A Hardware-Defined Approach to Software-Defined Radios: Improving Performance Without Trading In Flexibility

by

Omid Aryan

S.B., Massachusetts Institute of Technology (2012)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2013

Certified by ....
Prof. Hari Balakrishnan
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

A Hardware-Defined Approach to Software-Defined Radios: Improving Performance Without Trading In Flexibility

by

Omid Aryan

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The thesis presents an implementation of a general DSP framework on the Texas Instruments OMAP-L138 processor. Today's software-defined radios suffer from fundamental drawbacks that inhibit their use in practical settings. These drawbacks include their large sizes, their dependence on a PC for digital signal processing operations, and their inability to process signals in real-time. Furthermore, FPGA-based implementations that achieve higher performances lack the flexibility that software implementations provide. The present implementation endeavors to overcome these issues by utilizing a processor that is low-power, small in size, and that provides a library of assembly-level optimized functions in order to achieve much faster performance with a software implementation. The evaluations show substantial improvements in performance when the DSP framework is implemented with the OMAP-L138 processor compared to that achieved with other software implemented radios.

Thesis Supervisor: Prof. Hari Balakrishnan
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank Jonathan Perry for encouraging me to pursue this idea and for all his support and championship along the way. I would also like to thank Prof. Hari Balakrishnan for giving me the pleasure of working with his group, for all his great comments and insightful suggestions on this thesis, and for all his contributions to my education at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Overview

1.1 Introduction

In traditional wireless devices, the communication protocols are hard-wired into the circuitry. In other words, the ASICs used are customized and programmed for the particular operation of these protocols, and they cannot be reconfigured or updated. For example, a typical cell phone consists of several different chips that allow it to operate with different types of radio communications: cellular, WiFi, GPS, and Bluetooth. Moreover, the cell phone can only operate with a specific protocol for each of these communications, and the protocol cannot be upgraded to a newer version after the phone has been designed. For instance, a cell phone designed for the 3G cellular communications standard would be unable to operate or even updated to operate with a newer standard such as LTE.

Nonetheless, the concept of a "software defined radio" (SDR) is bound to overcome these limitations and to provide much more flexibility in designing wireless devices. Rapid advancements in the capabilities and speed of digital computation has made it possible to implement in software many components of a radio (such as filter banks, modulators, and mixers) that are usually implemented in hardware. This has led to the creation of a software defined platform that allows engineers to implement protocols spanning the entire spectrum and enabling them to easily modify and enhance their protocols by means of programming in software rather than having to change the circuitry in hardware.

The structure of a radio can be divided into two general parts: the digital signal processor (DSP) and the analog front-end (AFE). In an ideal SDR, the analog front-end consists only of an antenna that is connected via digital-to-analog/analog-to-digital converters (DAC/ADC) to the digital signal processor, where the DSP is controlled entirely by software. This is shown in Figure 1-1. Of course, practical limitations hinder the design of such a structure. For example, for protocols that operate in the gigahertz frequency range, it is very difficult and costly to design digital-to-analog/analog-to-digital converters that can operate on such high-frequency signals.

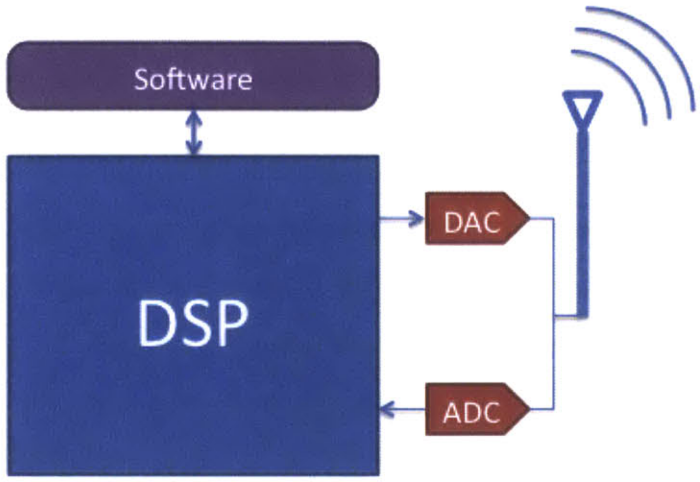

Figure 1-1: An ideal software defined radio.

Furthermore, handling the entire digital signal processing in software and in real-time can be quite challenging. Today's software defined radios implement a good portion of the digital signal processing in FPGA-like environments (which can only be modified through firmware) and the rest of the processing, which is done in software, occurs within a PC on prerecorded data. Not only are these systems non-real-time, but they are also entirely dependent on a PC for their software and are hence very power hungry and non-portable.

In this project I aim to overcome these limitations. Specifically, I present a general and basic software DSP implementation on the Texas Instruments OMAP-L138 processor. I show how one can improve the performance of the digital signal processing in an SDR by utilizing the processor's library of assembly-level optimized DSP routines, which offers the user a hard-wired hardware performance at the flexibility of a software implementation. The hardware will be further explained in Section 1.4.

1.2 Motivation

The concept of a *wireless software-defined network (WSDN)* has gained quite some recognition in the past few years [4]. In essence, the idea is to apply the principles behind software-defined networks [10, 11] to wireless systems by creating abstractions and modularity and by defining a wireless control plane and a wireless data plane. As opposed to regular (wired) software-defined radios, where the hardware at the switches need not be specialized, a WSDN node would require a software-defined radio in order to allow for a software implementation of the wireless protocols. Hence, a successful implementation of a WSDN depends heavily on high-performance software-defined radios that are also portable.

The goal of this project is to provide an implementation of a digital signal processing framework for an SDR to be used as part of a software defined wireless network that is currently under development at the Wireless Center at MIT. The long-term vision for this network is to establish a platform that allows researchers to experiment their wireless protocols with real users and to also pave the way for research in a software defined approach to wireless networks.

The envisioned network is composed of static base stations, which connect to the Internet through a backbone Ethernet, and portable SDRs, which users carry with them. The portable SDRs consist of a WiFi module that enables it to connect to the user's wireless devices (e.g., smart phone, laptop, etc.), as well as the DSP and analog front-end that provide it with a software defined platform to communicate with the base stations. The base stations are also envisioned to be software defined, and their implementation and functionality fall outside the scope of this thesis. The network is illustrated in Figure 1-2.

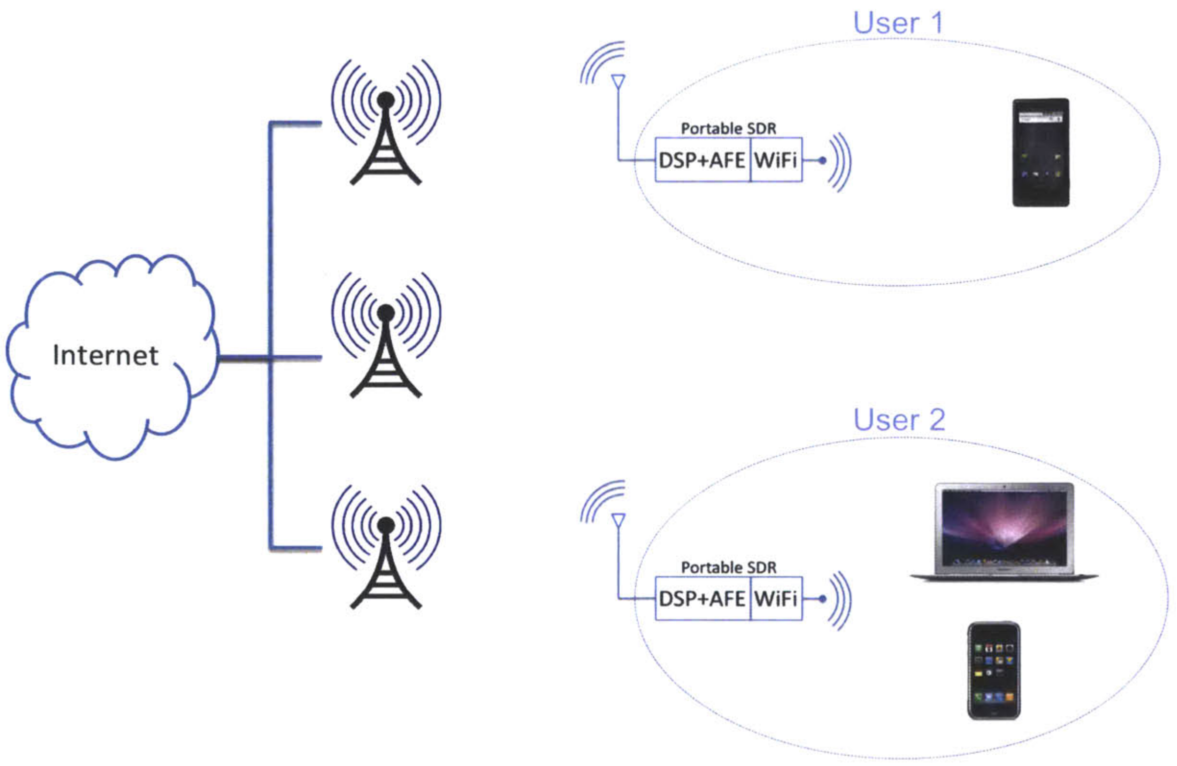

Figure 1-2: Long-term vision for portable SDR.

1.3 Related Work

Although software-defined radios have yet to find their way into today's wireless infrastructure and everyday consumer products, many vendors have produced them for both research and experimentation purposes [6, 7, 9, 10, 14, 17, 21, 22]. Each of these products offer different tradeoffs between software flexibility and hardware performance. For example, while platforms such as GNURadio [10] and Vanu [21] allow for total flexibility in software at the cost of performance, other platforms such as WARP [22] and Airblue [14] offer higher performances at the cost of an FPGA implementation.

The Universal Software Radio Peripheral (USRP) [6], designed by Ettus Research, is a common software-defined radio used in academia. Despite its wide usage in research settings, the USRP fails in many aspects to offer a practical implementation of a software-defined radio. First, the device itself occupies quite a large space ($22cm \times 16cm \times 5cm$). In addition, it requires a PC to handle the digital signal processing that is done in software.

Perhaps the most crucial aspect that makes the USRP unattractive even to the research community is the fact that it is unfeasible to carry out the required DSP operations of the system in real-time. As shown in the setup of Figure 1-3, once the USRP downconverts the wireless signal to complex baseband samples, the samples must travel through the Ethernet or USB cable that is connected to the PC, which by itself adds latency to the system. Furthermore, once the samples reach the PC, the digital signal processing of the samples occurs via a high-level language with which the protocol was implemented. Clearly, the processing time required for a program written in a language like C is substantially greater than that for a hard-wired system, and it is not fast enough to handle the DSP operations required by a real-time application.

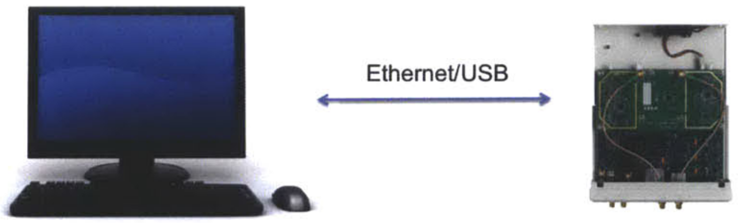

Figure 1-3: USRP Setup. For the receiver chain, the USRP downconverts the wireless signal to complex baseband samples and sends them to the PC for processing via an Ethernet or USB cable. The reverse procedure occurs for the transmit chain.

The SORA [17] platform is yet another type of software-defined radio commonly used in academic settings that mitigates the latency issue of an SDR architecture by applying parallel programming techniques on multicores. Despite the performance gains that these techniques offer, they fail to provide a convenient implementation for the user due to all the required software partitioning on multiple CPU cores. This platform also suffers from needing a PC to handle the DSP computations.

