data bus in time as required by the timing diagrams of Figure 5.6 and Figure 5.7. Since 8

kHz is the frequency of clock signal, this translates to 16kbps of data rate. However, this 16kbps data rate is a raw data rate where pulse shaping and receiver's Nyquist sampling rate have not been taken into considerations.

Table 5.1 Digital Baseband Rate Summary

Baseband data rate	2 kbps
TX pulse shaping rate	8 kbps
RX Nyquist sampling rate	16 kbps
ADC/DAC sampling rate	32 kHz
TX_IQSEL	32 kHz
T/RX_CLK	64 kHz

Table 5.1 presents the fixed data rate and signaling frequency for the digital baseband interface between the RUMPS401 and the Lime LMS6002D. According to the Nyquist theorem (Jr, Sethares, & Klein, 2011) the received data must be sampled at least at twice the transmit data rate. Since the RUMPS401 can only read or write at fastest rate of 16kbps, the receiver is set to read from data bus at that rate. The Nyquist theorem thus limits the transmit rate to be at most half of the receiver sampling rate, which is 8kbps. Additionally, the use of pulse shaping also limits the actual data rate since each bit must be oversampled to produce smoother pulse response. The pulse shaping filter used in this work oversamples the original data signal by four times, hence lowering the actual data rate further to 2kbps. As described above, the pulse shaping function and the receiver is detailed in the next section and subchapter, respectively. Lastly, the Lime LMS6002DADC/DAC is set to sample the data at higher rate of 32 kHz by generating 64 kHz and 32 kHz clock signals for the T/RX_CLK and TX_IQSEL, respectively.



(a) BPSK - 0˚ offset **(b) BPSK - 45˚ offset**

Figure 5.11 BPSK Constellation Diagrams

Figure 5.11 depicts constellation diagrams of two BPSK modulation schemes, which differ in the symbol phase offset. In the first scheme, the BPSK symbols only occupies I channel which is shown by the bit 1 and bit 0 represented as (1,0) and (-1,0) respectively. In the second scheme, 45˚ phase offset is applied to the BPSK symbols, thus occupying both I and Q component, shown by the bit 1 and bit 0 represented as (0.707, 0.707) and (-0.707, -0.707) respectively. Both schemes provide equal error rate, with the first 0˚ offset BPSK scheme being more common since the BPSK basically only need one channel to represent the two distinct symbols. In terms of demodulation, the first scheme provides a slight advantage since the decision device is only required to demodulate based on the I component, while the second scheme requires checking on both I and Q component.

Additionally, depending on the wireless system stack, there may be other processing elements before the demodulator which is designed to work only on the I component, assuming the more common form of BPSK with 0˚ phase offset. This is the case with the Turbo decoder in this work which only

accepts the I component as inputs. From this aspect alone, the 0˚ phase offset scheme is more suitable for pairing with the Turbo decoder. However, taking the digital line timing diagram described above into consideration, the second scheme with 45˚ phase offset is more feasible to implement in the RUMPS401. Recall that based on Figure 5.6, the TX_IQSEL's frequency must be half of the TX_CLK's, which is also reflected in the signaling rate defined for this work in Table 5.1.

TX_IQSEL is set at 32 kHz, and it was discussed that the RUMPS401 software could not access the data bus at that rate. If the 0˚ phase offset BPSK is implemented, the RUMPS401 must be able to provide different value for the I and Q component by accessing the data bus at the same rate as the TX_IQSEL. This is not affordable with signaling rate defined in Table 5.1. Implementing the 0˚ phase offset BPSK is possible by lowering the signaling rate, which in turn increases the radio air time. On the opposite, the implementation of 45˚ phase offset BPSK conserves the signaling rate since the I and Q components possess the same value, allowing the RUMPS401 to perform data bus access at any rate lower than the TX_IQSEL. The only drawback of implementing the 45˚ offset BPSK in this work is the Turbo decoder which assumes input of BPSK symbols with 0˚ phase offset. This problem is solved at the receiver by re-rotating the received symbols by 45˚. Although this solution imposes additional complex number calculation for the symbol rotation, it is more suitable since the higher data rate allows lower radio air time.

The RUMPS401 is a naturally low-power Multi-Processor System-on-Chip (MPSoC) which only draws 32mA current, while the Lime LMS6002D

consumes around 260mA when activated for either transmission or reception. Rather than having the Lime LMS6002D to transmit over longer period, it is more reasonable to receive and buffer the data, turn the Lime LMS6002D off and let the RUMPS401 consumes the extra time for the symbol re-rotation.

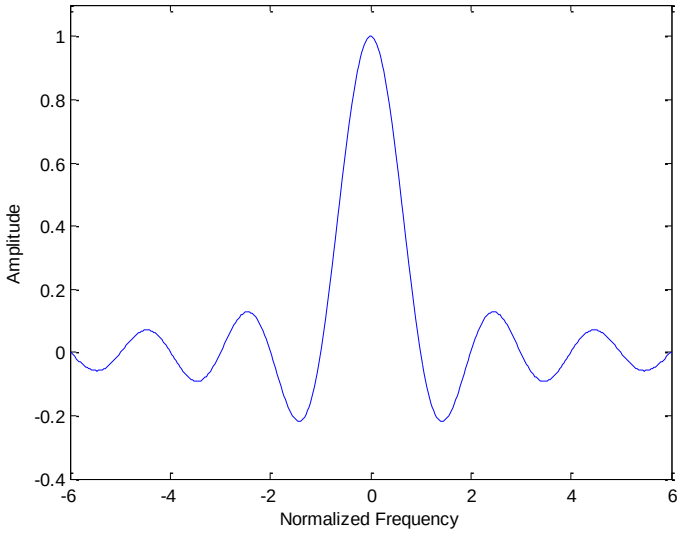
The IQ modulator is implemented in the IO Control Core since its functionality is to simply choose the I and Q component value based on the data bit. As described earlier in this chapter, the Lime LMS6002D internal ADC/DAC uses 12-bit two's complement number format, covering integer value of -2047 to +2048. In this work, the value of +1 and -1 in the IQ plane is represented by +1024 and -1024. The reason for choosing these middle values instead of the maximum values of +2048 and -2047 is related to the pulse shaping filter response which is detailed in the following section.

5.2.2 Raised-Cosine Pulse Shaping Filter Implementation

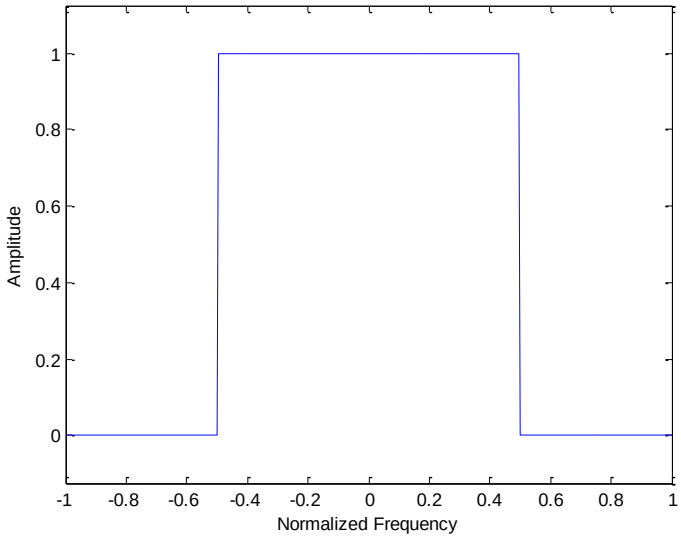
In practical wireless system, pulse shaping filter is commonly employed on the transmit side in pair with the IQ modulator. In case of plain digital IQ modulation, the I and Q baseband signals are square waves with sharp edge transitions, resulting in frequency response of a sinc function with high sidebands power (Instruments, 2014). Such waveform is not suitable for practical use due to two reasons. First, frequency spectrum is a scarce resource and it is expected that any transmission would only occupy the allocated channel.

A transmission with frequency response mentioned above will have its sidebands interfere with other adjacent channels. Second, signal sent by the transmitter may arrive at the receiver in several time-delayed versions due to multipath interference. This cause the delayed symbol to drift from its sampling time, and instead overlaps with the adjacent symbol. Since square wave amplitude is constant for the whole symbol or bit duration, the overlapping part contains the maximum power of the delayed symbol, causing interference during sampling of the next symbol, a phenomenon known as ISI (Instruments, 2014).

The two problems above can be mitigated by a pulse shaping filter. The filter basically shapes the impulse response of each symbol which was in form of square pulse into sine pulse. Sine pulse is chosen because it holds properties which are the opposite of the square pulse's (Proakis & Salehi, 2001). Frequency response of a square function is a sine function, and so does the reverse. As shown in Figure 5.12, contrary to the sine function that consumes large bandwidth due to the sidebands, square function consumes limited amount of bandwidth. Moreover, in time domain the symbol value is contained only at the peak of the sine pulse at the sampling instance. Hence, assuming that a symbol delay does not exceed the sampling instance, ISI can be reduced since the overlapping part does not contain the maximum power of the pulse. Figure 5.13 depicts the ISI for square and sine pulses. It is evident that interference produced by the delayed symbol is less pronounced on sine pulse since the overlapping part does not contain the peak power of the pulse (Jr, Sethares, & Klein, 2011).

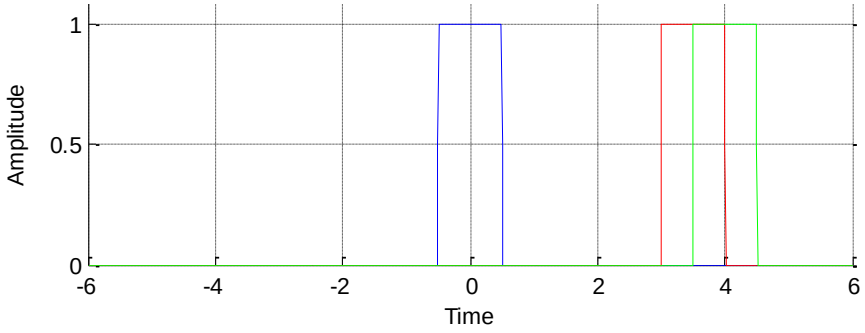


(a) Sinc Function

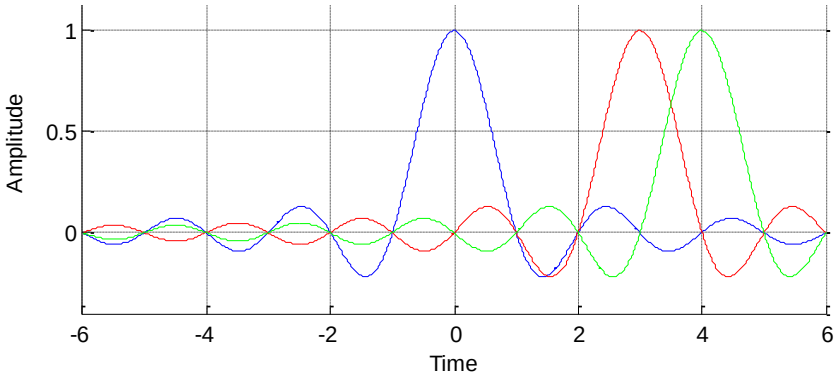


(b) Square Function

Figure 5.12 Sinc and Square Pulse



(a) ISI for Square Pulse

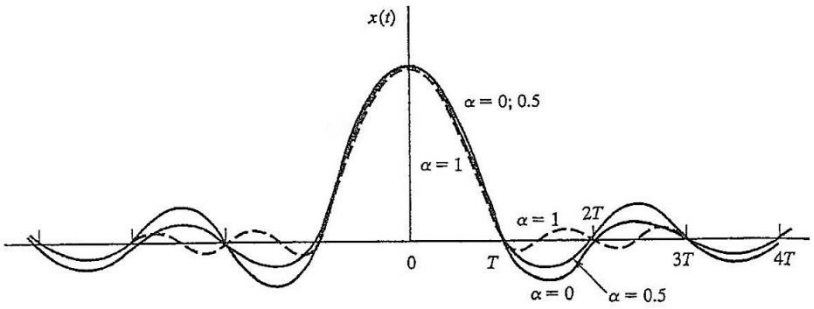


(b) ISI for Sine Pulse

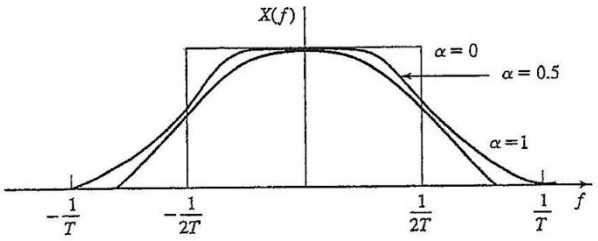
Figure 5.13 ISI for Various Pulse Shapes

A filter that has been widely used to produce such impulse response is raised cosine filter (Proakis & Salehi, 2001). In this work, a raised cosine pulse shaping filter is implemented with a simple lookup table and addition operations. The filter has a roll-off factor of 0.5 and oversamples the original IQ square pulses by four times. The pulse transmit rate hence is capped to 8kbps to satisfy the minimum Nyquist sampling rate as described earlier, yielding the original data rate of 2kbps. The roll-off factor is any decimal value between 0 and 1, which defines two things. In frequency domain, it defines how much excess bandwidth the pulse frequency response occupies compared to the ideal squared-shape frequency response. In time domain, it determines how quick the

pulse sidebands energy goes to zero (Proakis & Salehi, 2001). As shown in Figure 5.14, the closer the roll-off factor to 1, the more excess bandwidth occupied, and the quicker the sidebands energy goes to zero. The oversampling factor here provides common tradeoff where higher oversampling gives smoother signal shape in expense of more computing and memory resources.



(a) Raised Cosine Impulse

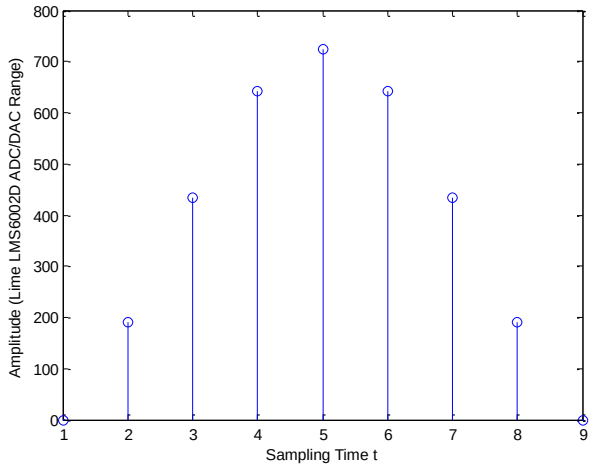


(b) Raised Cosine Frequency

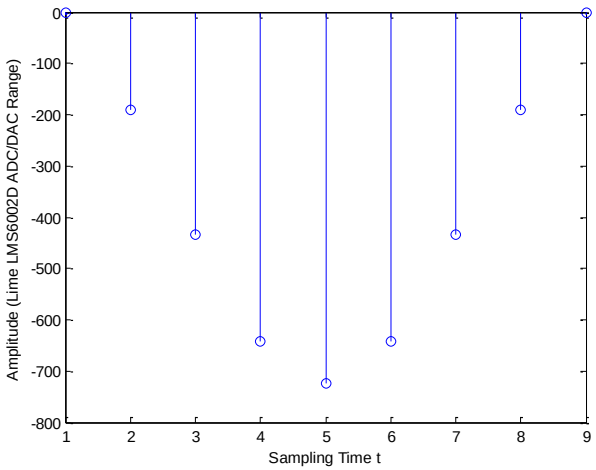
Figure 5.14 Raised Cosine Filter Response (Proakis & Salehi, 2001)

As described in previous section regarding the 45˚ offset BPSK modulation and the implementation in the RUMPS401, it can be concluded that the modulator is only required to output IQ value of either +0.707 or -0.707 to represent bit 1 or 0. It was also defined that the scaling ratio between IQ value and ADC/DAC 12-bit integer is 1:1024, i.e. IQ value of +1 and -1 are represented as +1024 and -1024, respectively. Hence, the two necessary IQ values are driven by the RUMPS401 into the Lime LMS6002D as +723 and -

723, representing bit 1 and bit 0. The pulse shaping filter is implemented by pre-calculating the filter response for those two values and storing them in a lookup table (LUT).