Furthermore, a number of other platforms have been proposed that delegate all or part of the signal processing and computations to an FPGA, making the platform less dependable on a PC. Examples of these platforms are WARP [22], where the

PHY layer is delegated to an FPGA while the MAC layer is done in software, as well as Airblue [14], were all the implementation is done on an FPGA using a high-level hardware description language (Bluespec [5]). Even though such platforms offer a much higher degree of performance compared to SDRs such as the USRP, they do not offer the same flexibility and convenience of programming with a language such as C.

In this thesis, I present an implementation that is capable of achieving the same degree of flexibility achieved with platforms such as GNURadio but without having to trade off much performance. The implementation is meant to be a very general and basic DSP framework that would give a sense of how a complete implementation would compare to other implementations of SDRs.

1.4 Hardware

The goal of this project is to implement a basic version of the digital signal processing framework for the SDR introduced in Section 1.2 on the Texas Instruments OMAP-L138 processor. The OMAP-L138 is a dual-core DSP+ARM processor designed for low-power applications, making it a suitable choice for a portable SDR with a limited power source. The work presented in this thesis is mainly focused on the DSP side of the processor, while the general-purpose ARM core of the device will become useful in future work when it becomes necessary to interface the SDR with the WiFi module. Some other features of the processor include the following: clock rate of 375 MHz, 326 kB of internal memory (RAM/cache), 256 MB of external memory (SDRAM), and 64 32-bit general-purpose registers.

Arguably the most important advantage of the OMAP's DSP processor is its collection of general-purpose signal-processing routines that are callable from C programs. These functions are assembly-optimized and are included in the processor's DSP Function Library (DSPLIB). By virtue of these functions, it is possible to achieve low-level hardware performance at the flexibility of programming in software, making the OMAP-L138 an ideal processor for use in a software-defined radio.

To carry out the implementations, the Logic PD Zoom OMAP-L138 eXperimenter kit was used, which is a development platform for the OMAP-L138 processor that makes available a large number of the processor's interfaces. The kit was generously donated by Texas Instruments and is shown in Figure 1-4. TI's Code Composer Studio v4.2.1 was used to develop the implementation (software written in C) and to program the processor.

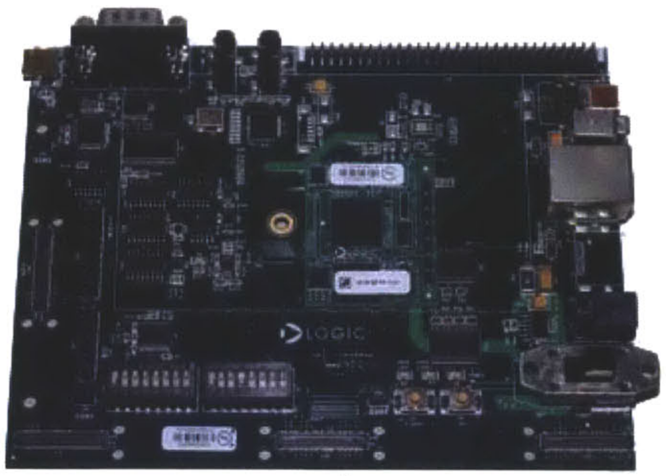

Figure 1-4: Logic PD Zoom OMAP-L138 eXperimenter Kit.

Chapter 2

DSP Building Blocks

The thesis presents an implementation of a general digital signal processing framework for both the transmit and receive chains of a wireless system. Our goal is to implement, in software, three of the common DSP building blocks that exist in today's wireless systems, namely the *encoder*, *constellation mapper*, and *OFDM*. Figure 2-1 illustrates the building blocks developed in each chain. The present implementation is meant to be a very general and basic DSP framework that would give a sense of how a complete implementation would compare to other implementations of SDRs. Hence, many features such as QAM equalizers, synchronization, scramblers, and CFO/SFO corrections have been omitted for the purposes of keeping the system simple.

The elegance of developing these building blocks in software lies in the flexibility it provides us with inventing new protocols and adding other blocks to each chain. Hence, the building blocks presented here are by no means a description nor a blueprint of how one should approach the digital signal processing of the SDR, and the hope is that the framework presented in this project lays the foundation for future innovations in designing wireless system protocols.

Furthermore, a software implementation of these building blocks enables us to easily define various types of processing that occur within each block as well as the ability to modify the parameters that are used for each type of processing. The following sections further elaborate on this flexibility and present the different versions

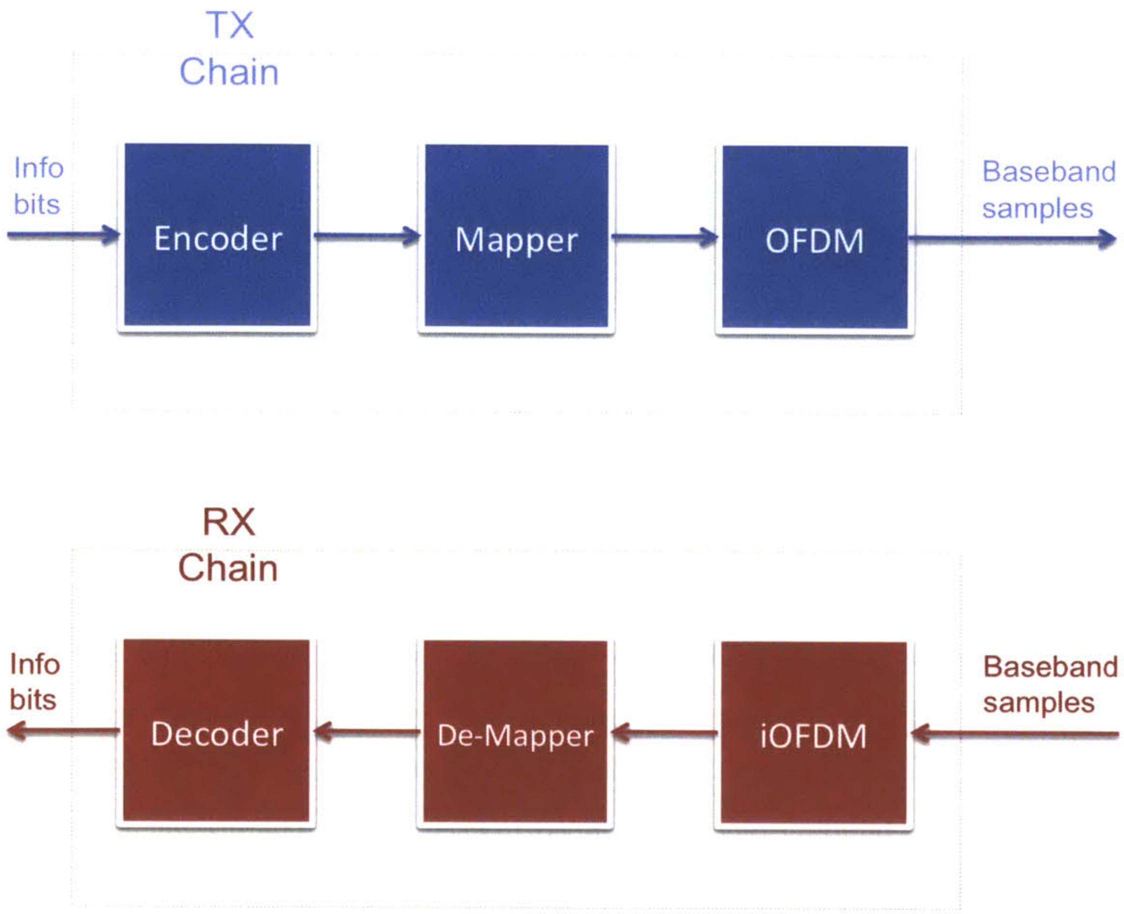

Figure 2-1: Proposed DSP building blocks to implement in software.

of processing that was implemented for each of these general DSP building blocks.

2.1 Encoder/Decoder

The encoding and decoding building blocks handle the processing between the information bits and codewords. By adding redundancy to the information bits, the encoder improves the chances of the received bits being properly decoded in the face of noise. In the present thesis I have implemented two of the common types of encoding/decoding schemes on the OMAP-L138 processor:

- Linear Block Encoding with Syndrome Decoding
- Convolutional Encoding with Viterbi Decoding

The following two subsections describe these two techniques, and the next chapter digs deeper into their software implementations.

2.1.1 Linear Block Encoding with Syndrome Decoding

Linear block encoding involves taking k-bit blocks of the information bits and converting each block into an n-bit coded block via a linear transformation that is defined by a *generator matrix*. This can be shown in mathematical terms as follows:

$$D.G = C$$

where D is the $1 \times k$ vector of the information bits, G is the $k \times n$ generator matrix, C is the $1 \times n$ codeword vector. All addition operations are done modulo 2.

Similarly, the decoding block is defined by a $(n-k) \times n$ *parity check matrix*, H, that has the following property:

$$H.C^T = 0$$

for any valid codeword C.

If the generator matrix yields a systematic form of the codeword (where C is of the form $D_1D_2...D_kP_1P_2...P_{n-k}$) then the matrix G can be decomposed into a $k \times k$ identity matrix concatenated horizontally with a $k \times (n-k)$ matrix (A) that defines the values for the code. We thus have:

$$G = I_{k \times k}|A$$

Furthermore, the parity check matrix associated with this generator matrix will be of the following form:

$$H = A^T|I_{(n-k)\times(n-k)}$$

Let R be the received word at the decoder, which may contain errors. Hence R

has the form

$$R = C + E$$

where C is the code word sent at the encoder and E is the error vector associated with noise. After receiving R, the decoder applies the parity check matrix to it as follows:

$$H.R^T = H.(C+E)^T = 0 + H.E^T = H.E^T = Q$$

Thus the vector Q that results from this operation depends on the error vector, E, and not the codeword sent, C.

Suppose we wish to decode all received words with at most t errors. Hence, there will be a total of

$$\binom{n}{0} + \binom{n}{1} + \binom{n}{2} + ... + \binom{n}{t}$$

patterns for the error vector E, and the same number of patterns for $H.E^T$. Assuming that our system yields a maximum of t errors, the resulting vector Q from the decoding operation described above can only be of one of the latter patterns. Since these patterns have a one-to-one relationship with the error vector patterns, the resulting vector Q maps to a unique error vector E that indicates which bits have the errors. We call each of these resulting vectors a *syndrome*. Therefore, by pre-computing which syndrome maps to which error vector, we can infer which bits have an error by mapping the syndrome that results from the decoding operation to an error vector.

2.1.2 Convolutional Encoding with Viterbi Decoding

Similar to linear block encoding, convolutional codes perform linear operations on the information bits. Instead of blocks, however, convolutional encoding operates on a *sliding window* of information bits. Specifically, with a window size of k, the encoder generates p parity bits as a result of encoding the information bits within the window. The window advances by one bit each cycle. Clearly, the larger the window size k, the more information bits are encoded each cycle and the more resilient the code.

A generator matrix can be defined for a convolutional code. The encoding can be shown in mathematical terms as follows:

$$G.W = P$$

where G is the $p \times k$ generator matrix, W is the $k \times 1$ sliding window, and P is the $p \times 1$ parity bits that the encoder outputs.

The Viterbi decoder is a popular method for decoding convolutional codes. The decoder uses an extended version of a state machine, called a *trellis*, to find the most likely sequence of transmitted bits. Each column in the trellis is composed of the different states (i.e., the different combinations of the first $k-1$ bits of the sliding window). Moreover, each state points to two other states in the next column, which represent the state transitions upon receiving a "0" or "1" bit. Each such transition also indicates the parity bits that the encoder would generate had it made that transition. An example of a trellis is shown in Figure 2-2.

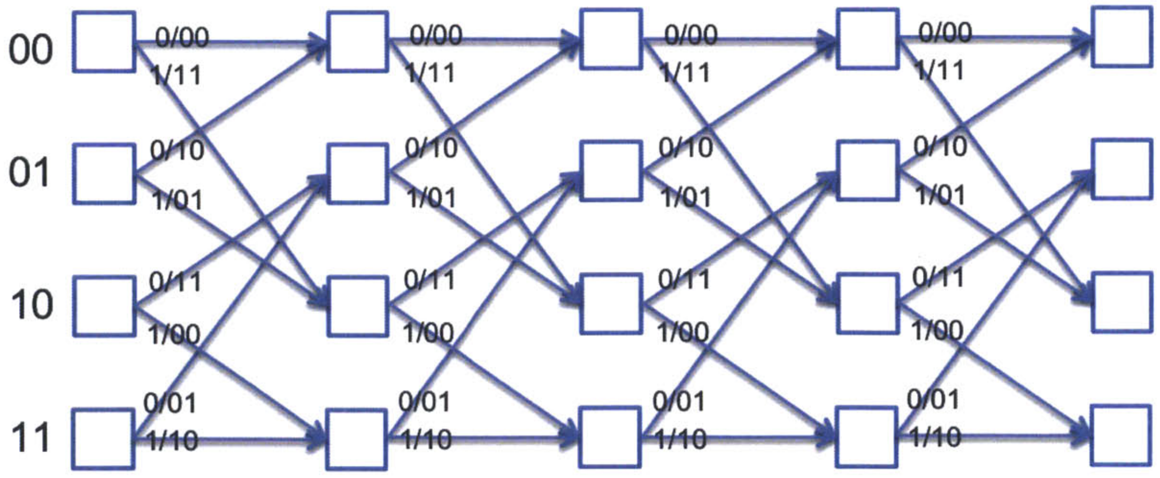

Figure 2-2: An example of a trellis used for Viterbi decoding. Each square represents a state and the arrows represent the transitions. The notation a/bc on each arrow indicates that bit a was encoded and the parity bits bc were sent as a result of the transition.

Using this trellis, the Viterbi decoder finds the most likely path the encoder took given the received parity bits. The algorithm finds this path using two metrics: the branch metric (BM) and the path metric (PM). The branch metric is associated with each state transition and is equal to the *Hamming distance* between the received parity bits and the expected parity bits for that transition (i.e., the number of positions at which these two strings of bits are different). The path metric is associated with each state and measures the Hamming distance between the sequence of parity bits along the most likely path from the initial state to this state with the sequence of received parity bits. The path metric of a state s at step i with predecessor states α and β can by iteratively defined as follows:

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

Thus, after a certain number of transitions, the algorithm chooses the state with the smallest path metric and traces the most likely path ending at that state back to the initial state while returning the bit sent on each transition.

2.2 Constellation Mapper/De-mapper

The constellation mapper is a digital modulation scheme whereby the codeword bits are "mapped" into two-dimensional vectors, referred to as the constellation points, whose values dictate the amplitude of the sine and cosine carrier waves. In our general framework, the values are then passed on to the OFDM block that further modulates these values, as described in the next section. The de-mapper performs the exact reverse of the mapper, mapping the constellation points back to the codeword bits after finding the constellation point that has the smallest Euclidean distance to the received I-Q sample.

Depending on the number and location of the constellation points, the modulation scheme is associated with different names (i.e., BPSK, QAM4, QAM16, etc.). The constellation maps implemented for the current framework are shown in Figure 2-3. Note that adjacent points on the same vertical or horizontal axis are equidistant from one another and that the exact I-Q (constellation) values depend on the power with which the complex samples are being sent.

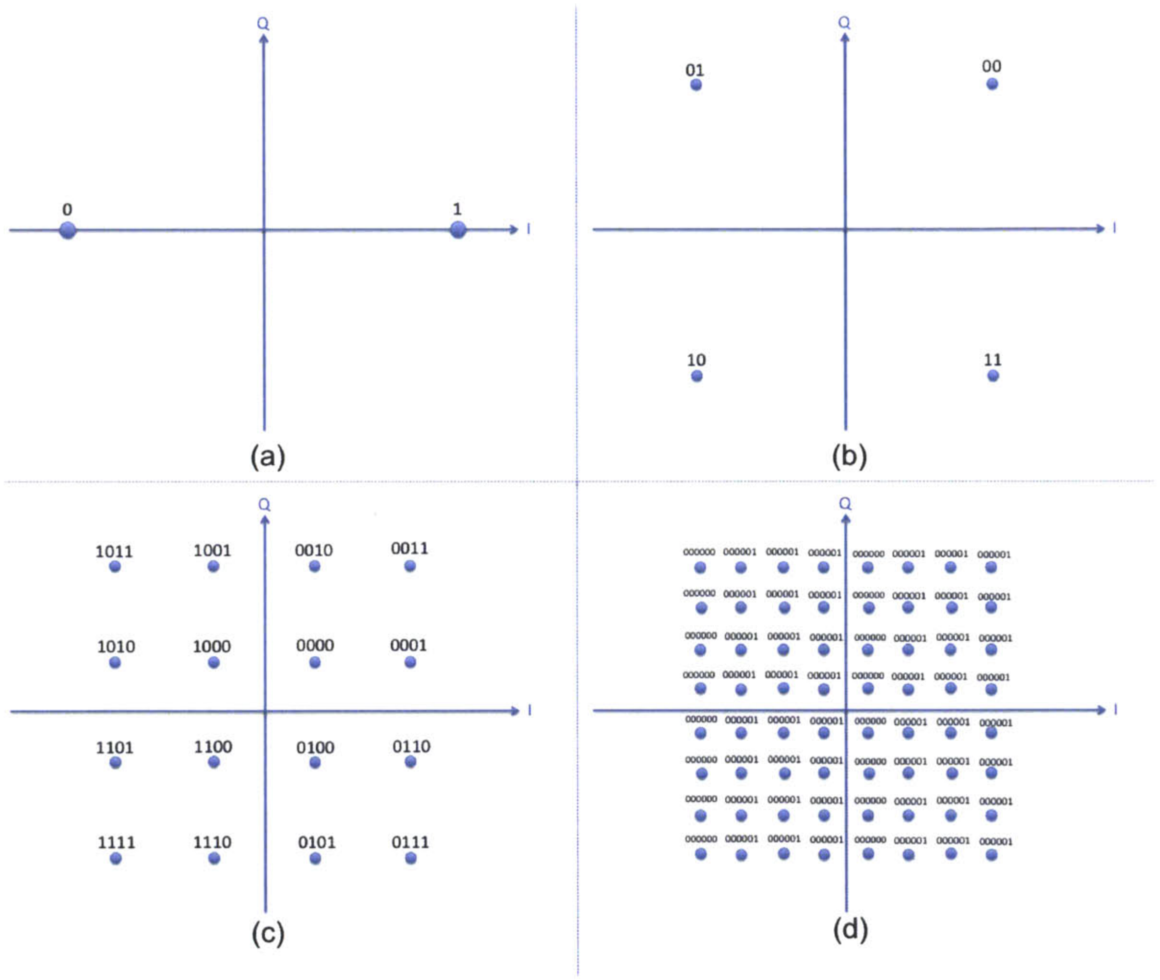

Figure 2-3: Constellation maps implemented for the current framework: (a) BPSK, (b) QAM4, (c) QAM16, and (d) QAM64.

2.3 Orthogonal Frequency-Division Multiplexing (OFDM)

The OFDM block performs further modulation on the complex samples in order to make the signal more resilient to wireless phenomena such as multi-path and frequency selectivity.

Two parameters are defined for this block: the number of subcarriers (N) and the length of the cyclic prefix (C). For the OFDM block in the TX chain, the following two steps are taken:

1. *iFFT:* the inverse discrete Fourier transform (iDFT) of the input sample block of length N is computed to form the output block. To perform the computation, the inverse *Fast Fourier Transform* is used in the current implementation.

2. *Cyclic Prefix:* the last C values of the previous operation is appended to the beginning of the output block.

Denote the input sample block (computed by the mapper) as $X[k]$ for $0 \leq k \leq N-1$, and let $x[n]$ be the inverse discrete Fourier transform of $X[k]$ (i.e., $x[n] = \text{IDFT}\{X[k]\}$ for $0 \leq n \leq N-1$. Hence, $x[n]$ is the output of step 1 above. Let $\tilde{x}[n]$, $-C \leq n \leq N-1$, denote the output of step 2 where

$$\tilde{x}[n] = \begin{cases} x[N+n] & \text{if } -C \leq n < 0, \\ x[n] & \text{if } 0 \leq n \leq N. \end{cases}$$

Suppose the discrete-time impulse response of the channel is $h[n]$. Assuming no noise and that $h[n]$ has at most C taps, the RX chain will receive the linear convolution of $\tilde{x}[n]$ and $h[n]$, which is the equivalent of the circular convolution of $x[n]$ and $h[n]$:

$$y[n] = \tilde{x}[n] * h[n] = x[n] \circledast h[n]$$

for $0 \leq n \leq N-1$. According to the properties of the DFT we have

$$Y[k] = \text{DFT}\{y[n]\} = \text{DFT}\{x[n] \circledast h[n]\} = \text{DFT}\{x[n]\} \times \text{DFT}\{h[n]\} = X[k]H[k]$$

for $0 \leq k \leq N-1$, where $H[k] = \text{DFT}\{h[n]\}$.

Therefore, to find the original values of $X[k]$, the iOFDM block in the RX chain must take the following steps upon receiving $y[n]$:

1. Remove the first C values of $y[n]$.
2. Compute the N-point fast Fourier transform (FFT) of the remaining values of $y[n]$.
3. Divide the result by the DFT of the discrete-time impulse response of the channel.

The final result is then given as input to the de-mapper block.

Chapter 3

Software Design

This chapter delves into the structure of the framework's software implementation. Section 3.1 presents an overview of the implementation and the code structure. Section 3.2 describes the DSPLIB routines. Section 3.3 illustrates the user interface to the framework, and Section 3.4 discusses the pipelining architecture that allows the framework to achieve high throughput.

3.1 Code Structure

The framework's software implementation is composed of the transmitter (TX) chain and the receiver (RX) chain, as shown in Figure 2-1. The TX chain takes information bits as input and outputs complex baseband samples, while the reverse process takes place in the RX chain. The code structure of each chain involves header files associated with each of the blocks in Figure 2-1, in addition to the other components. These files along with the main source file are as follows:

- *tx_process.c/rx_process.c*: main source file for each chain where the pipelining process (described in Section 3.4) occurs.
- *encoder.h/decoder.h*: header files for the encoding and decoding blocks. This file includes functions for initializing and processing the linear block and convolutional encodings and their respective decodings, which are called from the

tx_process.c/rx_process.c source files.

- *mapper.h/demapper.h*: header file for the mapper/de-mapper block, which include the initialization and processing functions.
- *ofdm.h/iofdm.h*: header file for the OFDM/iOFDM block, which include the initialization and processing functions.
- *parameters.h*: header file for setting the user-defined parameters of the system. (Further discussed in Section 3.3).
- *queue.h*: header file associated with the data-structure used for pipelining data between blocks. (Further discussed in Section 3.4).

Appendix A includes the code for each of these files.

3.2 DSPLIB

The OMAP's DSP processor offers a collection of general-purpose signal-processing routines that are callable from C programs. These routines are assembly-optimized DSP functions that allow for intensive digital signal processing computations to occur in real-time at the flexibility of programming in software. They are included in the processor's DSP Function Library (DSPLIB), and much of the present implementation makes substantial use of them. Table 3.1 shows a list of the DSPLIB functions used in the implementation, along with a description for each.

Table 3.1: List of DSPLIB functions used in the present implementation.