(a) FIR for bit 1 (value +723)



(b) FIR for bit 0 (value -723)

Figure 5.15 Finite Impulse Response for LUT

Figure 5.15 depicts the finite impulse responses of the raised cosine filter for the two values. The impulse responses were acquired by feeding the two values into Matlab's built-in `rcosine` function which was configured accordingly, i.e. roll-off factor of 0.5 and oversampling by factor of four. Output

of the `rcosine` function is then trimmed, resulting in the two impulse responses over finite points, each containing nine points of the main lobe. These impulse response values are stored in a simple LUT which is used during the transmission for calculating values to be driven to the data bus. It is evident that the points are spaced rather far away since the filter only oversamples by factor of four. As mentioned earlier, it is possible to have smoother finite impulse responses by increasing the oversampling factor in expense of much lower actual data rate. Additionally, more sampling points imposes the need for larger LUT size on the RUMPS401 limited memory, an issue which has been detailed in Chapter Four.

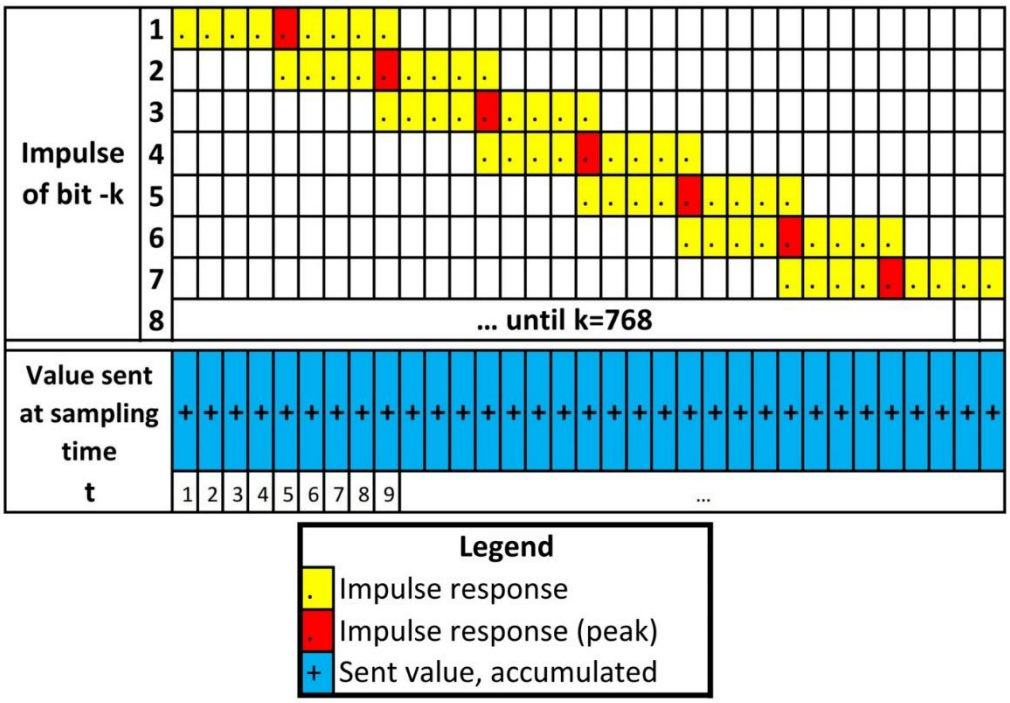


Figure 5.16 RUMPS401 Pulse Shaping Implementation

During transmission, the RUMPS401 is required to perform calculation based on the filter response LUT and the data bits to produce sequence of sampling points that represents the baseband signal. The RUMPS401 performs

this pulse shaping by calculating the value just in time before that certain sampling point must be transmitted. Figure 5.16 illustrates the pulse shaping calculation process. Each row of yellow and red blocks represents the FIR for each data bit, while each column represents the sampling time on which the RUMPS401 must write the signal value to the ADC/DAC.

As shown in the figure, adjacent FIR peak points represented by the red blocks are spaced by four sampling points since the filter uses oversampling factor of four. The value written by the RUMPS401 on each sampling time is the sum of FIR value that belongs to the same column. For example, at sampling time $t = 1$ the sent value contains only the first point of the first data bit's impulse response. At time $t = 9$, the sent value is the summation of FIR points from the first, second, and third data bit. This calculation is carried in the same fashion until the last FIR point of the last data bit is sent, i.e. the ninth FIR point corresponding to the 768^{th} data bit.

Output of the RUMPS401 pulse shaping function is shown in Figure 5.17. In this example, bits sequence of 1110 is used as the input. As shown in the figure, there are output values whose amplitude reach over the individual FIR peak value of +723 and -723, which is expected since the output is the sum of individual FIR points. This happens when there are consecutive bits with same value, i.e. consecutive 1s. Referring to Figure 5.16, due to the limited number of FIR points coupled with the oversampling factor, at one sampling time there are at most only three FIR points, whose sum does not exceed +1024 or -1024. From the transmitter point of view, these amplitude values pose no

problem since they fall within the range of values supported by the Lime LMS6002D internal ADC/DAC.

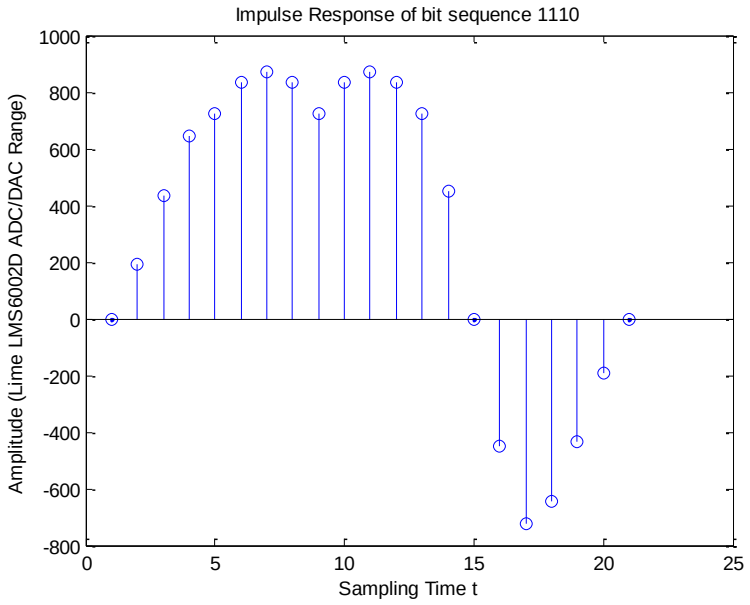


Figure 5.17 Example of the RUMPS401 Pulse Shaping Output

Assuming the ADC/DAC only ranges between +1024 and -1024, there is still no problem for the transmitter side since the values are still in the ADC/DAC range hence the pulse shape can still be conserved. However, this may pose problem for the receive side since the received signal amplitude changes during the wireless transmission. If the received signal is amplified, the pulse values may exceed the ADC/DAC range, yielding in signal whose peak is flattened. This is undesirable since pulse shaping filter is used so that the output signal only contains the actual IQ value at the peak which is the bit sampling instance. To avoid this the scaling between IQ values and the internal ADC/DAC values is set at 1:1024 instead of 1:2048.

5.2.3 Transmission Frame Format

As mentioned earlier, the transmit frame consists of the 768-bits Turbo encoded codeword and a few additional bits known as preambles, which aids the receiver in the synchronization process. Figure 5.18 describes the transmit frame format, which consists of the detection and timing preamble, the frequency preamble, and the codeword. Both frequency and timing preamble are fixed and known to the transmitter and receiver. The timing preamble is used by the receiver to correct the timing of signal sampling, as well as allowing the receiver to detect the start of the frame. The frequency preamble is used to correct the frequency offset experienced by the received signal due to the frequency difference between the transmitter and receiver local oscillator.

16 bits	2 bits	N bits	...	2 bits	N bits
Detection & timing preamble	Frequency preamble	Encoded data i=1, ... n	...	Frequency preamble	Encoded data i=n-1, ... 768
1,0,1,0,….,1,0	1,1	data	…	1,1	data

Figure 5.18 Transmit Frame Format

The detection and timing preamble is a sequence of 16 bits alternating between 1 and 0, while the frequency preamble is a pair of 1s. Reasoning for the preamble format is related to the receiver synchronization process which is detailed in the next subchapter. As shown by the frame format, the timing and detection preamble is only augmented to the beginning of the frame while the frequency preamble is inserted between the codeword. The insertion frequency is flexible depending on the trade-off between correction accuracy and computational load, which shall be detailed in the next subchapter as well.

5.3 Software-based BPSK Receiver on the RUMPS401

Like the transmit side, to retrieve the digital baseband data from the Lime LMS6002D, the RUMPS401 is required to access the digital interface according to the timing diagram in Figure 5.7. The RUMPS401 must generate a clock signal to drive the RX_CLK and read from the RXD data bus right after every positive edge of the RX_CLK. At the same time, the RUMPS401 has to monitor the RX_IQSEL generated by the Lime LMS6002D to determine whether the data being read belongs to the I or Q component of the signal. This signaling requirement is analogous to the signaling done on the transmit side, except that the IQSEL signal is driven by the Lime LMS6002D, and that the RUMPS401 must read from the data bus at twice the rate the RUMPS401 write into the data bus. As described in Table 5.1, clock signals frequency for both transmit path's DAC and receive path's ADC are identical, except the data bus read rate which is set at 16kbps to satisfy the Nyquist sampling rate.

For signal reception, the Lime LMS6002D down-converts the received signal into baseband analog signal, which is then converted to digital sampling points by the internal ADC based on the defined rate. Ideally, assuming the absence of noise, the down-conversion process should separate the baseband data signal from the carrier, hence perfectly recovering the original baseband signal generated by the transmitter. This can only be achieved if the local oscillators in transmitter and receiver are perfectly synchronized and generate carrier signals with the exact same frequency. Such a condition is impractical since each oscillator has characteristic variations that affect the generated frequency, resulting in signals whose frequency slightly differs. The frequency

mismatch between the transmitter and the receiver causes imperfect down-conversion, yielding baseband data signal that is shifted in frequency (Jr, Sethares, & Klein, 2011). The frequency shift affects the signal in a way that the signal's IQ values are rotated in the IQ plane by certain angle at certain rotation rate, which is undesirable. It is possible to synchronize the two oscillators, but again it is impractical as in real world the wireless transmitter and receiver are spaced apart with no reliable means for synchronization (Jr, Sethares, & Klein, 2011).

As mentioned above, the down-converted analog baseband signal is sampled by the ADC based on the RX_CLK signal, which is controlled by the RUMPS401. Ideally, the RUMPS401 in the receiver is expected to sample the baseband signal at the exact same time as the transmit sampling point at twice the rate. This way, assuming the absence of noise, the original transmitted signal's shape and values can be recovered at the receiver. However, such a thing requires the clock of baseband processors at both transmit and receive ends to be synchronized. Like the frequency mismatch case, there is no reliable means of synchronization in a real application where the wireless transmitter and receiver are spaced apart. This mismatch between sampling instances at the transmitter and receiver is known as the timing offset (Jr, Sethares, & Klein, 2011), causing the receiver to miss the correct sampling point of the symbol's pulse, which should be at the peak of the pulse.

In this work, both offsets arise. Frequency offset happens due to slight frequency variation of the carrier signals generated by the two Lime LMS6002D internal PLL frequency generators, coupled by the slight frequency difference

of the PLL reference clocks which are sourced by two separated Si5356-EVB clock generator board. Similarly, the timing offset happens due to slight variation of the RUMPS401 clock source, which is the 16MHz crystal oscillator. Hence, frequency and timing offset recovery must be performed by the RUMPS401 on the received signal. The recovery process, widely known as receiver synchronization attempts to estimate the frequency and timing offset between the receiver and the transmitter, and correct the signal based on the estimation (Haykin, 2001).

The complete reception process thus is arranged as follows. After the down-conversion by the Lime LMS6002D, the analog baseband signal is sampled by the RUMPS401 via the Lime LMS6002D internal ADC. This sampling process runs in real time as the analog baseband signal is not buffered. During the sampling process, the RUMPS401 also estimates the timing offset based on the received samples and adjust the sampling instance accordingly. This timing offset correction process is performed only for the first 16 bits of the transmit frame, which as described in Figure 5.18 is the preamble augmented specifically for that process. By then, the RUMPS401 is assumed to have the correct sampling instance. The RUMPS401 then proceed with the reception and buffers the rest of the frame, which as shown by Figure 5.18 consists of the frequency preambles and the data bits. The amount of frequency offset experienced by the signal is estimated with the help of frequency preambles, which is then used to correct the data bits.

Note that timing offset recovery is carried in real time along with the sampling of the received data, while frequency recovery is performed on the

buffered data. For each received pulse corresponding to a single symbol in the frame, the timing offset recovery requires multiple sampling points of that pulse to calculate and decide on the best sampling point. On the other hand, the frequency recovery only requires the symbol value, represented by one sampling point assumed to be the peak of the pulse. From Table 5.1 it is evident that the actual data rate of 2kbps is oversampled at the receiver by eight times at 16kbps. If the whole frame is buffered first before the timing offset correction, the memory requirement would be at least eight times the number of bits in the frame, which is unaffordable for the RUMPS401. Hence, the sampling instance correction must be done as the data is being received.

Recall from Chapter Four that the Turbo decoder software operates in fixed-point arithmetic with S16.15 number format. The same fixed-point data type is also used in the frequency offset recovery process. The frame detection and timing offset recovery do not require any decimal point number, thus can be performed with integer data type. The rest of this subchapter describes in detail the coherent BPSK receiver implementation, broken into several sections as follows. The first section discusses the frame detection. Section two describes the timing offset recovery, while the frequency offset recovery is described in section three. The fourth section presents the LUT-based sine and cosine functions which are essential for the frequency offset recovery. Lastly, section five discusses the BPSK receiver memory requirements.

5.3.1 Frame Detection

In this work, the transmission is done in burst, i.e. a transmit frame is constructed at the sending side and transmitted over the air without informing the receiver beforehand about the transmission. The receiver must be able to detect the transmission correctly at the beginning of the frame and perform the reception process as described above. In this work, once configured the Lime LMS6002D will continue to monitor transmission on the desired frequency and perform the down-conversion into baseband signal. Should transmission occur, the Lime LMS6002D will receive the transmitted signal and provide the digitally sampled signal to the RXD data bus. The RUMPS401 as the baseband processor hence must monitor the data bus and trigger the reception process accordingly.

A simple instantaneous power-based frame detection mechanism is implemented in the RUMPS401. As mentioned earlier, the sampling is done at 16kbps, producing eight samples for each pulse corresponding to a single symbol in the transmit frame. Each sampling point consists of an I and a Q values read from the RXD data bus. The start of a transmit frame is detected if the sum of the instantaneous power for three consecutive sampling points reach a certain threshold. Instantaneous power for an IQ sample is defined by the sum of its squared magnitude, $I^2 + Q^2$. Recall from Figure 5.18 that the first 16 bits preambles in the transmit frame alternate between one and zero. The detection goal is to find the first slope of the pulse corresponding to the first preamble, as shown in Figure 5.19.