Function	**Description**
void DSPF_sp_mat_mul (float *x, int r1, int c1, float *y, int c2, float *r)	Matrix multiplication
void DSPF_sp_mat_trans (float *x, int rows, int cols, float *r)	Matrix transpose
void DSPF_sp_w_vec (float *x, float *y, float m, float *r, int nr)	Weighted vector sum
float DSPF_sp_vecsum_sq (float *x, int n)	Sum of squares
void DSPF_sp_cfftr2_dit (float *x, float *w, short n)	Complex radix 2 FFT using DIT
void DSPF_sp_icfftr2_dif (float *x, float *w, short n)	Complex radix 2 inverse FFT using DIF

3.3 User Interface

In addition to achieving an optimal performance for real-time applications, an objective for the present implementation has been to provide a framework that any user can employ without having any knowledge about the internal structure or complexities within each of the building blocks. All the user must do is to set his desired parameters for each DSP block through the user interface that the implementation provides. This objective is in lines with the long-term intentions of a wireless software-defined network where abstractions are required to hide away the complexity and to allow for modifiable protocols. A list of the parameters that a user can set for the different building blocks is provided in Table 3.2.

Table 3.2: List of DSP Building Block Parameters.

Block	**Parameter**	**Description**
Encoder	ENCODER_V	Encoder version: 0 for linear block, 1 for convolutional
	ENCODER_LB_K	Linear Block Encoding: Message Length
	ENCODER_LB_N	Linear Block Encoding: Codeword Length
	code_matrix	Linear Block Encoding: Code Matrix
	ENCODER_CV_K	Convolutional Encoding: Constraint Length
	ENCODER_CV_P	Convolutional Encoding: Number of Parity Bits
	ENCODER_CV_TRIALS (RX chain only)	Convolutional Encoding: Number of Trials for Viterbi Decoding
	gen_poly_matrix	Convolutional Encoding: Generating Polynomial Matrix
Mapper	QAM	Modulation Scheme: 2 for BPSK, 4 for QAM4, 16 for QAM16, and 64 for QAM64
	QAM_POWER (TX chain only)	Maximum Power for Constellation Points
OFDM	OFDM_N	Number of OFDM Sub-Carriers
	OFDM_CP	Cyclic Prefix Length

3.4 Pipelining

To achieve maximum throughput, the data between the blocks is pipelined through the use of queues in order to avoid starvation at the ends of the two chains, as well as to prevent blocks from being called with insufficient data. The reason why the system may encounter starvation or insufficient data at a block is that an operation (or an integer number of operations) for one block does not necessarily produce the exact number of input data samples needed at the following block to perform an operation. Hence, the number of operations performed by each block varies from round to round, and a pipelining architecture is needed to schedule the operation of each block.

For example, consider the following parameters for each block in the TX chain:

- Encoder: linear block with $ENCODER\_LB\_K = 4$, $ENCODER\_LB\_N = 7$
- Mapper: QAM64
- OFDM: $OFDM\_N = 32$, $OFDM\_CP = 8$

In the first round, two operations of the encoder block is needed to satisfy the 8-bit input to the mapper. Notice that these two operations in fact will yield $2 \times 7 = 14$ bits, which means that $14 - 8 = 6$ bits will be left over from these two operations. Thus, in the next round the encoder block must only be called once to satisfy the mapper's 8-bit input. Furthermore, once the mapper has created a total of 32 complex samples and has ran out of bits to process, the scheduler must have the OFDM block perform its operation instead of the encoder. This is because the goal is to achieve maximum throughput, and since the OFDM block is closest to the output, it must be given priority to the encoder block.

The relevant pseudocode for pipelining in the TX chain and RX chain is shown in Figure 5 and Figure 6, respectively. For the TX chain, priority is given to the OFDM block in order to output the samples at the fastest possible rate. Priority is then given to the mapper, and then to the encoder. The reverse order of priority is true for the RX chain. Note that "x_queue" refers to the output queue of block x.

```
loop
  if mapper_queue.count $\geq OFDM\_N$ then
    OFDM()
  else if encoder_queue.count $\geq \log_2 QAM$ then
    Mapper()
  else
    Encoder()
  end if
end loop
```

Figure 3-1: TX Chain Pipelining Pseudocode.

```
loop
  if demapper_queue.count $\geq$ number of bits required for LB or CV decoding then
    Decoder()
  else if iofdm_queue.count $\geq$ 2) then
    Demapper()
  else
    iOFDM()
  end if
end loop
```

Figure 3-2: RX Chain Pipelining Pseudocode.

Chapter 4

Evaluation

To evaluate the performance of the aforementioned DSP framework on the OMAP-L138 processor, the framework was implemented in two different versions:

- *Normal version:* this version of the implementation was meant to mimic the performance that would be achieved with other software-defined radios, such as the USRP, where all the digital signal processing is done via C and no assembly-level optimizations are made. Hence, no use was made of any of the routines inside DSPLIB for this version.

- *Optimized version:* in this version of the implementation, the DSPLIB routines were used as much as possible in order to achieve the best performance from the OMAP-L138 processor.

4.1 Bit-Rate and Latency

Each version was run against five combinations of the parameters described in the previous section. The combinations are shown in Table 4.1. The evaluations were done as follows: for each run, the number of cycles needed to process about 1000 bits in the TX chain was measured for each version and the output samples were recorded. (The exact number of processed bits depended on how many were needed to produce an integer number of OFDM blocks). The result was used to calculate the achievable bit-rates for the TX chain as follows:

$$bit\_rate = \frac{bits \times clock\_freq}{cycles}$$

where *bits* refers to the number of bits processed, *clock_freq* refers to the frequency of the processor's clock (375 MHz), and *cycles* refers to the number of clock cycles that was measured to complete the processing.

The measured bit-rates are shown in Figure 4-1. The output samples were then processed in MATLAB to simulate a wireless channel. The resulting samples were fed to the RX chain of the two versions, and the reverse process for the TX chain was repeated to measure the bit-rates for the RX chain, which are shown in Figure 4-2.

As the results in Figure 4-1 and Figure 4-2 indicate, the optimized version outperforms the normal version against all the different combinations for both chains. For the TX chain, the average improvement in performance across the five combinations is 85.4%, while for the RX chain, where the computations are much more intensive, the average improvement is 975.6%.

Table 4.1: Combinations of DSP building block parameters against which the Normal version and the Optimized version were run against.

Combination	**Encoder**	**Mapper**	**OFDM**
1	Linear Block, $rate = 3/4$	BPSK	$N = 32$, $CP = 8$
2	Linear Block, $rate = 2/3$	QAM4	$N = 64$, $CP = 16$
3	Convolutional, $rate = 1/4$	QAM64	$N = 64$, $CP = 16$
4	Convolutional, $rate = 1/2$	QAM16	$N = 128$, $CP = 32$
5	Linear Block, $rate = 1/2$	QAM64	$N = 128$, $CP = 32$

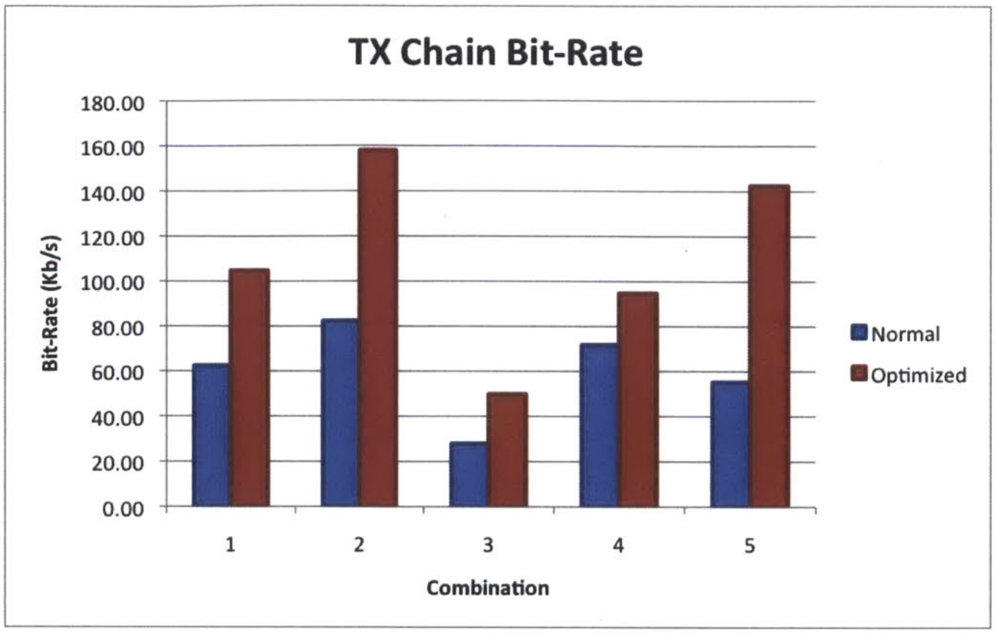

Figure 4-1: Bit-Rate comparison between the normal and optimized versions of the framework implementation in the TX chain against five different combinations of block parameters.

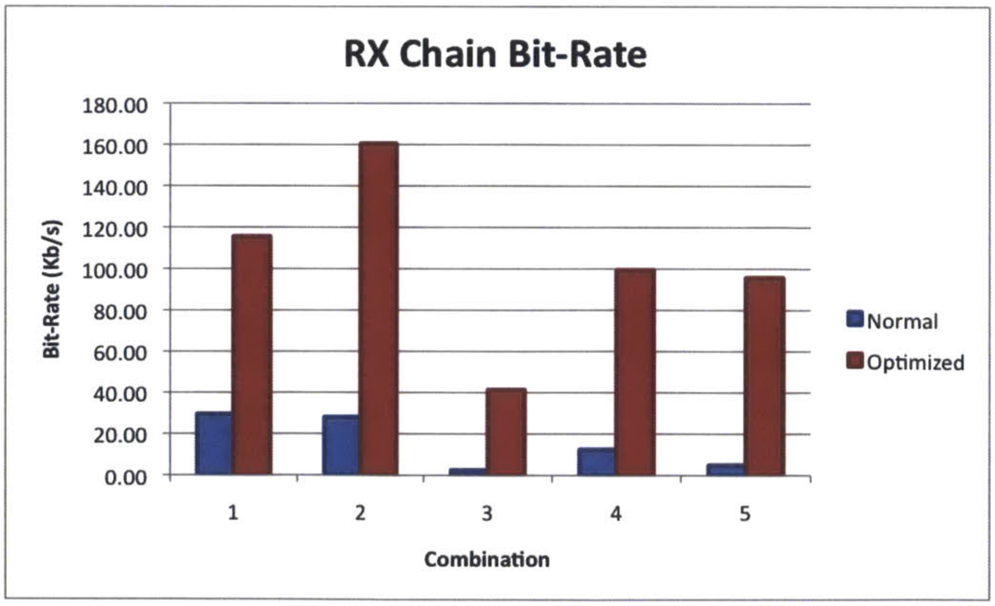

Figure 4-2: Bit-Rate comparison between the normal and optimized versions of the framework implementation in the RX chain against five different combinations of block parameters.

Furthermore, from the bit-rate we can also infer the latency associated with creating/processing a packet in the TX/RX chain:

$$latency = bit\_rate \times packet\_size.$$

Figure 4-3 and Figure 4-4 show the latencies for a packet of size 1000 *bytes* (8000 *bits*) in the TX chain and RX chain, respectively. Note that although we see a much better performance with the optimized version of the implementation, we still observe very high latencies for creating/processing a packet. This indeed hinders the use of even the optimized version in protocols with strict time constraints (such as 802.11), and we leave it to future work to further optimize this version of the implementation in order to make it more practical for real-world applications.