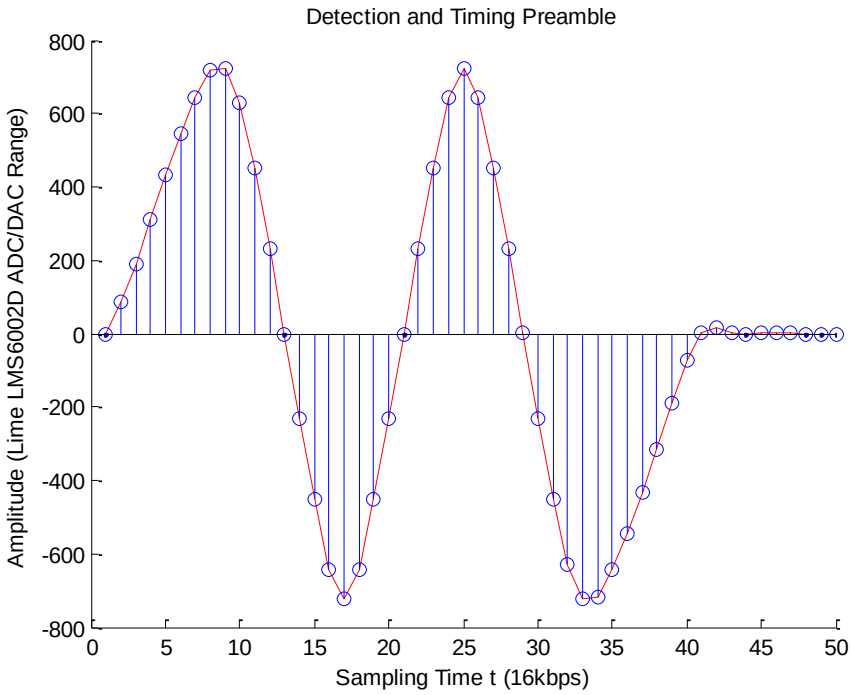


Figure 5.19 Frame Detection and Timing Offset Correction Points

This detection task is jointly performed by the IO Control Core and the DSP Core. Role of both cores in the task is straightforward. The IO Control Core is responsible for reading the I and Q value from RXD data bus, summing the instantaneous power of each sample point, and comparing them to the detection threshold. Instantaneous power computation is done by the DSP Core since it involves multiplications which can be performed quickly with the Multiply-Accumulator (MAC) hardware. Data transfers between the IO Control Core and the DSP Core are carried through the internal Network-on-Chip (NoC). As shown in Figure 5.19, once the detection threshold is reached the IO Control Core triggers the next step of the reception process, the timing offset correction which starts at the next sampling point. The frame detection process is summarized in Figure 5.20.

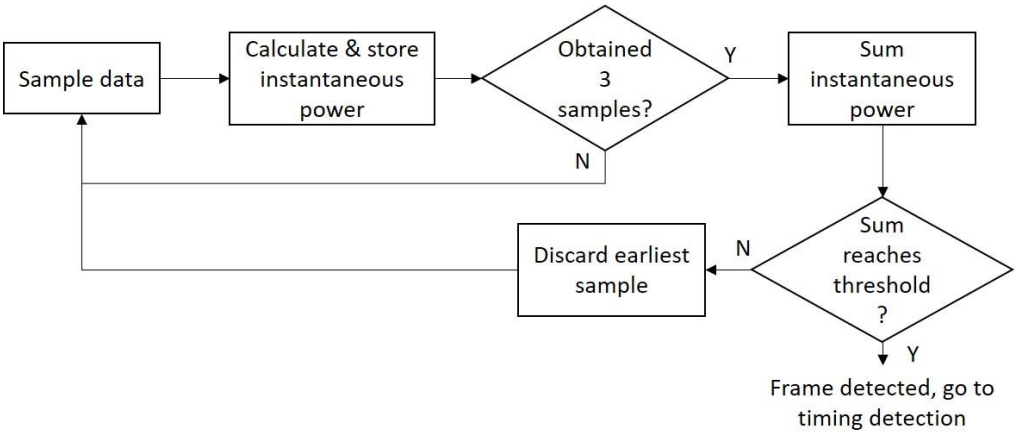


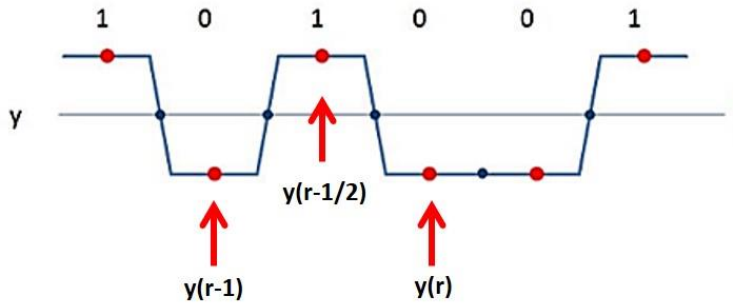
Figure 5.20 RUMPS401 Frame Detection Flow

5.3.2 Timing Offset Recovery

As discussed earlier, the timing offset recovery process must be done while the RUMPS401 is sampling the digital data from RXD bus at 16kbps rate. Considering the computational resource of the RUMPS401, the timing recovery method should be kept simple while providing reasonable performance. Gardner timing error detection (Gardner, 1986) satisfies these requirements, hence is adopted in this work. Figure 5.21 depicts the original algorithm of timing offset estimation developed for BPSK and QPSK. The algorithm only requires sampling rate of at least two samples per symbol.

The estimation formula uses three strobe points of the signal, each spaced by half of single symbol duration. These three strobe points are referred as the start, mid, and end points. As shown by the figure, should the sampling happen right at the peak of each symbol, the mid-point $y_I(r - 1/2)$ should be zero hence yielding the offset estimation zero. Should the sampling happen before or after the peak, the offset estimation formula results in negative or

positive value which indicate whether the sampling happen too early or too late. In the calculation, offset for the I and Q signals are estimated separately and summed to yield the symbol offset estimation. The Gardner estimation works the best on a periodic signal, which explains the arrangement of the timing preamble in this work.



$$TimingOffset = y_I\left(r-\frac{1}{2}\right)[y_I(r)-y_I(r-1)] + y_Q\left(r-\frac{1}{2}\right)[y_Q(r)-y_Q(r-1)]$$

Figure 5.21 Gardner Timing Error Estimation (Boschen, 2016)

As mentioned before, the timing detection starts at the next sampling points following the successful frame detection. The IO Control Core uses this sampling point as the start strobe point. Since the receiver oversamples the symbol by eight times, the mid and end strobe points are two sampling instances which are four and eight points away from the start strobe point, respectively. These three strobe points are sent to the DSP Core for calculation, which then return the result back to the IO Control Core. Adjustment of the sampling point is done by the IO Control Core based on the result in the following fashion. Positive estimation result indicates late sampling, hence the sampling of the next symbol is adjusted to be one sampling point earlier than the current point.

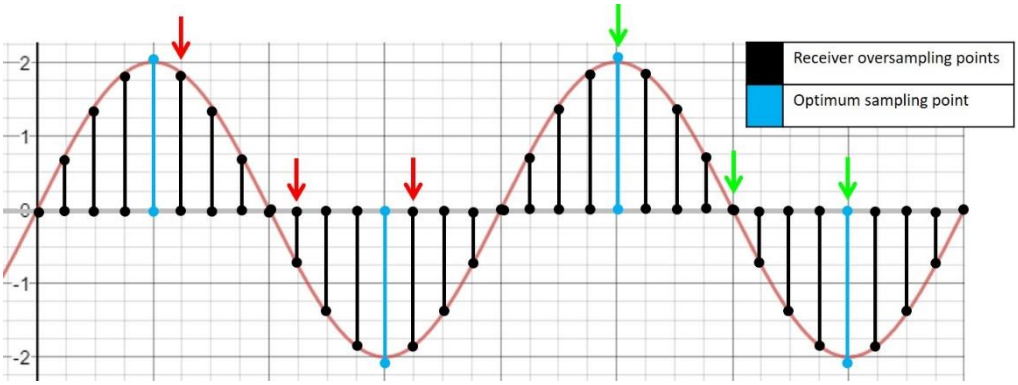


Figure 5.22 RUMPS401 Implementation of Gardner Timing Correction

Inversely, negative estimation result indicates early sampling, thus the IO Control Core moves the next symbol sampling to one sampling point later than the current point. Figure 5.22 illustrates the timing correction process. For the first pair of the timing preamble, the Gardner algorithm estimates a late sampling hence adjusts the strobe points of the next pair to be one sampling point earlier than the current one. The estimation is performed for all pairs of one-zero in the timing preamble, whose cumulative correction defines the sampling instance for the rest of the frame. Figure 5.23 summarizes the timing offset recovery process.

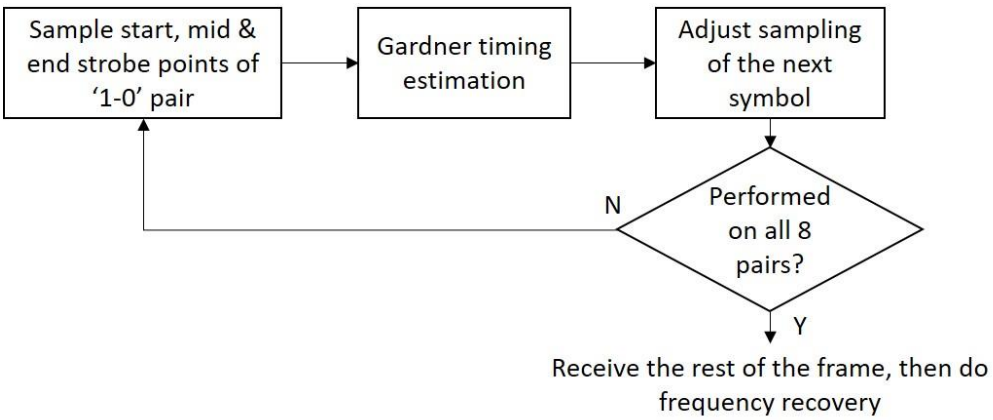


Figure 5.23 RUMPS401 Timing Offset Recovery Flow

5.3.3 Frequency Offset Recovery

As mentioned earlier, frequency offset experienced by the received signal results in rotation of the symbols in IQ plane by a certain angle. The amount of rotation experienced by the symbols are unknown to the receiver. To correct the frequency offset on received symbols, the receiver must estimate the amount of rotation and act accordingly. In this work, a simple cross-product phase detector is implemented, depicted in Figure 5.24. The detector works by comparing angle between two vectors, in which one vector is the received symbol and another one is the referenced symbol. In this case, the referenced symbol is the frequency preambles.

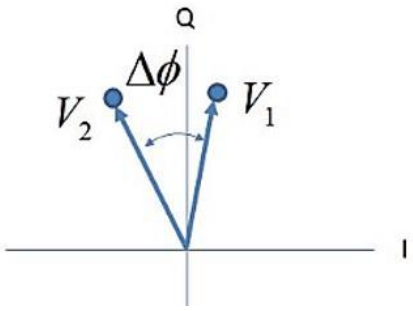


Figure 5.24 Cross-product Phase Detector (Boschen, 2016)

The angle between the two vectors is obtained using the formula in equation 5.1a, which only involves two multiplications and one subtraction. However, the phase detector imposes two problems. First, it can only detect accurately up to angle difference of 90°. Second, the $arcsin$ function is expensive to implement. For larger angle, such approximation cannot be used. The first problem is solved by moving the received symbol to the correct quadrant of the IQ plane before running the detector. In this work, the bits in

frequency preamble are known to be 1s, whose corresponding symbol based on the BPSK modulation scheme is located at the first quadrant.

$$\Delta\emptyset = \arcsin(I_1 Q_2 - I_2 Q_1) \qquad \textbf{(5.1a)}$$

$$\Delta\emptyset = (nLoop - 1) \cdot \left(\frac{\pi}{32}\right) + \Delta\emptyset_{nLoop} \qquad \textbf{(5.1b)}$$

$$\begin{aligned} I(i)_{corrected} &= I(i)_{recv}.\cos(\Delta\emptyset) + Q(i)_{recv}.\sin(\Delta\emptyset) \\ Q(i)_{corrected} &= I(i)_{recv}.\sin(\Delta\emptyset) - Q(i)_{recv}.\cos(\Delta\emptyset) \end{aligned} \qquad \textbf{(5.1c)}$$

The second problem is solved by exploiting the properties where the $arcsin$ operation can be neglected for small angle since the angle and its sine values are almost equivalent. The phase detection hence is improved by setting it in a loop with a subtractor and deciding on any small angle value satisfying the properties above. In this work, $\pi/32$ is chosen. The loop runs phase detection on the received symbol, and if the detected angle is larger than $\pi/32$, it loops back to the phase detector with input of the detected angle subtracted by $\pi/32$. The loop continues until the detected angle is equal or lower than $\pi/32$. The total angle between the two vectors thus can be stated as equation 5.1b, where $nLoop$ is the number of loops performed and $\Delta\emptyset_{nLoop}$ is the angle detected at the last loop. This way, the actual $arcsin$ operation is avoided while still retaining the calculation accuracy since the angle detection is performed in small angle step of $\pi/32$. The correction hence is performed on the IQ value of the received symbol based on equation 5.1c.

Assuming that the frequency offset experienced by the signal is constant, it causes the symbols to be rotated by certain angle with certain speed,

e.g. 5° per second. For example, if the symbol rate is one symbol per second then the first symbol will be rotated by 5°, the second by 10°, the third by 15°, and so on. If the first symbol is rotated one time, the second symbol experienced two rotations, the third symbol experienced three, and so on. Based on this behavior, the frequency offset experienced by the symbols can be defined by two rotation angles. The first is the certain angle that rotates the symbols over time, denoted as $\emptyset_{fix}$. The second is the amount of random rotation experienced by the first symbol, denoted as $\emptyset_{rand}$.

While it is true that the constant frequency offset results in each symbol being rotated with the same angle over time, by the time the symbols phase offset is calculated there is no information of how many times those symbols have been rotated. The frequency offset experienced by the symbols thus can be restated as follows. The first symbol is rotated by $\emptyset_{rand}$, while the second symbol is also rotated by $\emptyset_{rand}$ plus $\emptyset_{fix}$. Likewise, the third symbol is rotated by $\emptyset_{rand}$ plus $2\emptyset_{fix}$. This pattern of $\emptyset_{rand} + k.\emptyset_{fix}$ continues for the rest of the symbols, where k denotes the symbol index.

This work implements a simple frequency recovery scheme based on the phase detector and the two fixed rotation angles described above. The first and the second bit in the frequency preambles is used by the phase detector to obtain $\emptyset_{rand}$ and $\emptyset_{fix}$, respectively. These two angles are then used for correcting the rest of the symbols in the frame. This correction method is quite reliable for a small number of symbols, and deviates for larger number of symbols as the frequency offset also fluctuates over longer time. Large deviation causes error

during correction which in turn causes the increment of error rate. It can be resolved by recalculating the $\emptyset_{rand}$ and $\emptyset_{fix}$ every n symbols, which also requires re-insertion of frequency preamble after every n symbols. The multiple frequency preambles in the transmit frame format serves this exact purpose, where the choice of n defines the trade-off between data rate and correction accuracy.

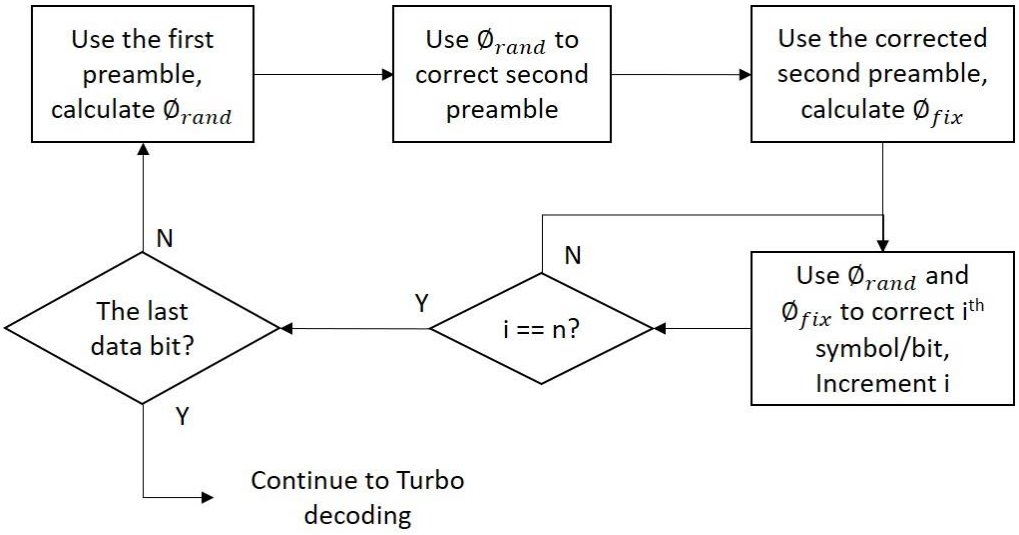


Figure 5.25 RUMPS401 Frequency Offset Recovery Flow

Figure 5.25 summarizes the frequency offset recovery process. As in the frame detection and timing offset recovery, the task is split between the IO Control Core and the DSP Core. The IO Control Core performs the data retrieval and indexing while the phase estimation and correction is loaded to the DSP Core due to the multiplications. This frequency offset recovery process requires fast operation, but not timing critical as the data has been buffered by the RUMPS401 during the receive process. Note that the phase correction computation involves trigonometry functions which are not natively supported

by the ARM Cortex-M0 nor the RUMPS401 hardware accelerators. Implementation of the trigonometry function is detailed in the next section.

5.3.4 Lookup Table-based Sine and Cosine Function

As the RUMPS401 is not equipped with any hardware supporting trigonometry function, the sine and cosine operations are implemented with a simple lookup table and the corresponding mapping function. The lookup table consists of 256 points, storing the sine values for angle zero to 2π in step of $\pi/128$. For every input angle, the mapping function finds the sine of that angle by looking for an index in the lookup table which contains the best approximate of its real sine value. Equation 5.2a defines the mapping function. $LUTsize$ defines the lookup table size which is 256, while $MaximumAngleValue$ defines the largest angle whose sine value is stored in the lookup table which is 2π. $LUTsize$ and $MaximumAngleValue$ hence can be substituted with fixed value of 256 and 205887, respectively. The value of 256 is the number of points in the lookup table, while 205887 is the integer representation of 2π in S16.15 fixed-point number format. Substituting these two values, equation 5.2a thus can be restated as equation 5.2b. This mapping function consists of only two operands, the $inputAngle$ and a constant of 0.0012434.

$$LUT\ index = \frac{inputAngle \times LUTsize}{MaximumAngleValue} \quad \textbf{(5.2a)}$$

$$LUT\ index = inputAngle \times 0.0012434 \quad \textbf{(5.2b)}$$

The $inputAngle$ value is the integer representation of any angle between zero and 2π in S16.15 fixed-point format. For example, angle zero, π, and 2π are represented as zero, 102943, and 205887, respectively. Thus, the fixed-point data type should support an integer value whose value ranges from zero to 205887 and a small decimal value of 0.0012434. These two numbers are clearly out of the S16.15 data type range. Its integer part only supports amplitude up to 65536, while the fractional part only provides precision of 0.00003. These values hence must be stored in a 64-bit variable, arranged as unsigned 32.32 fixed-point number. Both the integer and fractional parts can cover the number amplitude and precision requirements, while the sign is unnecessary for the sine function lookup.

The GNU Compiler Collection (GCC) fixed-point extension does not provide this data type. Additionally, the MAC hardware could only support up to 32-bit multiplication, hence the 64-bit multiplication is performed by splitting it into four 32-bit multiplications, as shown by pseudocode in Figure 5.26. It simply splits the operands to their lower and upper half bits and calculates the multiplications among them accordingly. The multiplication only calculates the upper half of the result since the lower half only contains the fractional part which is not significant for finding the lookup table index.

```
unsigned_int64_t a
unsigned_int64_t b

unsigned_int32_t a_low = lower_half(a)
unsigned_int32_t a_high = upper_half(a)
unsigned_int32_t b_low = lower_half(b)
unsigned_int32_t b_high = upper_half(b)

unsigned_int64_t p0 = a_low * b_low
unsigned_int64_t p1 = a_low * b_high
unsigned_int64_t p2 = a_high * b_low
unsigned_int64_t p3 = a_high * b_high

unsigned_int32_t carry = upper_half(upper_half(p0) +
                         lower_half(p1) +
                         lower_half(p2))
unsigned_int32_t result_high = p3 +
                               upper_half(p1)+
                               upper_half(p2) +
                               carry
```

Figure 5.26 64-bit Multiplication Pseudocode

The lookup table only contains sine value for positive angle between zero and 2π, hence mapping function must also consider various inputs such as negative values or values larger than 2π. These problems are solved by utilizing the same lookup table and trigonometry properties. Likewise, cosine of any input angle can be derived in the same way. Due to the number of multiplications involved in the mapping function, the LUT-based sine and cosine functions are also performed by the DSP Core.

5.3.5 Memory Requirements of the Complete Receiver

As in the Turbo decoder implementation, the memory requirement of the BPSK receiver is also computed to allow the whole software to fit into the RUMPS401 memory resources. Table 5.2 lists the memory usage of the receiver program. As shown by the table, the largest memory usage is imposed by the buffering of the received signal, which consists of both the I and Q components. It is evident that if the I and Q components are stored in only a single core, the memory required for the signal alone almost consumes the whole the 8kB

SRAM of that core, where in this case is the IO Control Core. Hence, the received signal is buffered in two different cores.

Table 5.2 RUMPS401 BPSK Receiver Memory Usage

The RUMPS401 Core	Variable	Required number	Actual size in bytes
IO Core	Received frame data (I component)	768 + 2(768/n)	At least 3080 bytes
DSP Core	Sine LUT	256	1024 bytes
Normal Core 0	Received frame data (Q component)	768+ 2(768/n)	At least 3080 bytes
Normal Core 1	Does not participate	-	-

The sine lookup table also consumes a considerable amount of memory from the DSP Core. For this case, the sine lookup table is not stored in other cores such as the Normal Core 1 due to performance consideration. If the lookup table is stored in another core, there will be overhead required by the DSP Core to get the lookup values via the NoC. Likewise, storing the lookup table in the local flash memory also introduces delay since the RUMPS401 flash access time is much slower compared to the RAM access. Coupled with the Turbo decoder memory usage as presented in Table 4.2, both the BPSK receiver and Turbo decoder fits into the RUMPS401 memory resources.

5.4 Summary

This chapter presented in detail the implementation of coherent-BPSK transmitter and receiver software in the RUMPS401 quad ARM M0 cores SoC. Connection and digital signaling between the RUMPS401 as the digital

baseband processor and the Lime LMS6002D as the programmable radio frontend are discussed as well. It was shown that based on the signaling required by the Lime LMS6002D, a digital baseband modulation-demodulation scheme can be implemented in the RUMPS401. The transmitter provides actual data rate of 2kbps, which is oversampled by four times during the pulse shaping process. The receiver satisfies the Nyquist criteria by sampling the signal at 16kbps which is twice the pulse shaping rate. To support these rates, 45° offset BPSK is implemented as the I and Q values are equal, hence lowering the signaling rate at which the RUMPS401 needs to cope.

The wireless system built in this work is designed to transmit in separated frames, where each frame consists of data bits augmented with synchronization preambles. A preamble-aided receiver synchronization scheme was devised, consisting of the transmit frame detection, the timing offset recovery, and the frequency offset recovery. The synchronization scheme stresses on simple computations and approximations by utilizing the RUMPS401 resources judicially. The analysis of memory consumed by the BPSK receiver is also provided, which shows that it can co-exist with the Turbo decoder software. This allows possible integration into a complete receiver system which shall be detailed in the next chapter.

CHAPTER 6

COMPLETE TRANSCEIVER SYSTEM INTEGRATION AND TESTING

Implementation and optimization of Turbo Code software on the RUMPS401 has been discussed in Chapter Four. The fifth chapter has detailed the software implementation of the BPSK modulator and demodulator based on the Lime LMS6002D signaling requirement and the RUMPS401 processing capability. This chapter presents the integration of both parts into a complete wireless transceiver system, along with the verification of its functionality through real wireless transmission and reception test. As discussed in the first chapter, this work utilizes the RUMPS401 as the baseband processor, which due to its inherently low processing power would not yield competitive performance on both the Turbo Code and the BPSK modulation-demodulation scheme. Hence, the testing is performed to verify that the SDR-based transceiver system built in this work is fully functional on the physical layer level with reasonable performance.

The rest of this chapter is arranged into three sections as follows. Section one describes the software-side integration of the Turbo Code and the BPSK modulator-demodulator, along with the physical setup of the complete transceiver module. The second section presents the system testing procedures as well as its results and analysis. Lastly, section three summarizes the chapter.

6.1 Transceiver System Integration

As mentioned in previous chapters, a single transceiver module is composed of the RUMPS401 as the baseband processor and the Lime LMS6002D which performs all necessary analog signal processing and transmission. The Turbo Code software runs regardless of the Lime LMS6002D specification, while the BPSK modulator-demodulator software was designed with interfacing against Lime LMS6002D in mind. However, both software was developed with the RUMPS401 resources as the most crucial consideration.

To allow the transceiver module to function, the RUMPS401 is required to run the Turbo Code alongside the BPSK modulator-demodulator software, while also interfaces with an application processor on which the transmit data is generated and received. The two software as well as the interface to the application processor therefore must be integrated to allow a fluent and repetitive function of data transmission and reception. Furthermore, the hardware setup is as important to allow the actual over the air wireless transmission and reception. This subchapter discusses these things in separate sections presented below.

6.1.1 Software Integration of Turbo Encoder and BPSK Modulator

Figure 6.1 depicts the complete flow of the software running on the RUMPS401 for the transmitter module. As shown in the figure, the software comprises of two phases, the setup process and the transmitter loop. The setup starts with the RUMPS401 initializing its internal timer and variables required

for both the Turbo Code and BPSK modulation-demodulation. It then waits for a signal from the application processor instructing the configuration and calibration of the Lime LMS6002D, which are carried according to the procedure discussed in Chapter Five and the Radio Frequency (RF) specification of the wireless system. Specification details such as the carrier frequency and transmission bandwidth shall be discussed in the next subchapter during the system test setup. Once the Lime LMS6002D is initialized, the software goes into the transmitter loop.

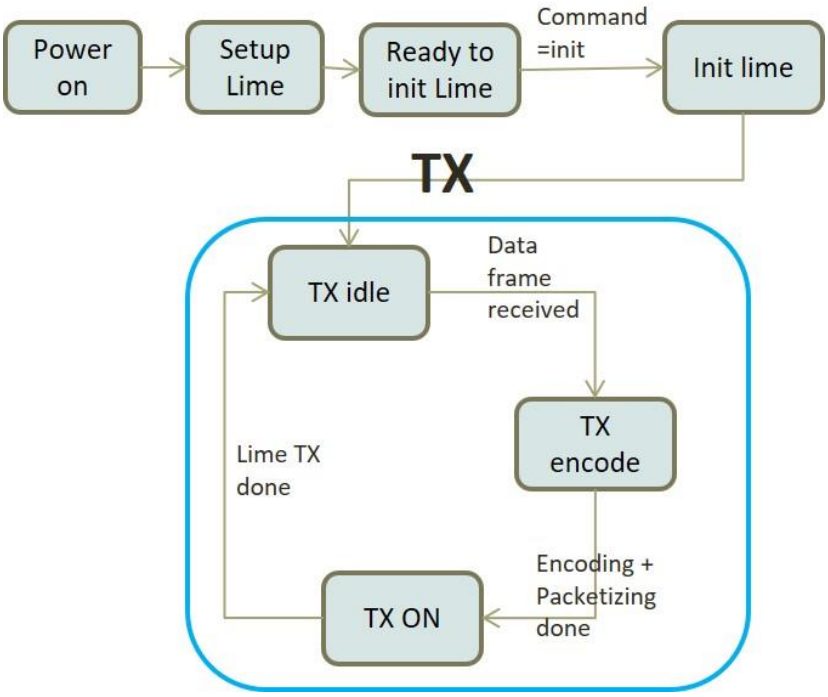


Figure 6.1 Transmitter Software Flow

The transmitter loop comprises of three main states, which are the idling state, the encoding state, and the transmitting state. Idling state is the first state entered by the transmitter software after the Lime LMS6002D initialization. In this state, the RUMPS401 waits for 256-bit data frame from the application processor, which upon its reception causes the software to transition into the

encoding state. In the encoding state, the data frame is Turbo-encoded into a 768-bit codeword and augmented with the synchronization preambles to form the transmit frame. The RUMPS401 then enters the transmitting state where it starts its internal timer, performs the in-time pulse shaping calculation according to the transmit frame, and drives the pulse values into the Lime LMS6002D for transmission as described in Chapter Five. Once the last pulse of the transmit signal is sent, the RUMPS401 turns off its timer and re-enters the idling state, in which it is ready for the next data frame. The loop continues until the transmitter module is reset or powered off.

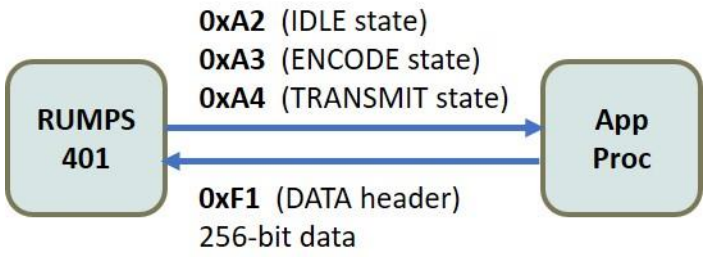


Figure 6.2 Transmit Side RUMPS401-Application Communication

The transmitter software only accepts data frame from the application processor in the idling state and treats each transmitted frame as completely separated data, i.e. it does not retain information regarding past transmissions. Should a transmission of two identical frames is required, the application processor must send the original data frame to the RUMPS401 twice which then send them on two distinct burst transmissions, even if the retransmissions are consecutive. The RUMPS401 and the application processor which in this case is a Python-based program running on a desktop PC are communicating based on a simple protocol defined in Figure 6.2.

The protocol requires data exchange to be done with a set of defined codes known to both parties. As shown by the figure, the codes are simply the RUMPS401 software state represented by 8-bit values, except for one. The state-based codes are used by the RUMPS401 to update its current state to the desktop program hence it can act accordingly, i.e. keeping track of the transmission and sending the data frame only when the RUMPS401 is in idle state. The only code that is not based on the RUMPS401 state is the data frame header which is sent by the application processor to the RUMPS401, indicating the start of the 256-bit data frame transfer. Communication between the RUMPS401 and the desktop program is performed via Universal Asynchronous Receiver-Transmitter (UART) interface, which is part of the transceiver hardware detail described in later section.

6.1.2 Software Integration of Turbo Decoder and BPSK Demodulator

Shown in Figure 6.3 is the complete flow of the receiver software running on the RUMPS401. Similar to the transmit side, the software consists of two phases, the setup process which is identical to the transmit side and the receiver loop. The only difference of the setup process between the transmitter and the receiver modules is the active chain. The transmitter module only activates the Lime LMS6002D transmit chain, while the receiver module only activates the receive chain. This is straightforward since the wireless system implements one-way communication. Additionally, activating only one chain reduces power usage by half since each chain consumes around 260mA. After the Lime LMS6002D configuration and calibration, the receiver software enters the receiver loop.