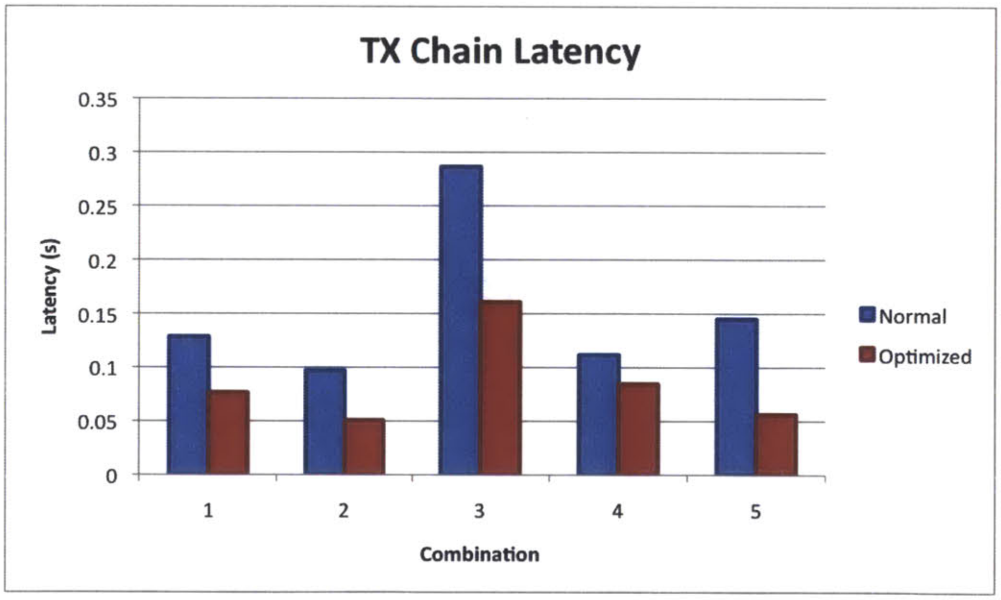

Figure 4-3: Latency comparison between the normal and optimized versions of the framework implementation in the TX chain for a packet of size 1000 *bytes* against five different combinations of block parameters.

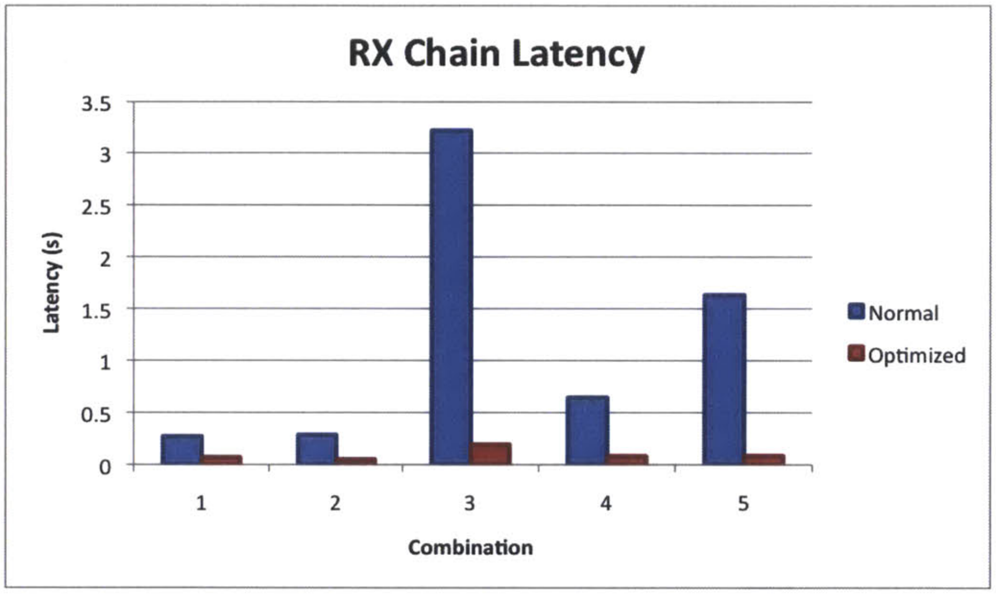

Figure 4-4: Latency comparison between the normal and optimized versions of the framework implementation in the RX chain for a packet of size 1000 *bytes* against five different combinations of block parameters.

4.2 Bandwidth

Using the same measurements done for the bit-rate, one can also calculate the bandwidth that can be achieved with each version for the different combinations. This can be done by measuring the number of complex samples that were outputted in the TX chain per unit time and the number of samples that were processed in the RX chain per unit time. More formally,

$$bandwidth = \frac{samples \times clock\_freq}{cycles}$$

where *samples* refers to the number of complex samples that were outputted/processed, *clock_freq* refers to the frequency of the processor's clock (375 MHz), and *cycles* refers to the number of clock cycles that was measured to complete the processing. The results are shown in Figure 4-5 and Figure 4-6.

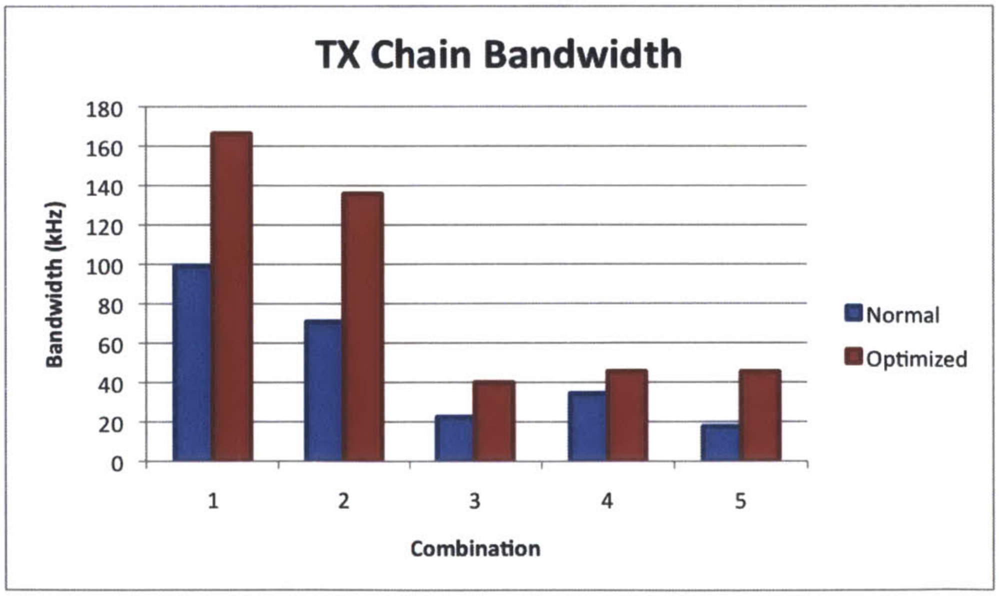

Figure 4-5: Bandwidth comparison between the normal and optimized versions of the framework implementation in the TX chain against five different combinations of block parameters.

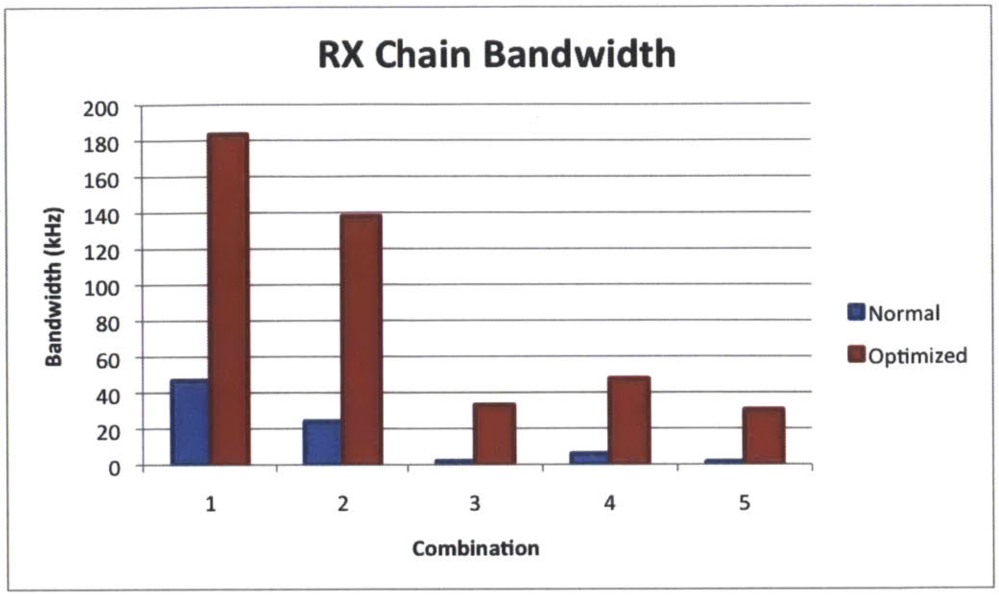

Figure 4-6: Bandwidth comparison between the normal and optimized versions of the framework implementation in the RX chain against five different combinations of block parameters.

4.3 Block Analysis

This section presents an analysis of the amount each building block in the system contributes to the overall improvement in performance. The contribution made by a particular block also varies by setting different parameters for that block. To measure these contributions, the number of cycles needed to complete one operation of each block was measured and the latency of that operation was computed as follows:

$$latency = \frac{cycles}{clock\_freq}$$

where $clock\_freq$ refers to the frequency of the processor's clock (375 MHz). An operation for each block is defined to be a single call of that block's function (see Figures 3-1 and 3-2). The following lists the operation that takes place when the function of each block is called:

- *Encoder, Linear Block*: encoding one message block (of length $ENCODER\_LB\_K$) into one codeword (of length $ENCODER\_LB\_N$).

- *Encoder, Convolutional*: generation of $ENCODER\_CV\_P$ parity bits.
- *Decoder, Linear Block*: decoding one codeword (of length $ENCODER\_LB\_N$) into one message block (of length $ENCODER\_LB\_K$).
- *Decoder, Convolutional*: Viterbi decoding for $ENCODER\_CV\_TRIALS$ trials.
- *Mapper*: map $\log_2 QAM$ bits into a constellation point.
- *De-mapper*: de-map a constellation point into $\log_2 QAM$ bits.
- *OFDM*: construction of one OFDM block (with length $OFDM\_N+OFDM\_CP$ complex samples).
- *iOFDM*: deconstruction of one OFDM block (with length $OFDM\_N+OFDM\_CP$ complex samples) into constellation points.

Figure 4-7 shows the improvement that the encoder block exhibited using linear block encoding for different code rates. Note that the code rate for this type of encoding is simply the ratio of the message block length to the codeword length (i.e., $\frac{ENCODER\_LB\_K}{ENCODER\_LB\_N}$). For these measurements, the codeword length was set to 48 bits and the message length was varied from 24 bits to 40 bits to yield the given rates. The average gain in performance for this type of encoding is 787.7%.

Figure 4-8 shows the improvement that the encoder block achieved using convolutional encoding for different code rates. The code rate for this type of encoding is simply the inverse of the number of parity bits sent for each input bit (i.e., $\frac{1}{ENCODER\_CV\_P}$). The average gain in performance for this type of encoding is 93.7%, which is less compared to the linear block encoding case, although the latency is smaller.

Figure 4-9 and Figure 4-10 show the improvement that the mapper and OFDM blocks achieved, which were 59.2% and 118.9% respectively. The results of Figures 4-7, 4-8, 4-9, and 4-10 are all associated with the TX chain.