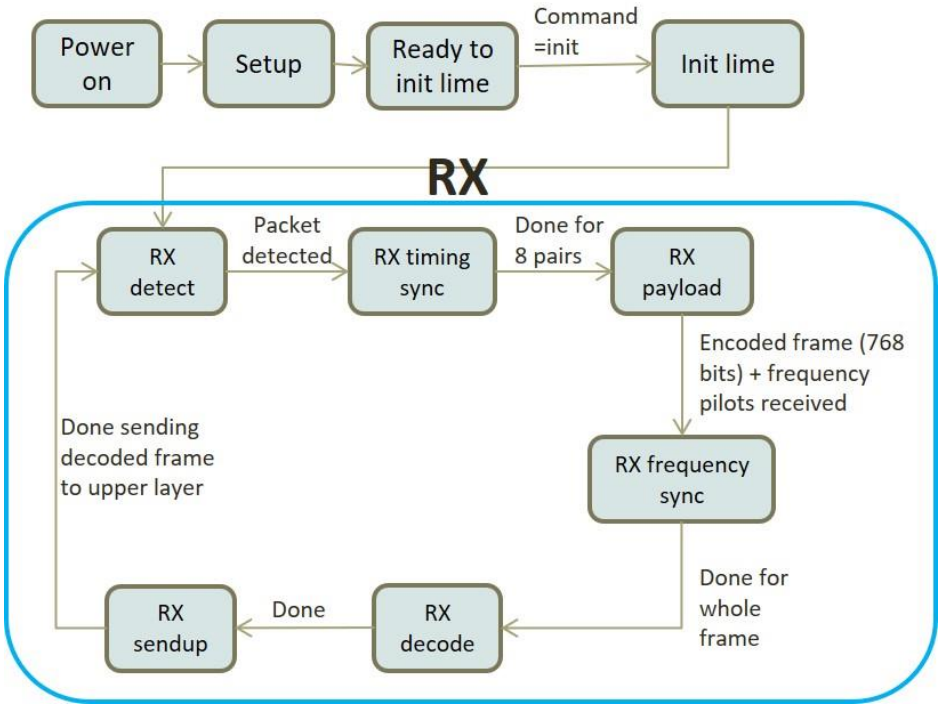


Figure 6.3 Receiver Software Flow

The receiver loop consists of six distinct states, which are the detection state, the timing correction state, the payload retrieval state, the frequency correction state, the decoding state, and the sending state. Upon the Lime LMS6002D initialization process, the RUMPS401 enters the detection state where it monitors start of any transmission through the signal received by the Lime LMS6002D. Once the start of a frame or transmission is detected, the RUMPS401 enters the timing correction state for synchronizing the receiver sampling instance. As described in Chapter Five, the timing correction is performed with the help of 16 bits of timing preambles located in the beginning of the frame. The RUMPS401 then enters the payload retrieval state where it samples and buffers the rest of the frame based on the timing corrected on the previous state. Operations within these three states are timing critical since they are performed while receiving the signal.

$$Noise\ variance = \frac{\sum_{k=1}^{N}(x_k)^2}{N} - \left(\frac{\sum_{k=1}^{N}|x_k|}{N}\right)^2 \qquad \textbf{(6.1)}$$

The other three states in the receiver loop operates around the buffered data and have less timing constraint. In the frequency correction state, the buffered data is re-rotated based on the phase offset calculated from the frequency preambles, as described in Chapter Five. The RUMPS401 then enter the decoding state where it decodes the frequency-corrected data based on the Turbo decoder software discussed in Chapter Four, preceded by estimation of noise variance based on those data. The noise variance estimation is performed using a simple formula provided in equation 6.1 (Reed & Asenstorfer, 1997), where x_k denote the k^{th} data bit and N denote the total data bit, which in this case is 256. Since the formula includes a number of multiplications, it is cooperatively performed by the IO Control Core and the DSP Core in a similar fashion to the joint-calculation described in previous chapters, i.e. the multiplications are loaded to the DSP Core's MAC hardware. The estimated noise variance value is then used for the rest of decoding process. Continuing to the sending state, the RUMPS401 uses the resulting likelihood ratio for hard-decision of the bits value and passes those values to the desktop program via UART interface before returning to the detection state.

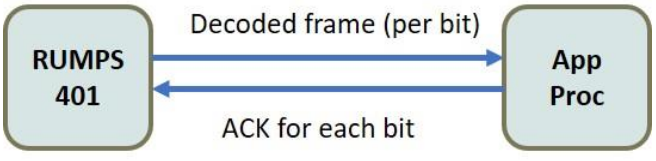


Figure 6.4 Receive Side RUMPS401-Application Communication

As in the transmit side, the application processor is a simple Python program running on a desktop PC. Contrary to the transmit side that requires protocol for communication between the RUMPS401 and the application processor, the receive side is much simpler since the communication is only one way, i.e. from the RUMPS401 to the desktop program. The only feedback from the application processor to the RUMPS401 is the acknowledgement for flow control during transfer of the received data bit. Figure 6.4 depicts the simple communication between the RUMPS401 and the desktop program on the receiver side, which implements no protocol. Details of the desktop program for both transmitter and receiver end are presented in the next subchapter as the programs are developed for testing purpose.

6.1.3 Transceiver Hardware Setup

Figure 6.5 depicts a detailed single transceiver module setup. The RUMPS401 connects with the Lime LMS6002D's digital baseband line through the compact Hirose FX10A-80A connector, along with the 4-pins SPI connection for register configuration. Silicon Lab's Si5356-EVB clock generator board supplies a 40MHz clock signal to the Lime LMS6002D as the PLL reference clock. The RUMPS401 is connected to the PC via a USB-to-UART module, which provides translation between the transistor-transistor-logic (TTL) signal on the RUMPS401 side and USB signal on the PC side. This facilitates communication between the RUMPS401 and the Python-based transmitter or receiver program, based on the UART signaling protocol.

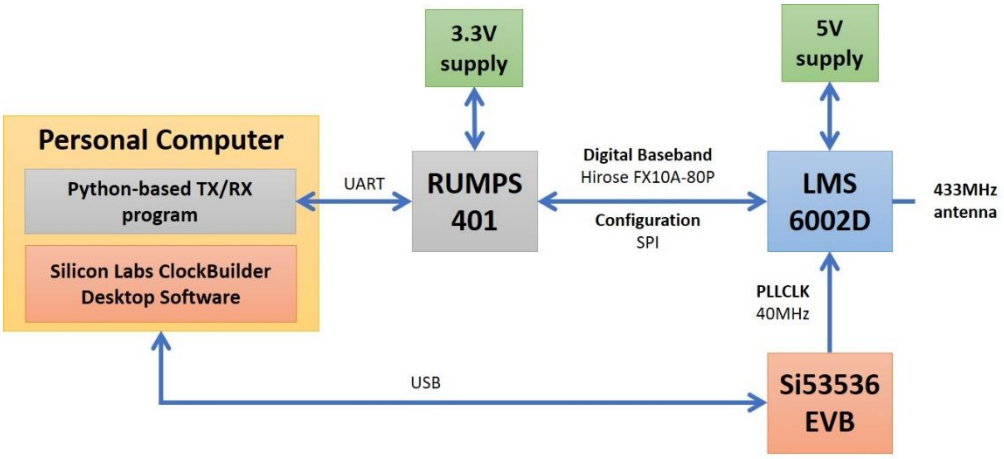


Figure 6.5 Wireless Module and Application Processor Setup

The PC also runs Silicon Labs ClockBuilder Desktop Software which allows simple management of the clock signal generated by the Si5356-EVB over the USB connection. This software provides graphical user interface (GUI) in which user can configure the output clock frequency by simply specifying the desired frequency. Two separated power supplies unit provides 3.3V and 5V power to the RUMPS401 and the Lime LMS6002D, respectively. Lastly, the Lime LMS6002D RF interface connects to a 433MHz stub antenna from Siretta. The antenna has a 50 ohms input impedance and provides gains of 3dB. The carrier frequency is fixed at the license-free 433MHz due to the widely available antenna support and considering the testing environment, it would have less interference compared to the other 2.4GHz. More details on the testing environment is provided in the next subchapter.

6.2 Transmit and Receive Testing

As discussed previously, testing carried on the transceiver system built in this work is intended for functionality verification. The test measures and

compares the system's receiver error rate under two setups, with and without Turbo Code. This subchapter describes the testing setup and results in three sections. Section one describes the testing environment and the Python-based desktop application developed for the testing. The second section presents the test result of the system without Turbo Code. Finally, section three presents the test result of the system with Turbo Code along with the comparison against the result of section two.

6.2.1 Transceiver Test Environment Setup

The test was carried in a simplex configuration with two wireless modules, each set up with a PC as shown in Figure 6.5. One module acts as transmitter while the other acts as receiver, both running the respective software as described previously. In this test, the error rate is measured by calculating the bit error rate of forty 256-bit data frames that has been pre-randomized and stored by both the transmitter and receiver. The transmitter desktop application loads those frames and send them over on forty burst transmissions. Upon a complete reception process, the receiver desktop application compares the received frames against the stored frames and computes the bit error rate.

Due to the receiver requiring more time for the frequency correction and Turbo decoding, the transmitter should insert sufficient delay between transmission of each frame to assure that each frame is received. Note that the transmission and reception are performed on a physical level without any automatic flow control or re-transmission between the modules. Hence, delay between frame transmissions is inserted manually by instructing the transmitter

desktop application to transmit on a keystroke. Figure 6.6 and Figure 6.7 depicts the flow of the transmitter and receiver desktop application, respectively. The two applications are developed to match the software running on the RUMPS401 and the testing scenario.

The transmitter desktop application starts by instructing the RUMPS401 to initialize the Lime LMS6002D. It then waits for a user keystroke before sending a frame to the RUMPS401 for transmission. The application tracks the RUMPS401 software state before allowing the initiation of another frame transmission. This loop continues until all forty pre-randomized frames has been sent. The receiver desktop application starts in the same way by instructing the Lime LMS6002D initialization process. It then waits for the received data frame from the RUMPS401 and calculates the bit error rate based on the comparison against the stored frame.

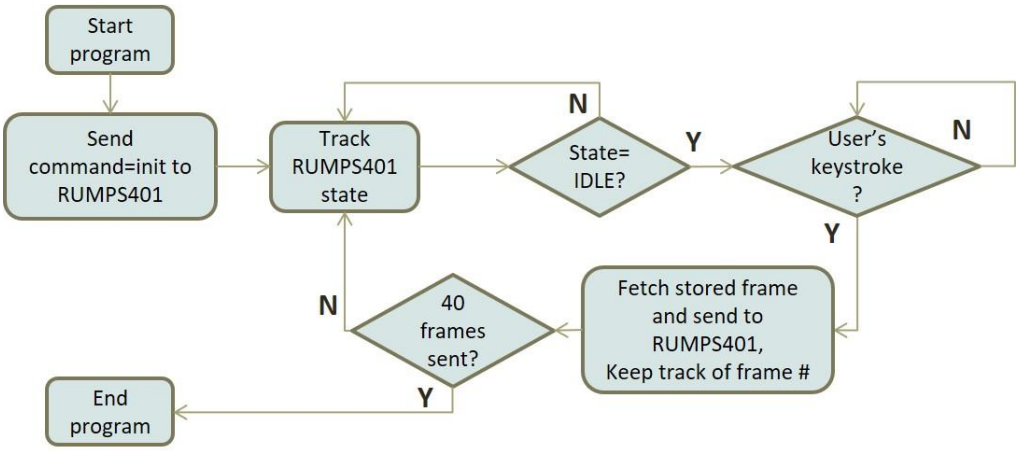


Figure 6.6 Transmitter Desktop Application Flow

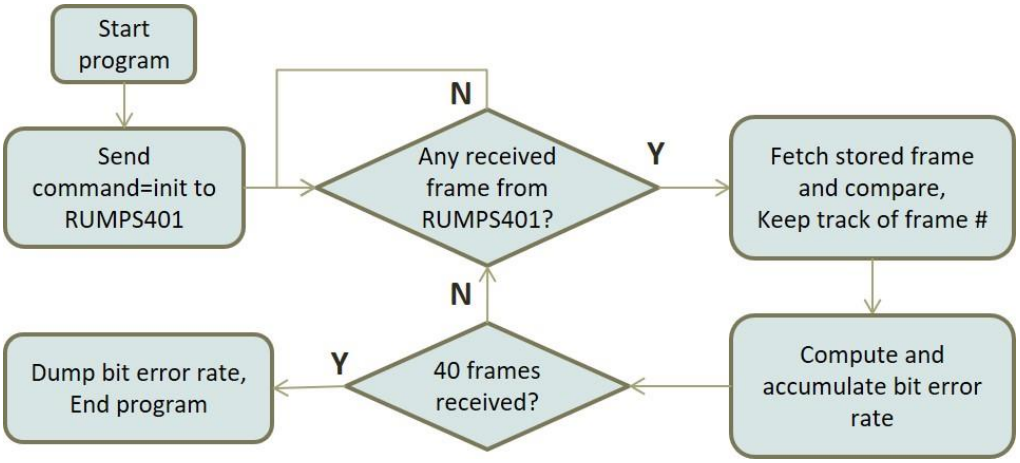


Figure 6.7 Receiver Desktop Application Flow

The two wireless modules operate on 433MHz center frequency and transmission bandwidth of 1.5MHz in a line-of-sight (LoS) configuration. The communication test was performed in a computer lab, in an empty condition out of normal office hour to minimize any interference from other devices. Rather than the license-free 2.4GHz frequency band, 433MHz frequency was chosen as it suffers less interference than the more common 2.4GHz band used by Wi-Fi, Bluetooth, or any wireless peripherals. Throughout the test, the receiver error rate was measured under varying transmit power and modules distance, as well as the number of frequency preambles. Details regarding the test schemes and results are presented in the following sections.

6.2.2 Test of Transceiver System without Turbo Code

The transceiver system test was performed gradually in two steps. First, the BPSK modulator-demodulator functionality was tested by varying the transmit power, the LoS range between modules, and the number of frequency preambles inserted between data. The transmit power is varied by changing the

first stage amplifier of the Lime LMS6002D transmit chain, the TXVGA1 through registers configuration (Microsystem, 2012). Changing the LoS distance is straightforward, as well as the number of frequency preambles insertion.

At this point, Turbo encoder and decoder block was excluded from the transmitter and receiver. The test was performed to verify that the transceiver system behaves properly, such as the error rate dropping as transmit power increase. The second step of the test intends to verify whether the Turbo Code can provide improvement over the non-coded system as the number of iteration increases. This section covers the first test step, while the second is presented in the next section.

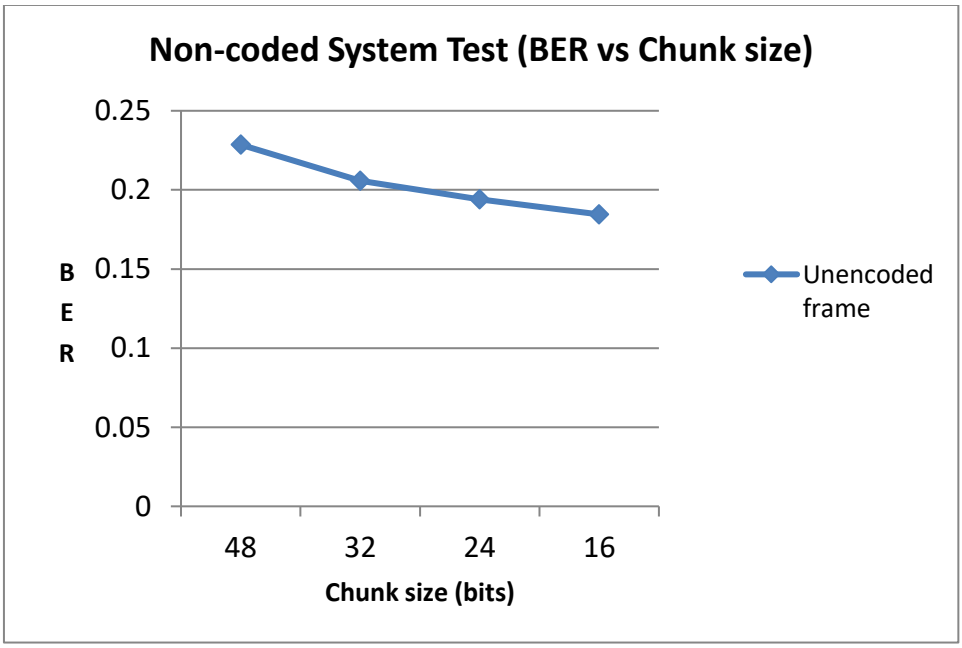


Figure 6.8 Non-coded Test Result for Varying Data Bit Chunk Size

As described in Chapter Five, the transmit frame supports flexible number of frequency correction preambles which provides trade-off between

the correction accuracy and the data rate. Figure 6.8 presents the test result of the varying number of frequency preambles. Each point in the figure represents the cumulative bit error rate for forty frames of 256 bits data. As shown by the figure, the receiver bit error goes down as the data chunk size decreases. Chunk size is the number of data bits between every insertion of the two bits frequency correction preamble. Chunk size of forty-eight means there are sixteen pairs of frequency preamble, each inserted every forty-eight data bits.

The amount of phase offset computed at certain pair frequency preambles is used to correct the consequent data bits in a cumulative manner. The computation introduces certain degree of inaccuracy which accumulates if used for continuous correction and in turn causes the increment of error rate. Hence, it is straightforward that the smaller chunk size yields better error rate due to the more frequent computation of phase offset, which is shown by the test result. In this test, the transmitter transmits with the TXVGA1 fixed at -17dB to the receiving module at one-meter LoS distance.

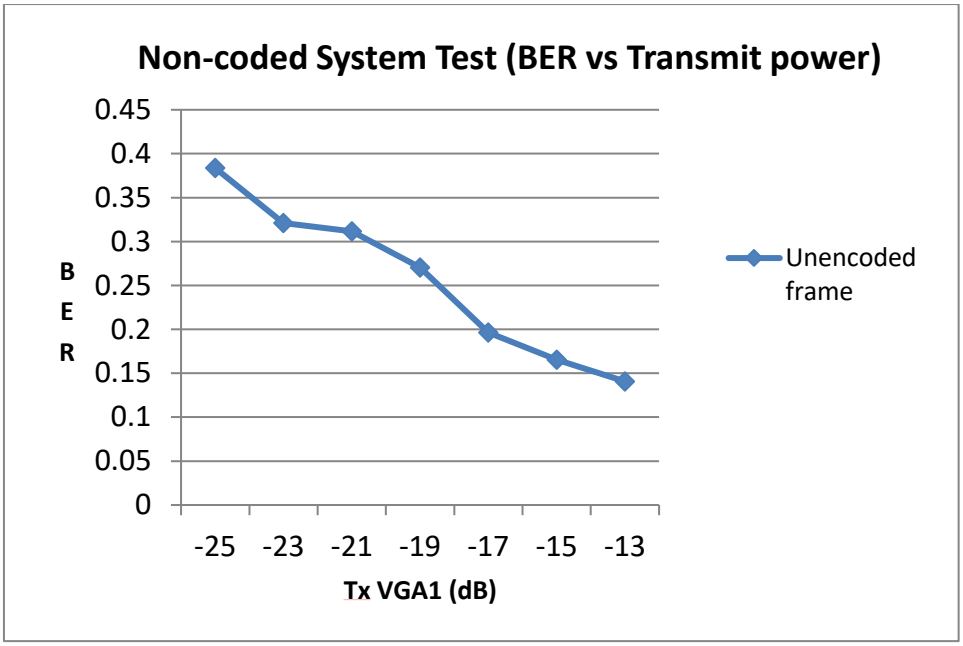


Figure 6.9 Non-coded Test Result for Varying Transmit Power

The test continues by verifying the receiver error rate against varying transmit power, whose result is shown in Figure 6.9. It is evident from the graph that the transceiver system is behaving properly since the error rate goes down as the transmit power increases. In this test, the chunk size is fixed at 24 bits and the distance between the two modules are kept in one-meter LoS distance. Figure 6.10 presents the test result in which the distance between the two modules is varied. The transmitter transmits with the TXVGA1 fixed at -17dB, while the chunk size is fixed at 48 bits. As shown by the graph, the system demonstrated proper behavior of the diminishing error rate as the modules are spaced closer.

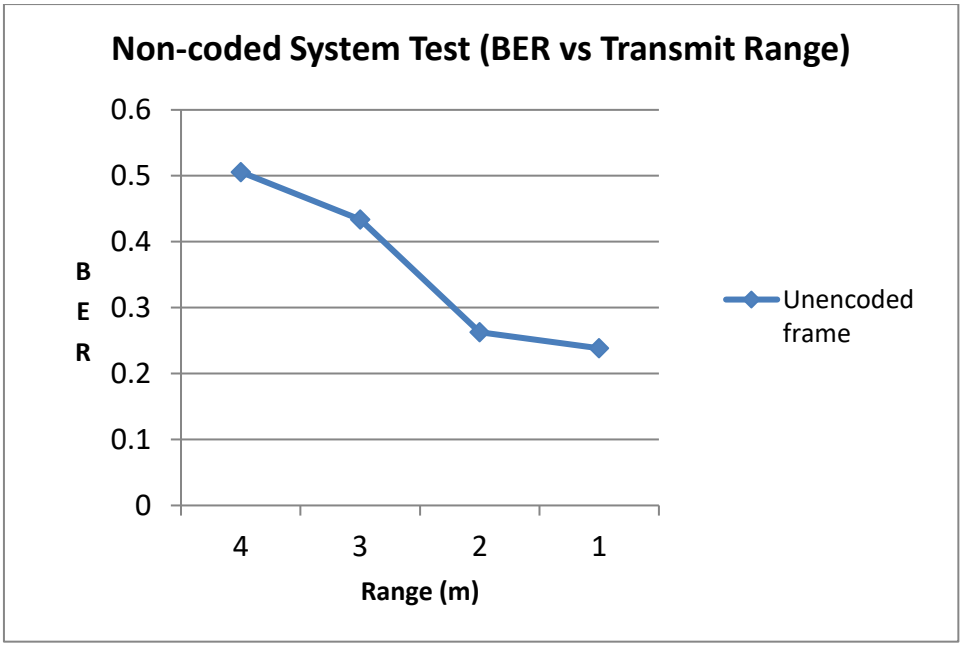


Figure 6.10 Non-coded Test Result for Varying Transmit Range

6.2.3 Test of Transceiver System with Turbo Code

Figure 6.11 shows the test result of the Turbo-coded transceiver system for varying transmit power. The Turbo-coded system error rate performance is tested for one and three Turbo decoding iterations and compared against the non-coded transceiver system under the same setup, i.e. chunk size of 48 bits and one-meter LoS transmit range. As in the non-coded system test, the error rate is calculated as the bit error rate percentage of total of forty 256-bits frames received. As shown by the figure, at the lower TXVGA1 power of -25dB the Turbo-coded system performs better than the non-coded system by 1.6 and 8.4 percent on the first and third iteration, respectively. The improvement is more significant at higher TXVGA1 power of -13dB, where the Turbo-coded system has 23.2 and 45.7 percent better error rate than the non-coded system. On average, throughout the four different TXVGA1 power, the Turbo-coded

system provides error rate improvement of 12.4 and 33.8 percent on the first and third iteration, respectively.

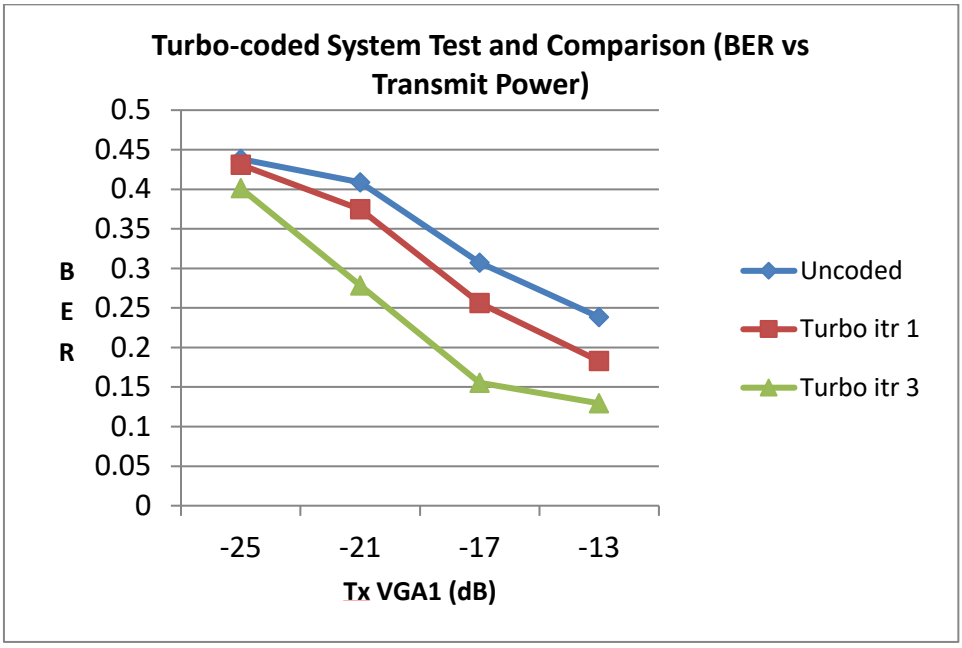


Figure 6.11 Comparison of Turbo-coded and non-coded Transceiver for Varying Transmit Power

Similarly, test result for the varying transmission range is presented in Figure 6.12, where the error rate of Turbo-coded system at the first and three iterations are compared against the non-coded system. For this test, the chunk size is fixed at 48 bits and TXVGA1 power of -13dB. It is evident from the graph that the Turbo-coded system starts to perform better on transmit range of two meters, improving the error rate by 13.5 and 31 percent on the first and third iteration, respectively. The Turbo-coded system of one decoding iteration performs worse compared to the non-coded system at longer transmit range of three and four meters. At the same range, the Turbo-coded system at the third iteration provides almost no improvement. This behavior is a common characteristic of the Turbo Code, which is also shown by the Turbo Code system

simulation discussed in Chapter Three, where the Turbo decoding provides equal or worse error rate performance at low SNR.

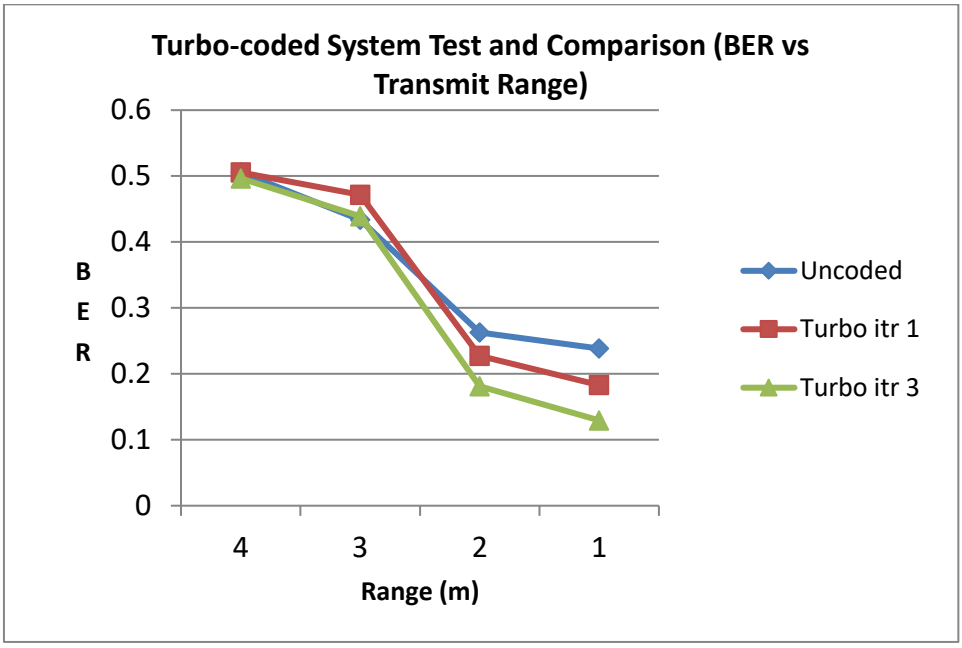


Figure 6.12 Comparison of Turbo-coded and non-coded Transceiver for Varying Transmit Range

It is observable from both Figure 6.11 and Figure 6.12 that the both the non-coded and Turbo-coded system are exhibiting poor error rate, with the third iteration Turbo-coded system achieving only around 10^{-1} BER. This BER value is much higher than the desired practical value of at least 10^{-6} (Li, Maunder, Al-Hashimi, & Hanzo, 2013). The poor BER performance is defined by a lot of factors which can be grouped into three domains, which are the digital computation accuracy, the analog RF design, and the testing environment.

In this work, the digital computation suffers from numerous degradation due to the use of various approximations which exists throughout the Turbo Code and BPSK modulator/demodulator implementation, such as the Max-

BCJR algorithm and receiver synchronization method. Moreover, the 32-bit fixed-point data type used also suffers from inaccuracy due to the more limited precision compared to floating point type of the same size. The analog RF design includes numerous factors such as carrier frequency, bandwidth, antenna design, receiver sensitivity, internal thermal noise. These factors are just couples of dozen factors affecting the link quality, which are yet to be considered in the scope of this work. Lastly, the testing was performed in a normal room with no means of measuring or controlling the channel condition, especially its SNR. These factors altogether contribute to the error rate exhibited by the wireless system.

6.2.4 Transceiver Data Rate and Power Usage Analysis

Based on the variable-sized transmission frame format, data rate of 2kbps, and Turbo decoding time of 0.73s, the radio air time as well as the actual data rate can be derived in Table 6.1 below.

Table 6.1 Transceiver Data Rate and Air Time

Chunk size (bits)	No of freq-pilot pairs	Total bits	Air time (s)	Air time percentage (Rx-decode)	Actual data rate (bps)
48	16	816	0.408	35.85%	224.9560633
32	24	832	0.416	36.30%	223.3856894
24	32	848	0.424	36.74%	221.8370884
16	48	880	0.44	37.61%	218.8034188

The number of frequency preamble pairs equals to the number of chunks, which is the 768 bits in codeword divided by the chunk size. The total bits in the frame hence is defined by 768 bits codeword, 16 bits detection and

timing preamble, and number of frequency preamble times two bits. With symbol rate of 2kbps, the radio air time for each configuration of chunk size can be derived as well. Assuming a continuous operation of receive-decode on the receiver end, the Lime LMS6002D is active for 35-37% of the operation time, which allows the device to be temporarily turned off while the RUMPS401 is processing the frame. If a periodical transmit-receive scheme is considered, the transmitter would have a similar air time to the receiver and in fact consumes less power as the RUMPS401 can be turned off as well after the frame is sent. The transceiver actual data rate is defined for the 256 original data bits transmission over a single receive-decode cycle.

Table 6.2 RUMPS401 Current Consumption Under Idle Loop

Cores active	Current consumption (mA)
All	32.1
IO Core	3.4
Normal Core 0	6.5
Normal Core 1	6.8
DSP Core	3.3
NoC	12.1

Table 6.2 lists the RUMPS401 current consumption measurement on 3.3V power supply with all cores running in an idle loop, and alternatively putting only one core doing idle loop while others are put to sleep. On the other hand, the Lime LMS6002D operates on 5V power supply and consumes 260mA when turned on. Table 6.3 presents the transceiver energy consumption in terms of transceiver active time, which is the variable air time plus the decoding time of 0.73s. The energy consumption is derived under the same repetitive receive-decode cycle as Table 6.1, in which none of the cores are put to sleep.

Table 6.3 RUMPS401-LMS6002D Transceiver Energy Consumption

Chunk size	Air time (s)	Active time (air + decoding) (s)	LMS6002D energy consumption (Watt-air time)	RUMPS401 energy consumption (Watt-active time)	Total energy consumption (Watt-active time)
48	0.408	1.138	0.530	0.120	0.651
32	0.416	1.146	0.541	0.121	0.662
24	0.424	1.154	0.551	0.122	0.673
16	0.44	1.17	0.572	0.124	0.696

Despite being active for the whole receive-decode cycle which is almost three times the air time, the RUMPS401 consumes much less power compared to the Lime LMS6002D at only one fifth of the later. A preferable countermeasure to reduce the whole transceiver energy consumption is by keeping the radio air time as low as possible. This can be achieved by increasing the baseband data rate with proper considerations. As discussed in Chapter Two, the use of higher degree of modulation such as QPSK or 8-PSK can greatly increase the data rate while keeping spectral efficiency, but in expense of the error rate. Another possible approach is to stick with BPSK and simply increase the data rate, e.g. doubling the data rate would reduce the radio air time and energy consumption by half.

In practice, the energy limited IoT devices (Bormann, Ersue, & Keranen, 2014) generally operates under two scenarios. The simplest method is to have the device normally turned off and turned on periodically for transmit-receive operation. A more complex approach set the devices in low-power mode while still maintaining a connection to the wireless network, either by some wake-up signal or other intelligent receive method, commonly implemented in higher communication layer. Assuming a periodical wireless operation of five seconds, for chunk size of 48 bits the LMS6002D and the RUMPS401 only needs to be

turned on for 8.16 and 22.76 percent of the time, respectively. In terms of continuous operation for long duration, larger wireless operation period will obviously lead to lower energy consumption as the radio is only turned on for smaller fraction of time. To lower energy consumption further, partial sleep can be integrated into the RUMPS401 decoding process.

6.3 Summary

This chapter has covered the integration of the Turbo encoder-decoder and BPSK modulator-demodulator as well as the interfacing to the desktop program into a complete functioning transmitter and receiver software running on the RUMPS401 MPSoC. The transmitter software accepts input of 256-bits data frame from the desktop program via the UART, Turbo-encodes the data and augments necessary synchronization preambles before sending it wirelessly. On the other hand, the receiver software monitors for any wireless transmission, receive the frame and performs necessary timing and frequency correction before passing it to the Turbo decoder. The decoder frame is then sent to the desktop program via the UART. The Python-based desktop program for both the transmitter and receiver was also described, along with the communication protocol between the programs and the RUMPS401. The complete set of functioning wireless module and the application processor was presented, upon which the testing was carried out.