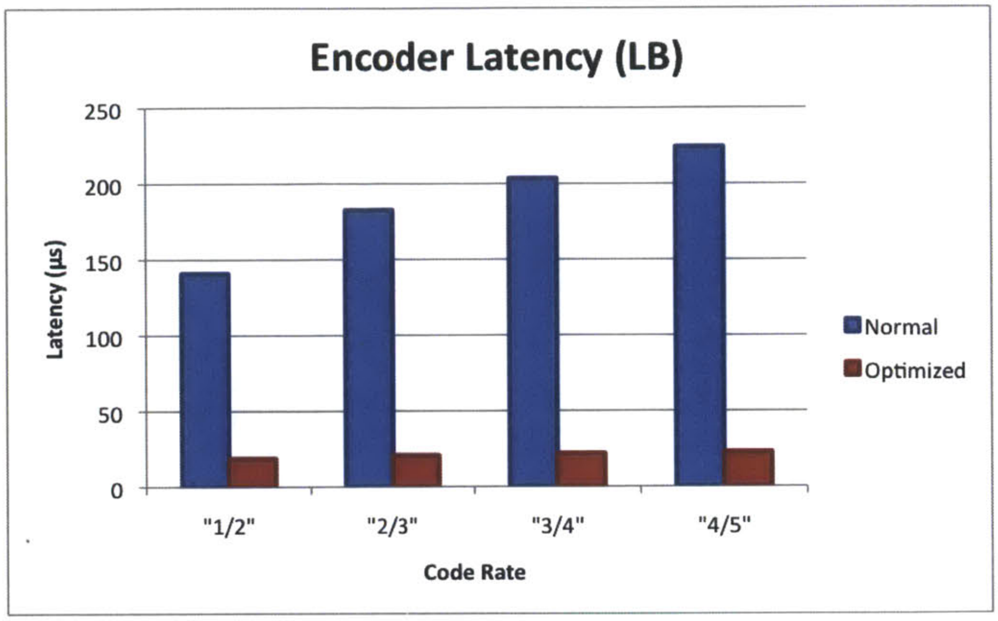

Figure 4-7: Performance comparison of linear block encoding for the normal and optimized versions. The figure compares the latency associated with converting a message block of length *ENCODER_LB_K* into a codeword of length *ENCODER_LB_N* between the two versions of the implementation for different code rates (for each rate, *ENCODER_LB_N* was set to 48 bits and *ENCODER_LB_K* was chosen accordingly). The average gain in performance is 787.7%.

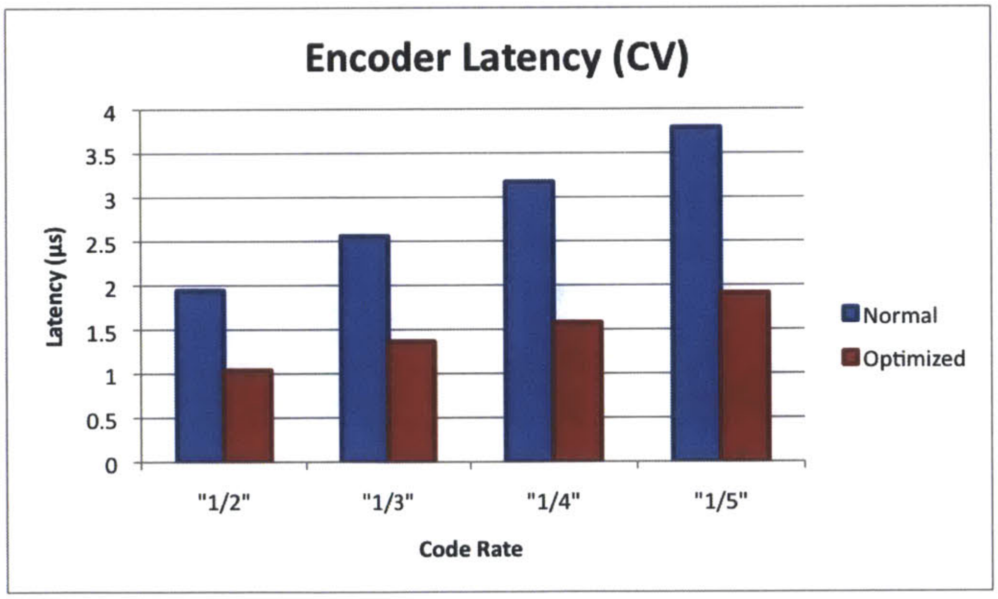

Figure 4-8: Performance comparison of convolutional encoding for the normal and optimized versions. The figure compares the latency associated with the generation of *ENCODER_CV_P* parity bits between the two versions of the implementation for different code rates (with *ENCODER_CV_P* values of 2, 3, 4, and 5 bits). The average gain in performance is 93.7%.

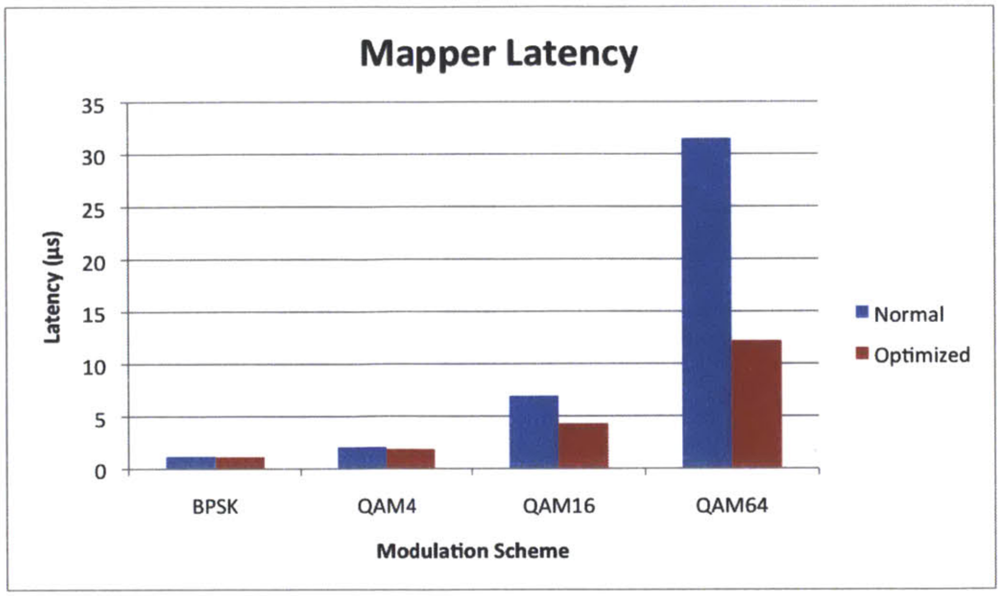

Figure 4-9: Performance comparison of the mapper block for the normal and optimized versions. The figure compares the latency associated with the mapping of $\log_2 QAM$ bits into a constellation point between the two versions of the implementation for different modulation schemes. The average gain in performance is 59.2%.

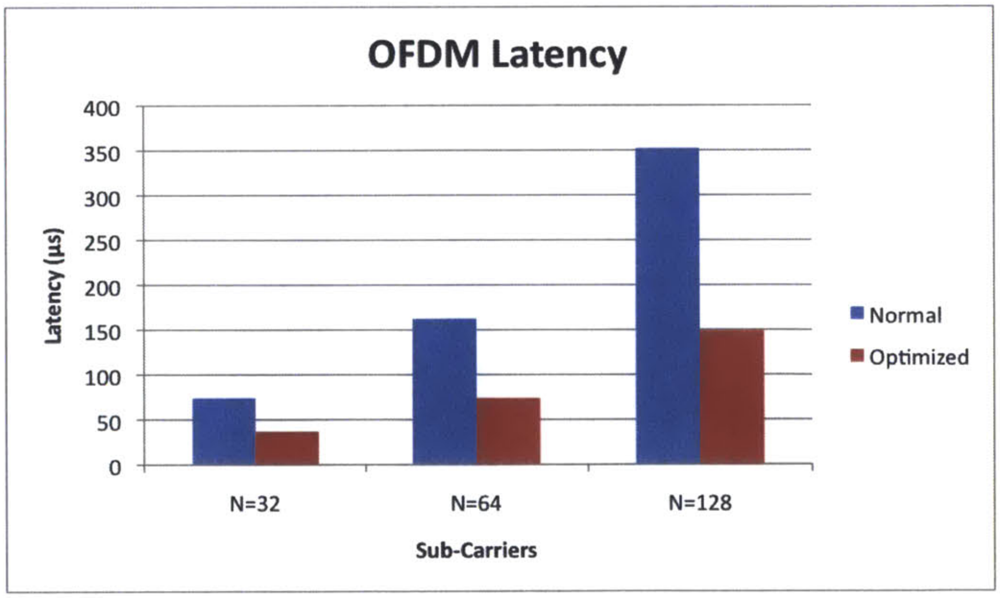

Figure 4-10: Performance comparison of the OFDM block for the normal and optimized versions. The figure compares the latency associated with the construction of one OFDM block (with length $OFDM\_N + OFDM\_CP$ complex samples) between the two versions of the implementation for different numbers of sub-carriers. Note that the length of the cyclic prefix ($OFDM\_CP$) was set to $1/4^{th}$ of the number of sub-carriers ($OFDM\_N$) for each case. The average gain in performance is 118.9%.

Figures 4-11, 4-12, 4-13, and 4-14 show the improvement for the corresponding blocks in the RX chain. Note that as opposed to the case of the TX chain, the demapper block achieved the largest performance gain (1818.9%). Moreover, notice that with Viterbi decoding for the decoding block, worst performance is achieved with the optimized version compared to the normal version, which indicates the inefficiency of the DSPLIB routines for this operation. Hence, an implementation that is optimized for performance would use the same exact code as in the normal version for this operation.

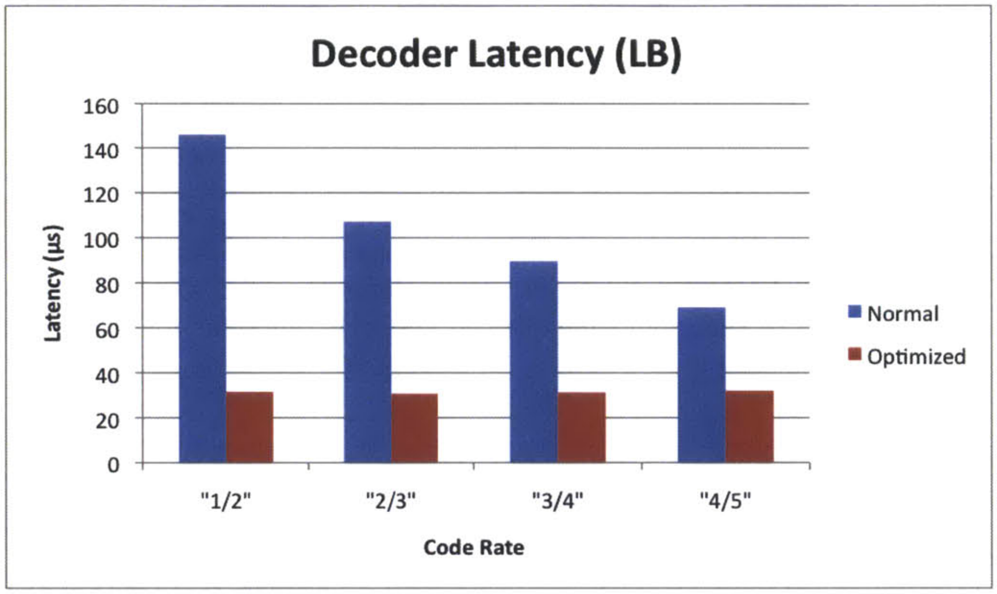

Figure 4-11: Performance comparison of Syndrome decoding for the normal and optimized versions. The figure compares the latency associated with decoding one codeword (of length $ENCODER\_LB\_N$) into one message block (of length $ENCODER\_LB\_K$) between the two versions of the implementation for different code rates (for each rate, $ENCODER\_LB\_N$ was set to 48 bits and $ENCODER\_LB\_K$ was chosen accordingly). The average gain in performance is 229%.

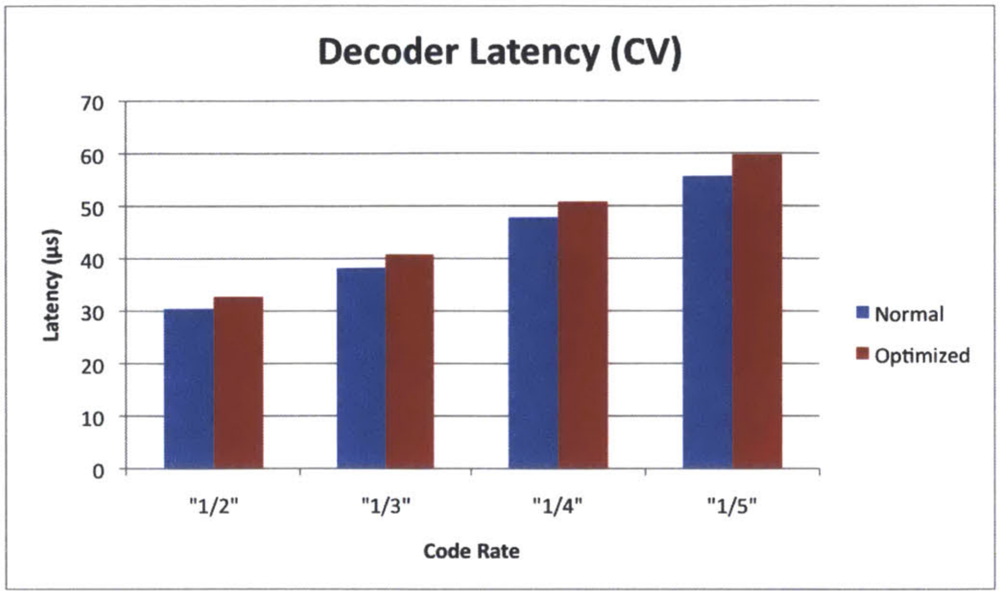

Figure 4-12: Performance comparison of Viterbi decoding for the normal and optimized versions. The figure compares the latency associated with the Viterbi decoding for $ENCODER\_CV\_TRIALS = 8$ trials. No performance gain was achieved for this type of decoding due to the inefficiency of the DSPLIB functions for this operation.

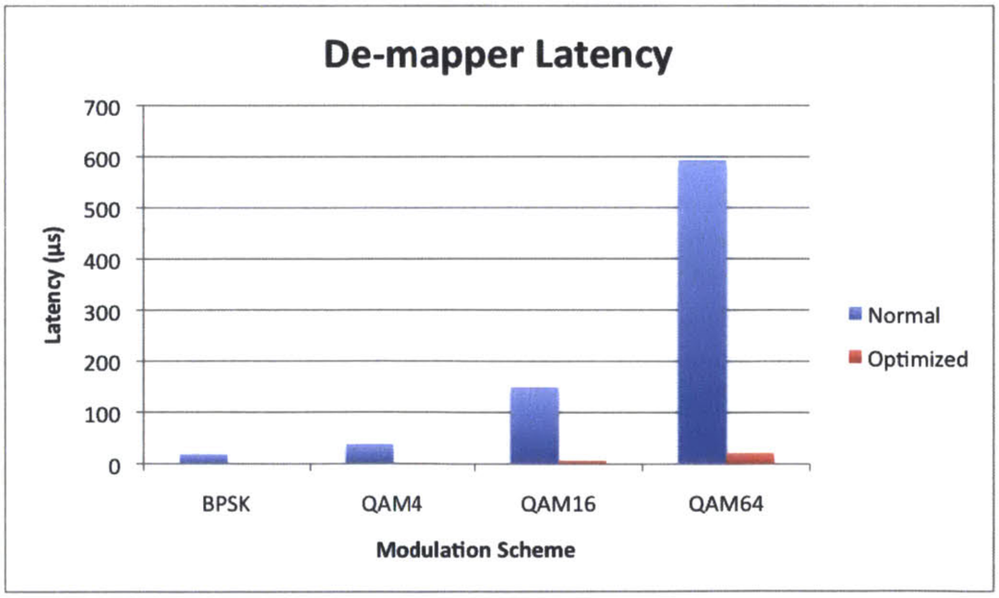

Figure 4-13: Performance comparison of the de-mapper block for the normal and optimized versions. The figure compares the latency associated with the de-mapping of a constellation point into $\log_2 QAM$ bits for different modulation schemes. The average gain in performance is 1818.9%.

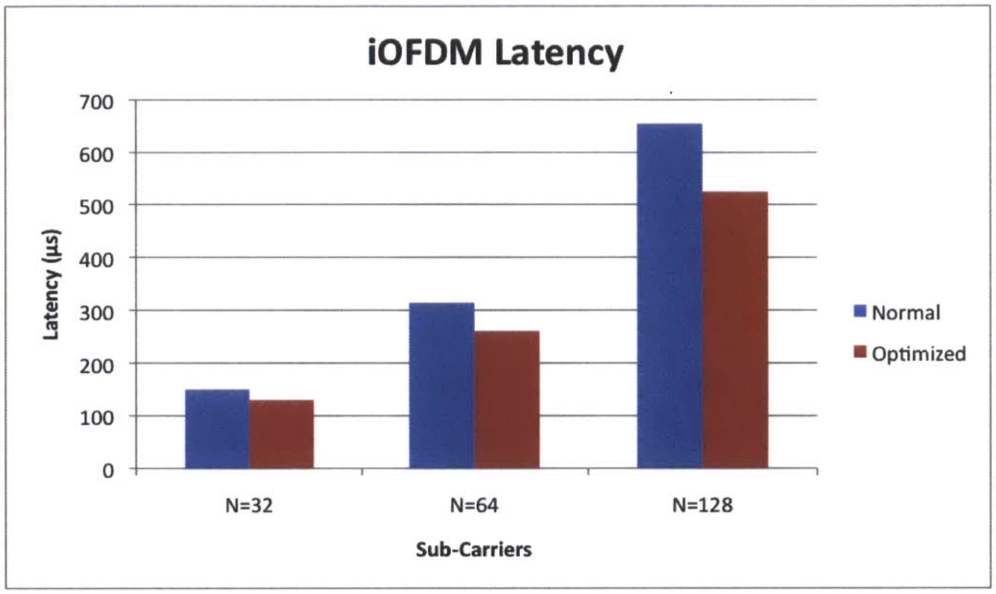

Figure 4-14: Performance comparison of the iOFDM block for the normal and optimized versions. The figure compares the latency associated with the deconstruction of one OFDM block (with length $OFDM\_N + OFDM\_CP$ complex samples) into constellation points for different numbers of sub-carriers. Note that the length of the cyclic prefix ($OFDM\_CP$) was set to $1/4^{th}$ of the number of sub-carriers ($OFDM\_N$) for each case. The average gain in performance is 20.4%.

Figures 4-15 and 4-16 summarize the latencies for each block as a percentage of the total latency corresponding to the construction/processing of an OFDM block in the TX/RX chain for each combination in Table 4.1. These figures also represent the amount of processing resource that is allocated to each block as a percentage of the total processing resource required for the different combinations in each chain.

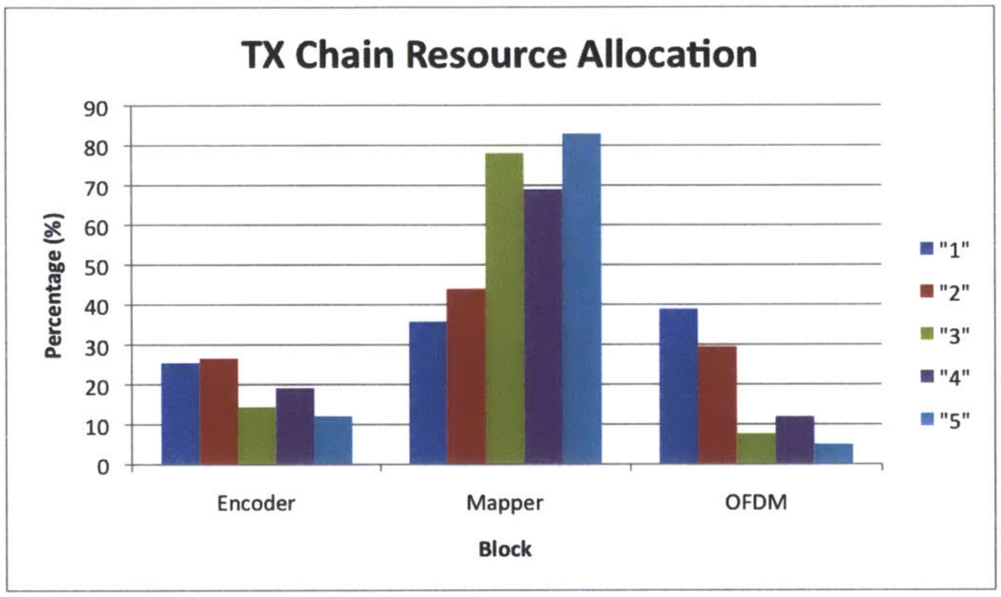

Figure 4-15: TX Chain summarization of processing resource for each block as a percentage of total. The figure can also be viewed as a comparison of the latency due to each block in the construction of an OFDM block.

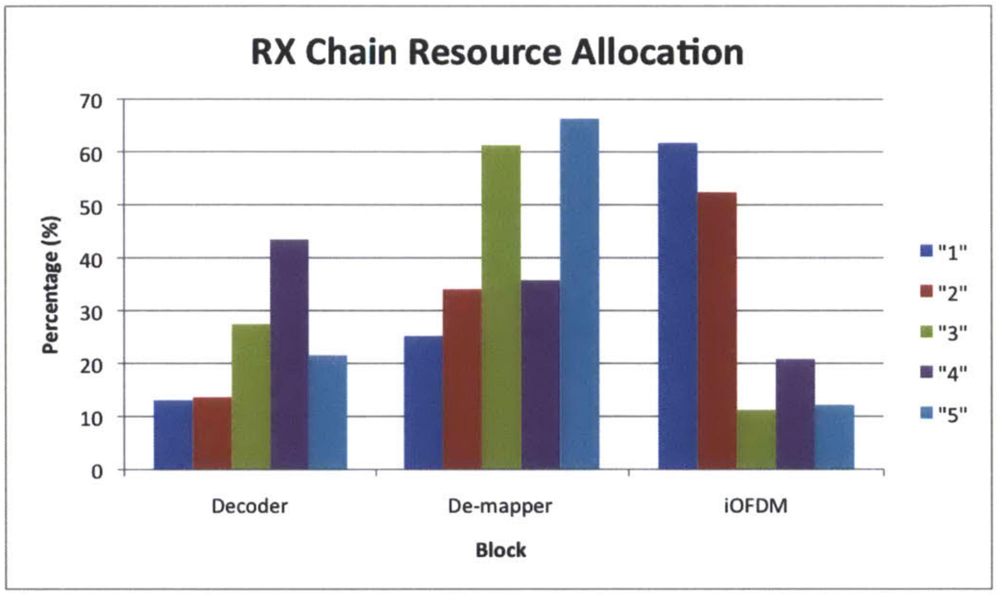

Figure 4-16: RX Chain summarization of processing resource for each block as a percentage of total. The figure can also be viewed as a comparison of the latency due to each block in the processing of an OFDM block.