Testing for the transceiver system was carried out by simple one-way communication setup in a LoS configuration. Unit of measurement is the receiver error rate of the forty 256-bit frames, each transmitted over separated

burst transmission. The forty frames are pre-randomized and stored by the transmitter as well as the receiver and used throughout all test scenarios. Test on non-coded system has verified the functionality and behavior of the BPSK modulator-demodulator part. The error rate diminishes as the transmission power increases and as the modules are communicating in shorter range. The test result also shows the system range limitation at four meters, which is affected by numerous factors that are not in the scope of this work. Effect of the number of frequency preambles in the frame format described in Chapter Five was also verified here, where more frequency preambles results in better correction accuracy as the cumulative error introduced by phase offset estimation inaccuracy is suppressed.

As shown by the test result, the Turbo-coded system works properly as well, providing significant improvement over the non-coded system at the third decoding iteration. At the best setup of this test, i.e. TXVGA1 power of -13dB and range of one meter, the Turbo-coded system provides 23.2 and 45.7 percent better error rate than the non-coded system. At longer transmission range of three meters and above, the Turbo-coded system provides equal or worse error rate than the non-codes system, which is similar to the behavior observed during the Turbo Code simulation presented in Chapter Three.

On the other hand, the wireless system is performing poorly in terms of BER, with the third iteration of Turbo Code reaching only 10^{-1} BER, which is far from the practically desirable BER of 10^{-6} (Li, Maunder, Al-Hashimi, & Hanzo, 2013). This poor performance is affected by computation inaccuracy of various algorithm approximations, the design of RF region which is beyond the

scope of this project, and the test environment which has not been measured and characterized. As mentioned in the fourth and fifth chapters, implementing specialized hardware block as processor's peripheral can increase the computation speed as well as the accuracy, which will serve as an interesting study for future work. Furthermore, more future work can be done around the design of RF region along with proper setup of test environment to realize a practical programmable wireless transceiver system with desirable error rate.

Based on the baseband data rate of 2kbps, the transceiver system has around 0.4s air time that slightly varies depending on the frame's chunk size. For a complete receive-decode cycle of around 1.1s, the Lime LMS6002D RF frontend need to be active only for around 35-37% of the total time, while the RUMPS401 is active the whole time. Depending on the application with periodical receive-decode over larger period, the Lime LMS6002D active time percentage will drop further. In the long run, the RUMPS401 energy consumption can be decreased further by integrating partial sleep into its decoding process.

Compared to the Lime LMS6002D, the RUMPS401 is active for almost three times longer whilst consuming only one fifth energy during the receive-decode cycle. As the programmable RF frontend is dominating the power consumption, reduction of the transceiver's energy consumption can be achieved by lowering the air time, either by using higher degree of modulation or simply increasing the raw data rate with proper consideration of its effect against error rate. This opens the development of low-power programmable SDR as another interesting area of research for future work.

CHAPTER 7

CONCLUSION AND RECOMMENDATIONS

As described in the beginning of this writeup, the ultimate goal of this work is to contribute to the development of wireless system that is suitable for Internet of Things (IoT) applications, mainly considering the ultra-low power operation and high programmability aspects of the wireless system. Low power performance is critical as IoT devices must last as long as possible on a given battery power source. This leads to the comparative study of narrowband and spread spectrum system as well as the use of Turbo Code in attempt to significantly reduce the receiver error rate, hence resulting in lower power consumption. The wireless module programmability is as important to allow multitude of devices to adapt to the rapid development of the IoT wireless protocol, by implementing the radio functionality in software rather than hardware. The digital processor on which the software runs must be programmable and with low power consumption but computationally powerful enough for the required radio operation. The RUMPS401 is chosen as it satisfies the first two requirements, while the computation performance aspect was achieved through proper optimization of the radio software.

This chapter highlights the major contributions of the project and presents the conclusions drawn from the results and analysis given in Chapter 4 and 6. Extending from this some recommendations are proposed as directions for further works.

7.1 Summary of Contributions

A significant amount of time and efforts in this work is equally spent into the study of ASIC design principle as well as the comprehension of wireless communication system with emphasis on the physical layer which includes modulation and Turbo Code. The baseband processor used in this work, the RUMPS401 is the product of UTAR VLSI research center resulted from the application of First-Time-Success ASIC design methodology. The design, tape-out, and verification of the four-core Multi-Processor System-on-Chip (MPSoC) is a joint-effort of the VLSI research center's members, in which this author is fully responsible for the whole chip physical design. The chip is designed with industrial-standard EDA tools and fully functional in the first fabrication attempt. This work has demonstrated the chip functionality and capability for a complex, real-world application as a baseband processor.

The wireless transceiver system built in this work has served as a proof of a fully functional wireless transmitter and receiver based on software-defined radio (SDR) implementation on a low-power and programmable MPSoC. The RUMPS401 is not equipped with any hardware accelerator that aids signal processing except a single-cycle multiplier hardware. Two main building elements of the wireless system, the Turbo Code and the BPSK modulator-demodulator were fully implemented in software, which is designed with the RUMPS401 quad ARM M0 cores and UTAR NoC architecture and resources in mind. Despite the software being developed and optimized specifically for the RUMPS401, the simplification and parallel processing method devised for the Turbo Code can be generally adapted to other platforms. Likewise, the

BPSK modulator-demodulator implementation in this work proposes an approach that is suitable for low-power platforms where computational and memory resources is scarce.

As a part of VLSI Research Center's project, this work has contributed to the research center long-term goals of buildings human design resources, platforms and tools in form of hardware and software, as well capabilities to design and explore MPSoCs in the wider IoT settings. In terms of human resources, individuals involved in this research work as well as the RUMPS401 development has nurtured capabilities in complete ASIC design methodology along with its use for IoT applications. The use of RTL platform from former works, and the application-level hardware and software designed in this work for the RUMPS401 has implied the research center eagerness towards long-term and continuous development cycle.

Specifically, here are some major contributions and benefits from the development and implementation of SDR on low-power platforms:

- The Turbo decoder optimization method that emphasizes around the reduction and simplification of complex mathematical operations can be adapted into other low-complexity processors with limited computational hardware resource. Most of the simplified operations consist of addition and subtraction, along with several multiplications. Fast multiplicator hardware can be commonly found in modern processors, even for the low-power variant.

- The Turbo decoder parallelism scheme devised for the RUMPS401 can be used as the reference for implementation on heterogeneous MPSoC platforms. The decoding algorithm is split into smaller tasks and assigned to the cores based on each core capabilities. Analysis on the use of sliding window for parallel processing as well as the memory requirements also provides a useful insight.
- Gardner-based timing correction on the BPSK demodulator uses a simple approach for readjustment of the sampling time by the step of the receiver oversampling points. This allows the correction to be done only by the digital processor without changing the RF receiver's Analog-to-Digital Converter (ADC) sampling time. Furthermore, the interpolation that is usually required to predict the value at correct sampling time can be avoided by using multiple pairs of preamble bits.
- The frequency correction method that estimates phase offset only on preambles and cumulatively correct the consequent data bits demands very simple calculations as well as relaxing the system from real-time calculation. The method trades off complexity for data rate and is suitable for low-power platforms.
- Design of the wireless system software is modular. The Turbo Code, the BPSK modulator/demodulator, the Lime LMS6002D interfacing, and the desktop program interface are implemented as separated modules. This allows reusability of those components in other SDR-based transceiver systems.

7.2 Conclusion

The cycle-accurate simulation performed on the Turbo decoder software during the optimization process, as well as the testing of the complete transceiver system yields several conclusions as follows:

- Software implementation of Turbo decoding in low-power MPSoC could achieve significant speed improvement through simplification of calculations and parallel tasks distribution. The simplification focuses on reducing the number of mathematical operations that is expensive to carry such as multiplication, division, exponential, logarithm, etc. In the case of the RUMPS401, the multiplication is not a problem due to the multiplicator hardware available to the DSP Core. Parallel processing of the Turbo decoding can be achieved by splitting the algorithm into smaller tasks which are performed in parallel on the same or different chunks of the frame.
- For heterogeneous MPSoC such as the RUMPS401, it is better to assign different tasks to each core rather than having them performing the same type of task over different data sets. Data-centric task assignment requires each core to perform multiple type of tasks, which results in complex communication among the cores as they must identify which task's data is being communicated now. This complex communication introduces data overhead and additional control system, which, as shown by the simulation, produces slower decoding performance.
- The BPSK modulator/demodulator implementation adheres to the same concept observed during the Turbo Code optimization process, which

are the simplification of complex calculations and task parallelization by over different processing elements according to their resources. This demonstrates that the same concept can be adopted in any software implementation where hardware resources are limited for the given application.

- Pulse shaping function of the BPSK modulator, along with the receiver synchronization performed on the BPSK demodulator is proven to work through a real wireless transmit and receive testing. Despite the compromised and simple calculations on the synchronization part, the wireless system demonstrates proper behavior in which the receiver error rate diminishes as the transmit power increase and the wireless range gets closer. Likewise, the trade-off in the proposed frequency correction scheme was also verified, where the correction accuracy increases as preambles insertion increases.
- Compared to the ideal algorithm, the Turbo decoding implementation in this work suffers from many accuracy degradations. The use of the Max-BCJR algorithm, the sliding window method, the approximation of logarithmic function, and the use of fixed-point arithmetic contribute to the loss of decoding accuracy. The tests result shows that the Turbo-coded system provides quite significant improvement over the non-coded system, which is 23.2 and 45.7 percent at the first and third iteration.
- The transceiver system exhibits BER of 10^{-1} at the third iteration of Turbo Code, which is poor compared to the practically desired BER of 10^{-6}. This stems from the accuracy degradations in the Turbo Code and

receiver synchronization, the design of RF region which is beyond this work's scope, and the test environment.

- The transceiver system's energy consumption is dominated by Lime LMS6002D despite being active for only 35-37% of a receive-decode period. Lower energy consumption over longer period can be achieved by increasing the periodical transmit-receive duration or reducing the air time with higher baseband data rate. Further power saving can be done by integrating partial sleep into the RUMPS401 decoder software.

7.3 Recommendations

Based on the conclusions, below are some recommendations for future works:

- A significant speed improvement on the whole SDR software which includes Turbo Code and BPSK modulator/demodulator can be achieved by enhancing the RUMPS401 with additional divisor or logarithmic hardware accelerator. As shown during the Turbo decoder optimization process, most of the performance bottleneck lies in the logarithmic operation and the arbitrary division. This allows fast performance improvement with very little change on the RUMPS401 hardware and allows the reuse of most of the current software. Additionally, hardware accelerator for certain operation commonly provides better accuracy than the approximated version.

- Implementation of the transceiver system on other MPSoC platforms with similar characteristic and resources can be performed and compared against the current work.
- More study can be performed on the design of the radio specifications such as the choice of antenna, carrier frequency, modulation technique to provide longer transmission range and better robustness against interference. Study of the system error rate on properly set and measured environment can be carried out as well, along with the implementation of the countermeasures.
- The whole transceiver system can be further improved by incorporating power management system. The RUMPS401 is capable of individual core sleep-wake, while the Lime LMS6002D can work straight after power on by simply configuring the registers. The design of a periodical transmit-receive or wake-on-carrier-detect protocol would further complement the pursuit of a power saving transceiver module which is only powered up during use.
- The complete transceiver module built in this work stresses on the RUMPS401 usage as the baseband processor for handling wireless communication. The module can be directly implemented as a complete IoT node by adding sensors and actuators with the RUMPS401 as the microcontroller unit. This eliminates the need of additional microcontroller unit.

REFERENCES

About GNU Radio. (n.d.). Retrieved February 2016, from GNU Radio: http://gnuradio.org

Ahmed, A., Awais, M., Rehman, A. u., Maurizio, M., & Masera, G. (2011). A high throughput turbo decoder VLSI architecture for 3GPP LTE standard. *IEEE 14th International Multitopic Conference*, (pp. 340-346). Karachi.

ARM. (2009). *Cortex M0 Technical Reference Manual.* Retrieved June 2016, from ARM Information Center: http://infocenter.arm.com/help/topic/com.arm.doc.ddi0432c/DDI0432C_cortex_m0_r0p0_trm.pdf

ARM. (n.d.). *GNU ARM Embedded Toolchain*. Retrieved from ARM Developer: https://developer.arm.com/open-source/gnu-toolchain/gnu-rm

Atmel. (2007). AT89C51CC03 UART Bootloader. Atmel.

Bahl, L., Cocke, J., Jeinek, F., & Raviv, J. (1974). Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Trans. Inform. Theory,vol. IT-20,* 248-287.

Becker, T., Luk, W., & Cheung, P. (2009). Parametric Design for Reconfigurable Software-Defined Radio. *5th International Workshop ARC (Karlsruhe)*, 15-26.

Berrou, C., Glavieux, A., & Thitimajshima, P. (1993). Near Shannon Limit Error-Correcting Coding and Decoding: Turbo-Codes (1). *International Communications Conference*, (pp. 1064-1070). Geneva, Switzerland.

Bormann, C., Ersue, M., & Keranen, A. (2014, May). *Terminology for Constrained-Node Networks.* Retrieved from Internet Engineering Task Force (IETF): https://tools.ietf.org/html/rfc7228

Boschen, D. (2016, June). *Gardner Timing Recovery.* Retrieved from DSP Stack Exchange: https://dsp.stackexchange.com/questions/31528/gardner-timing-recovery-for-repeated-symbols

Boschen, D. (2016, 6). *How to Correct Phase Offset for QPSK IQ Data.* Retrieved from DSP Stack Exchange: https://dsp.stackexchange.com/questions/31497/how-to-correct-the-phase-offset-for-qpsk-i-q-data/31506

Burdett, A. (Spring 2015). Ultra-Low-Power Wireless Systems: Energy-Efficient Radios for the Internet of Things. *IEEE Solid-State Circuits Magazine, vol. 7, no. 2,* 18-28.

Clermidy, F., Lemaire, R., Popon, X., Kténas, D., & Thonnart, Y. (2009). An Open and Reconfigurable Platform for 4G Telecommunication: Concepts and Applicatio. *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference,* 449-456.

Condo, C., Martina, M., & Masera, G. (2012). A Network-on-Chip-based turbo/LDPC decoder architecture. *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE),* (pp. 1525-1530). Dresden.

Crozier, S., & Guinand, P. (2001). High-Performance Low-Memory Interleaver Banks for Turbo-Codes. *Vehicular Technology Conference 54th*, (pp. 2394-2398). Atlantic City.

Dam, T. v., & Langendoen, K. (2003). An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. *The First ACM Conference on Embedded Networked Sensor Systems*, 171-180.

Gardner, F. (1986). A BPSK/QPSK Timing-Error Detector for Sampled Receivers. *IEEE Transactions on Communications Volume 34 Issue 5*, 423-429.

Garello, R., Chiaraluce, F., Pierleoni, P., Scaloni, M., & Benedetto, S. (2001). On error floor and free distance of turbo codes. *IEEE International Conference on Communications, ICC 2001*, (pp. 45-49). Helsinki.

Ghazi, A., Boutellier, J., Hannuksela, J., Silvén, O., & Janhunen, J. (2013). Low-complexity SDR implementation of IEEE 802.15.4 (ZigBee) baseband transceiver on application specific processor. *Proc. SDR-WInnComm*.

Hartono, D. (2014). *The Design and IMplementation of a Scalable Multi-Processor System-on-Chip using Network Communication for Parallel Coarse-Grain Data Procesing.* Kampar: Universiti Tunku Abdul Rahman.

Haykin, S. (2001). *Communication Systems 4th Edition.* Wiley.

Hu, J., Ma, Z., & Sun, C. (2011). Energy-efficient MAC Protocol Designed for Wireless Sensor Network for IoT. *Computational Intelligence and Security (CIS) 7th International Conference*, 721-725.

Huang, L., Lluo, Y., Wang, H., Yyang, F., Shi, Z., & Gu, D. (2011). A high speed turbo decoder implementation for CPU-based SDR system. *IET International Conference on Communication Technology and Application (ICCTA 2011)*, (pp. 19-23). Beijing.

Ifeachor, E., & Jervis, B. (2002). *Digital Signal Processing: A Practical Approach, 2nd Edition.* Pearson.

Instruments, N. (2014, 11 05). *Pulse-Shape Filtering in Communications Systems.* Retrieved from National Instruments: http://www.ni.com/white-paper/3876/en/

Instruments, T. (2012, November). *Multicore DSP+ARM KeyStone II System-on-Chip (SoC).* Retrieved April 2015, from Texas Instruments: http://www.ti.com/product/66AK2H12/datasheet

Jalier, C., Lattard, D., Jerraya, A., Sassatelli, G., Benoit, P., & Torres, L. (2010). Heterogeneous vs homogeneous MPSoC approaches for a Mobile LTE modem. *Design, Automation & Test in Europe Conference & Exhibition (DATE) 2010*, 184-189.

Jondral, F., Wiesle, A., & Machauer, R. (2000). A software defined radio structure for 2nd and 3rd generation mobile communications standards. *IEEE Sixth International Symposium on Spread Spectrum Techniques and Applications*.

Jr, C. R., Sethares, W. A., & Klein, A. G. (2011). *Software Receiver Design: Build your Own Digital Communication System in Five Easy Steps.* Cambridge University Press.

Kuo, P., & Kung, H. (2014). Subcarrier index coordinate expression (SICE): An ultra-low-power OFDM-compatible wireless communications scheme tailored for internet of things. *2014 International Conference on the Internet of Things (IOT)*, 97-102.

Labs, S. (n.d.). *Si5356-EVB Clock Generator User Guide.* Retrieved from Silicon Labs User Guide: https://www.silabs.com/documents/public/user-guides/Si5356EVB.pdf

Lassen, T. (2014). *Long-range RF communication: Why narrowband is the de facto standard.* Texas Instrument.

Lattard, D., Beigne, E., Clermidy, F., Durand, Y., Lemaire, R., Vivet, P., & Berens, F. (2008). A Reconfigurable Baseband Platform Based on an Asynchronous Network-on-Chip. *IEEE Journal of Solid-State Circuits*, 223-235.

Li, A., Xiang, L., Chen, T., Maunder, R. G., Al-Hashimi, B. M., & Hanzo, L. (2016). VLSI Implementation of Fully Parallel LTE Turbo Decoders. *IEEE Access Vol. 4*, 323-346.

Li, L., Maunder, R. G., Al-Hashimi, B. M., & Hanzo, L. (2013). A Low-complexity Turbo Decoder Architecture for Energy-Efficient Wireless Sensor Networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14-22.

Lim, Z. (2015). *Design and Application of Network-on-Chip Virtual Prototyping Platform.* Kampar: Universiti Tunku Abdul Rahman.

Liu, C., Bie, Z., Chen, C., & Jiao, X. (2013). A parallel LTE Turbo decoder on GPU. *15th IEEE International Conference on Communication Technology*, (pp. 609-614). Guilin.

Lokananta, F. (2015). *Netwok-on-Chip Communication Architecture Design, Analysis, Optimization and Evaluation in a Multi-Processor System-on-Chip.* Kampar: Universiti Tunku Abdul Rahman.

Ma, S., Marojevic, V., Balister, P., & Reed, J. H. (2014). Porting GNU Radio to Multicore DSP+ARM System-on-Chip – A Purely Open-Source Approach. *8th Karlsruhe Workshop on Software Radios*, (pp. 21-28). Karlsruhe.

Marandian, M., Fridman, J., Zvonar, Z., & Salehi, M. (2001). Performance analysis of turbo decoder for 3GPP standard using the sliding window algorithm. *12th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications. PIMRC 2001. Proceedings (Cat. No.01TH8598)*, (pp. E-127-E-131). San Diego, CA.

Mathworks. (n.d.). *Matlab Overview*. Retrieved from Mathworks.com: https://www.mathworks.com/products/matlab.html

Matsumoto, M., & Nishimura, T. (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*

(TOMACS) - Special issue on uniform random number generation Vol.8 , 3-30.

Microsystem, L. (2012, 12 3). *Lime Microsystem LMS6002D Datasheet.* Retrieved from Lime Microsystem: http://www.limemicro.com/download/LMS6002Dr2-DataSheet-1.2r0.pdf

Minden, G. J., Evans, J. B., Searl, L., DePardo, D., Petty, V. R., Rajbanshi, R., . . . Agah, A. (2007). KUAR: A Flexible Software-Defined Radio Development Platform. *IEEE New Frontiers in Dynamic Spectrum Access Networks*, 428-439.

Mulally, D., & Lefevre, D. (1991). A Comparison of Digital Modulation Methods for Small Satellite Data Links. *5th Annual AIAA/USU Conference on Small Satellites.*

MyriadRF. (2013, 05). *MyriadRF-1 Development Kit User Manual.* Retrieved from Lime Microsystem: https://www.limemicro.com/download/Myriad-RF%20Development%20Kit_1.0r4.pdf

Okumura, T., Ohmori, E., Kawano, T., & Fukuda, K. (1968). Field strength and its variability in VHF and UHF land mobile service. *Review Electronic Communication Lab 16 No 9-10*, 825-873.

Owen, M. (n.d.). *Calculate exp and log Without Multiplications.* Retrieved from Quinaplus: https://www.quinapalus.com/efunc.html

Pickholtz, R. L., Schilling, D. L., & Milstein, L. B. (1982). Theory of Spread Spectrum Communications-A Tutorial. *IEEE Trans. Commun. Vol. 30 No 5*, 855-884.

Proakis, J. G., & Salehi, M. (2001). *Communications System Engineering, 2nd Edition.* Pearson.

Reed, M. C., & Asenstorfer, J. (1997). A Novel Variance Estimator for Turbo-Code Decoding. *International Conference on Telecommunications*, (pp. 173-178). Melbourne.

Robertson, P., Hoeher, P., & Villebrun, E. (1997). Optimal and sub-optimal maximum a posteriori algorithm suitable for turbo decoding. *Euro. Trans. Telecommun., vol. 8, no. 2*, 119-125.

Roth, Y., Dore, J., Ros, L., & Berg, V. (2015). Turbo-FSK: A new uplink scheme for low power wide area networks. *Signal Processing Advances in Wireless Communications (SPAWC) IEEE 16th International Workshop*, (pp. 81-85).

Sadjadpour, H. (2000). Maximum a posteriori decoding algorithms for turbo codes. *Proc. SPIE Digit. Wireless Commun. II, vol. 4045*, (pp. 73-83).

Sklar, B. (2001). *Digital Communications: Fundamentals and Applications 2nd Edition.* Prentice Hall.

Sklar, B. (2008). *Maximum a posteriori decoding of turbo codes.* Retrieved from InformIT: http://www.informit.com/content/images/art_sklar4_map/elementLinks/art_sklar4_map.pdf

Standardization, I. O. (2006, 4 4). *Programming languages (C) Extensions to support embedded processors.* Retrieved from Open Standard Org: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1169.pdf

Sublime. (n.d.). *Sublime Text 3*. Retrieved from Sublime Text: https://www.sublimetext.com/3

Valenti, M., & Sun, J. (2001). The UMTS Turbo Code and an Efficient Decoder Implementation Suitable for Software-Defined Radios. *International Journal of WIreless Information Networks, Vol. 8, No. 4*, 203-216.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymtotically optimum decoding algorithm. *IEEE Trans. Inform. Theory vol. IT-13*, 260-269.

Walfisch, J., & Bertoni, H. L. (1988). A Theoretical Model of UHF Propagation in Urban Environment. *IEEE Transactions, Antennas & Propagation*, 1788-1796.

Wang, C., Yang, Y.H., Y., & R.D. (2009). On the Performance of DS Spread-Spectrum Systems in AWGN Channel. *IEEE Communications and Mobile Computing '09*, (pp. 311-315).

APPENDIX A

RUMPS401 SOFTWARE-BASED TURBO DECODER PSEUDOCODE

```
/*
  IO Core - RUMPS401 Turbo Decoder Pseudocode
  V1.4
*/

constant frameSize = 256
constant windowSize = 32
constant maxIteration = variableValue

while decoder is on
{
  get systematicBit[frameSize] from RX_buffer
  get parityBit1[frameSize] from RX_buffer
  get parityBit2[frameSize] from RX_buffer

  init LLRarray[frameSize] = 0
  init pktType = null // track multiple type of transfer
  init bitSent = 0 // tracks no of bit-parities sent to dspcore
  init bitDecoded = 0 // tracks no of decoded bits
  init bitSendHold = false // flag, control bit transfer to dspcore
  init decoderNo = first // identify 1st or 2nd decoder
  init halfIterationCtr = 0 // track no of decoding iteration (half)
  init calcLLRorLe = Le // flag, control calculation of LLR/Le

  noiseVar = co-calculate received frame's noise variance with dspCore
  noiseVar = 1 / noiseVar

  loop until last bit and last iteration reached
  {
    if thereIs NocPkt from dspCore && pktType == null
      pktType = getNocPkt from dspCore

    if thereIs NocPkt from normalCore0
    {
      if decoderNo == first
        index = bitDecoded
      else if decoderNo == second
        index = drpInterleave(bitDecoded)

      LLRarray[index] = getNocPkt from normalCore0
      increment bitDecoded
      nocSend(ACK) to normalCore0

      if bitSent < frameSize &&
         bitSendHold == trueWaitDecoding &&
         bitDecoded is multiple of windowSize &&
         numberOfWindowSent - numberOfWindowDecoded < 2
      {
        bitSendHold = false
      }
```

```
  if bitDecoded = frameSize
  {
    reset bitSent = bitDecoded = 0
    bitSendHold = false

    toggle decoderNo = first to second or second to first

    increment halfIterationCtr

    if halfIterationCtr == one iteration before the last
      calcLLRorLe = LLR

    if halfIterationCtr == the last iteration
    {
      clear RX_buffer
      re-allow data reception // blocked after a frame received
      reset halfIterationCtr = 0
      calcLLRorLe = Le

      frame fully decoded, break loop
    }
  }

}

if pktType == PktHeader_bitACK && thereIs NocPkt from dspCore
{
  pktType = null
  read ACK from dspCore
  bitSendHold = false

  if bitSent == frameSize ||
    (bitSent is multiple of windowSize &&
     numberOfWindowSent - numberOfWindowDecoded >= 2)
  {
     bitSendHold = trueWaitDecoding
  }

}

if bitSent < frameSize && bitSendHold == false
{
  if bitSent ==0
  {
    nocSend(PktHeader_channelControlInfo) to dspCore
    nocSend(calcLLRorLe) to dspCore
    nocSend(noiseVar) to dspCore
    wait for ACK from dspCore before starting the decoding
  }

  if decoderNo == first
  {
    index = bitSent
    parity = parityBit1[bitSent]
  }
  else if decoderNo == second
  {
    index = drpInterleave(bitSent)
```

```
        parity = parityBit2[bitSent]
      }

      nocSend(PktHeader_Bits) to dspCore
      nocSend(systematicBit[index]) to dspCore
      nocSend(parity) to dspCore

      if halfIterationCtr == 0
      {
        nocSend(logn(0.5)) to dspCore
        nocSend(logn(0.5)) to dspCore
      }
      else
      {
        nocSend(logn(LLR[index])) to dspCore
        nocSend(logn(1-LLR[index])) to dspCore
      }

      increment bitSent
      bitSendHold = trueWaitACK
    }
  }
}
```

```
/*
  Normal Core 0 - RUMPS401 Turbo Decoder Pseudocode
  V1.4
*/

constant frameSize = 256
constant windowSize = 32

init deltaBuffer[windowSize]
init alphaBuffer[windowSize]
init betaBuffer[windowSize]

init bitDeltaRecvd = 0 // count number of complete d metric received

while decoder is on
{
  if bitDeltaRecvd == 0
    init alphaBuffer[0] according to trellis
  else
    init alphaBuffer[0] from last window's iteration

  for i=1 to windowSize
  {
    wait for noc transfer from dspCore
    deltaBuffer[i] = getNocPkt from dspCore

    increment bitDeltaRecvd
    bitDeltaRecvd MOD frameSize // keep number in range of frameSize

    nocSend(ACK) to dspCore
  }

  calculate alpha for a window
  calculate beta for a window

  nocSend(alphaBuffer) to dspCore
  nocSend(betaBuffer) to dspCore
}
```

```
/*
  Normal Core 1 - RUMPS401 Turbo Decoder Pseudocode
  V1.4
*/

constant frameSize = 256
constant windowSize = 32

init calcLLRorLe = unknown
init llrConverted = 0

while decoder is on
{
  if calcLLRorLe == unknown && thereIs NocPkt from dspCore
  {
    calcLLRorLe = getNocPkt from dspCore
    nocSend(ACK) to dspCore
  }

  if calcLLRorLe != unknown && thereIsNocPkt from dspCore
  {
    LLR = getNocPkt from dspCore

    if calcLLRorLe == Le
      convert LLR from ratio to probability value

    if llrConverted+1 is multiple of 8
      nocSend(ACK) to dspCore

    nocSend(LLR) to ioCore
    wait for ACK from ioCore

    increment llrConverted

    if llrConverted == frameSize
    {
      reset calcLLRorLe = unknown
      reset llrConverted = 0
    }
  }
}
```

```
/*
  DSP Core - RUMPS401 Turbo Decoder Pseudocode
  V1.4
*/

constant frameSize = 256
constant windowSize = 32

init pktType = null
init bitRecvd = 0 // tracks number of bit received from ioCore
init bitDeltaSent = 0 // tracks number of bit sent to NC0 for delta
calc
init deltaSendHold = false // flag, control transfer to NC0

init calcLLRorLe = unknown
init llrSent = 0
init llrSendHold = false
init noise_var = 0

init deltaBuffer[2*windowSize]
init alphaBuffer[windowSize]
init betaBuffer[windowSize]

init alphaRecvd = false
init betaRecvd = false

while decoder is on
{
  co-calculate received frame's noise variance with ioCore

  loop until last bit and last iteration reached
  {
    if pktType == null && thereIs NocPkt from ioCore
      pktType = getNocPkt from ioCore

    if pktType == PktHeader_channelControlInfo &&
       thereIs NocPkt from ioCore &&
       number of NocPkt from ioCore >= 2
    {
      pktType = null
      calcLLRorLe = getNocPkt from ioCore
      noise_var = getNocPkt from ioCore
      nocSend(ACK) to ioCore
    }

    if pktType == PktHeader_Bits &&
       thereIs NocPkt from ioCore &&
       number of NocPkt from ioCore >= 4
    {
      temp_systematicBit = getNocPkt from ioCore
      temp_parityBit = getNocPkt from ioCore
      temp_apriori1 = getNocPkt from ioCore
      temp_apriori0 = getNocPkt from ioCore

      calculate_delta with above's parameter
      save result to deltaBuffer[bitRecvd]
      increment bitRecvd
```

```
  nocSend(PktHeader_bitACK) to ioCore
  nocSend(ACK) to ioCore

  pktType = null
}

if bitRecvd > bitDeltaSent &&
   deltaSendHold == false &&
   bitRecvd is multiple of windowSize
{
  for i=1 to windowSize
  {
    nocSend(deltaForSingleBit) to normalCore0
    increment bitDeltaSent
    wait for ACK from normalCore0
  }

  deltaSendHold == true
}

if thereIs NocPkt from normalCore0 &&
   alphaRecvd == false && betaRecvd == false &&
   number of NocPkt from normalCore0 >= 8
{
  alphaBuffer = getNocPkt from normalCore0
  betaBuffer = getNocPkt from normalCore0
}

if llrSendHold == true && thereIs NocPkt from normalCore1
{
  read ACK from normalCore1
  llrSendHold = false
}

if alphaRecvd == true && betaRecvd == true && llrSendHold == false
{
  calculate LLR for bit index [llrSent]

  if llrSent == 0
  {
    nocSend(calcLLRorLe) to normalCore1
    wait for ACk from normalCore1
  }

  nocSend(LLR) to normalCore1
  increment llrSent

  if llrSent is multiple of 8
    llrSendHold = true

  if bitDeltaSent is multiple of windowSize &&
     numberOfDeltaWindowSent - numberOfLLRWindowSent < 2
  {
    deltaSendHold = false
  }

  if llrSent is multiple of windowSize
    reset alphaRecvd = betaRecvd = false
```

```
      if llrSent == frameSize
      {
        bitRecvd = bitDeltaSent = llrSent = 0

        if calcLLRorLe == LLR
        {
          wait ACK for last bit's LLR from normalCore1
          read ACK from normalCore1

          frame fully decoded, break loop
        }
      }
    }

  }
}
```