Chapter 5

Future Work

The current thesis provided a very general DSP framework for the OMAP-L138 processor. The next step would be to complete the framework by adding additional building blocks and enhancing those that already exist. Afterwards, the processor is to be integrated with an analog front-end to form a complete software-defined radio in order to achieve the vision described in Section 1.2. Of course, designing the analog front-end carries with it challenges of its own, especially in maintaining our desire for a low-power software-defined radio that is small in size. Nevertheless, having eliminated the need for a PC and an FPGA for the DSP operations while achieving faster performance with the OMAP-L138 processor is surely considered to be a big step towards achieving this goal.

Chapter 6

Conclusion

The thesis discussed the implementation of a general DSP framework on TI's OMAP-L138 processor. This processor was used in order to overcome the disadvantages that exist in software-defined radios today: large size, dependence on a PC, and high latencies. Aside from being low-power and small in size, the processor offers a collection of assembly-level optimized routines for common DSP functions. These routines allow us to achieve the flexibility of implementing wireless protocols in software while having to trade in much less performance. Our evaluations also confirmed the enhanced performance of implementing protocols on this processor and showed substantial improvements to implementations that utilize non-optimized software.

Appendix A

Code

A.1 TX Chain Code

A.1.1 tx_process.c

```
#include <stdio.h>
#include <c6x.h>
#include <csl_types.h>
#include <cslr_dspintc.h>
#include <soc_OMAPL138.h>
#include <cslr_tmr.h>
#include <cslr_syscfg0_OMAPL138.h>
#include <math.h>
#include "parameters.h"
#include "queue.h"
#include "encoder.h"
#include "mapper.h"
#include "ofdm.h"

void Initialize(void);
void Process(void);
void Encoder(void);
void Mapper(void);
void OFDM(void);
```

```
/************Initialize Queues***********/
queue_f encoder_queue = {0,0,"Encoder_Queue"};
queue_f mapper_queue = {0,0,"Mapper_Queue"};
queue_f ofdm_queue = {0,0,"OFDM_Queue"};

int data_ptr = 0;

/*********MAIN FUNCTION*************/
void main (void)
{
        Initialize();

    while(1){
        Process();
    }
}

/*********INITIALIZATION************/
void Initialize(void)
{
        int i;
        int j;
        int n;

        for (j = 0; j<BUFLENGTH; j++){
                output_buffer[j] = 0;
                output_buffer_r[j] = 0;
                output_buffer_i[j] = 0;
                qam_buffer_r[j] = 0;
                qam_buffer_i[j] = 0;
        }

        encoder_lb_init();
        mapper_init();
        ofdm_init();
}

/*********PROCESS*************/
void Process(void){
        if (mapper_queue.count >= 2*(int)OFDM_N){
                OFDM();
        }else{
                if (encoder_queue.count >= log((int)QAM)/log(2)){
                        Mapper();
                }else {
                        Encoder();
                }
        }
}

/*********ENCODER BLOCK*************/
void Encoder(void)
{
        if ((int)ENCODER_V==0){
                encoder_lb (data, &data_ptr, &encoder_queue);
        }else{
                encoder_cv (data, &data_ptr, &encoder_queue);
        }
}

/*********MAPPER BLOCK*************/
void Mapper(void)
{
        mapper(&mapper_queue, &encoder_queue);
}

/*********OFDM BLOCK*************/
void OFDM(void)
{
        ofdm(&ofdm_queue, &mapper_queue);
}
```

A.1.2 parameters.h

```
#ifndef PARAMETERS_H_
#define PARAMETERS_H_

/*****************USER INTERFACE*****************/
#define ENCODER_V 0          //0 for LB, 1 for CV
#define ENCODER_LB_N 48
#define ENCODER_LB_K 24
#define ENCODER_CV_K 3
#define ENCODER_CV_P 2
#define QAM 64
#define QAM_POWER 1
#define OFDM_N 128
#define OFDM_CP 32
/************************************************/

#define ENCODER_LB_N_K (int)ENCODERN-(int)ENCODERK
#define ENCODER_LB_CM_SIZE (int)ENCODERK*(int)ENCODERN_K
#define ENCODER_LB_GM_SIZE (int)ENCODERK*(int)ENCODERN
#define ENCODER_CV_GPM_SIZE (int)ENCODER_CV_K*(int)ENCODER_CV_P
#define PI 3.14159265358979
#define MAXQUEUESIZE 6000

#include "input_data.h"
#include "dsplib674x.h"

float gen_matrix[ENCODER_LB_GM_SIZE];


/*****************USER INTERFACE*****************/
float gen_poly_matrix[ENCODER_CV_GPM_SIZE] = {0,1,1,1,1,1};
float code_matrix[ENCODER_LB_CM_SIZE] =
    {1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,
        1,0,1,0,1,0,1,1,0,0,1,0,0,1,1,0,1,0,0,0,0,1,1,0,
        0,0,0,0,1,0,1,1,1,0,0,0,1,1,1,0,1,0,1,0,0,0,1,1,
        0,1,1,0,1,1,1,1,0,0,1,0,1,0,1,0,1,0,1,0,0,0,1,0,
        1,0,0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,0,1,0,1,1,
        1,0,1,0,1,0,0,0,0,0,1,0,1,1,0,0,1,0,1,0,1,0,1,0,
        1,1,1,0,0,1,1,0,1,0,1,0,1,0,0,0,1,0,1,0,1,0,1,0,
        0,0,1,0,1,0,1,0,0,0,1,1,1,0,1,0,1,1,0,0,0,0,1,1,
        1,0,0,0,1,0,1,1,1,0,1,0,0,1,1,0,1,0,1,0,1,1,1,0,
        0,0,1,0,1,1,1,0,1,0,0,0,1,1,1,0,1,0,1,1,1,0,1,0,
        1,0,0,1,1,0,1,0,0,0,1,0,1,1,1,0,1,0,1,0,1,0,1,0,
        1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,0,1,0,1,0,1,0,1,0,
        0,1,1,0,1,1,1,0,1,0,1,1,1,0,0,1,0,1,0,0,1,1,1,0,
        1,0,0,1,1,0,1,1,1,0,0,0,0,0,1,0,1,0,1,0,1,0,1,0,
        1,0,1,0,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1,0,0,0,1,1,
        0,0,1,0,0,1,1,1,1,0,1,0,1,0,1,0,1,0,0,0,1,1,1,0,
        1,0,1,0,1,0,1,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1,0,
        1,1,1,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,0,
        0,0,0,0,1,1,0,0,1,0,1,1,1,0,1,0,0,0,1,0,1,0,1,1,
        1,0,1,0,1,0,1,1,1,0,1,0,0,1,1,0,1,1,0,1,1,0,1,0,
        0,0,0,0,1,1,1,1,0,1,1,0,1,0,1,1,0,0,1,1,1,1,1,1,
        1,0,1,0,0,1,1,1,1,0,1,0,1,0,0,0,1,0,0,0,1,0,1,0,
        1,0,1,0,1,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,1,0,1,0,
        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
/********************************************************/

#endif /*PARAMETERS_H_*/
```

A.1.3 queue.h

```
#ifndef QUEUE_H_
#define QUEUE_H_

#include "parameters.h"

/*******Queue Structure*********/
struct queue_F{
        int front;
        int count;
        char *name;
        float buffer[MAXQUEUESIZE];
};
typedef struct queue_F queue_f;

/*******Push Element into Queue*************/
void push_Queue_f(queue_f *q, float x){
        if (q->count < (int)MAXQUEUESIZE){
                int index = (q->front + q->count)%MAXQUEUESIZE;
                q->buffer[index] = x;
                q->count = q->count + 1;
        }
        else{
                printf("ERROR: Buffer %s Overflow! \n",q->name);
        }
}

/*******Pop Element from Queue*************/
float pop_Queue_f(queue_f *q){
        if (q->count > 0){
                float v = q->buffer[q->front];
                q->front = (q->front+1)%MAXQUEUESIZE;
                q->count = q->count - 1;
                return v;
        }else{
                printf("ERROR: Buffer %s Empty! \n",q->name);
                return 0;
        }
}

#endif /*QUEUE_H_*/
```

A.1.4 encoder.h

```
#ifndef ENCODER_H_
#define ENCODER_H_

#include "parameters.h"
#include "queue.h"

/***LINEAR BLOCK ENCODING INITIALIZATION***/
void encoder_lb_init(void){
        //Initialize Generating Matrix
        int i, j;
        for (i = 0; i<ENCODER_LB_K; i++){
                for(j = 0; j<ENCODER_LB_N; j++){
                        if (j<ENCODER_LB_K){
                                if (i==j){
                                        gen_matrix[(i*(int)ENCODER_LB_N)+j] = 1;
                                        }
                                else {
                                        gen_matrix[(i*(int)ENCODER_LB_N)+j] = 0;
                                        }
                        }
                        else{
                                gen_matrix[(i*(int)ENCODER_LB_N)+j] =
                                        code_matrix[(i*(int)ENCODER_LB_N_K)+(j-
                                            ENCODER_LB_K)];
                        }
                }
        }
}

/***LINEAR BLOCK ENCODING OPERATION***/
void encoder_lb (float *data, int *data_ptr, queue_f *encoder_queue){
        int i;
        float k_data_buffer[ENCODER_LB_K];
        float n_data_buffer[ENCODER_LB_N];

        for (i = 0; i<ENCODER_LB_K; i++){
            k_data_buffer[i] = data[*data_ptr];
            *data_ptr=(*data_ptr+1)%BUFLENGTH;
        }

        DSPF_sp_mat_mul (k_data_buffer, 1, (int)ENCODER_LB_K, gen_matrix, (int)
            ENCODER_LB_N, n_data_buffer); //DSPLIB
        for (i = 0; i<ENCODER_LB_N; i++){
            push_Queue_f(encoder_queue,(float)(((int)n_data_buffer[i])%2));
        }
}

/***CONVOLUTIONAL ENCODING OPERATION***/
void encoder_cv (float *data, int *data_ptr, queue_f *encoder_queue){
        int i;
        float k_data_buffer[ENCODER_CV_K];
        float p_data_buffer[ENCODER_CV_P];

        for (i = 0; i<ENCODER_CV_K; i++){
            if ((*data_ptr)-(((int)ENCODER_CV_K)-1)+i >= 0){
                k_data_buffer[i] = data[(*data_ptr)-(((int)ENCODER_CV_K)-1)+
                    i];
            }else{
                k_data_buffer[i] = 0;
            }
        }

        DSPF_sp_mat_mul (gen_poly_matrix, ENCODER_CV_P, ENCODER_CV_K, k_data_buffer
            , 1, p_data_buffer); //DSPLIB
        for (i = 0; i<ENCODER_CV_P; i++){
            push_Queue_f(encoder_queue,(float)(((int)p_data_buffer[i])%2));
        }
        *data_ptr=(*data_ptr+1)%BUFLENGTH;
}

#endif /*ENCODER_H_*/
```

A.1.5 mapper.h

```
#ifndef MAPPER_H_
#define MAPPER_H_

#include "parameters.h"
#include "queue.h"
#include <math.h>

/*******************DEFINE CONSTANTS**********************/
#define SQ2 1.4142135623
#define q4a SQ2/2
#define q16a SQ2/6
#define q16b SQ2/2
#define q64a SQ2/14
#define q64b (3*SQ2)/14
#define q64c (5*SQ2)/14
#define q64d SQ2/2

float BPSK_const[2][2] = {{1,0},{-1,0}};
float BPSK_encoding[2][1] = {{1},{0}};
float QAM4_const[4][2] = {{q4a,q4a},{-q4a,q4a},{-q4a,-q4a},{q4a,-q4a}};
float QAM4_encoding[4][2] = {{0,0},{1,0},{0,1},{1,1}};
float QAM16_const[16][2] = {{q16a,q16a},{q16b,q16a},{q16a,q16b},
       {q16b,q16b},{q16a,-q16a},{q16a,-q16b},{q16b,-q16a},
       {q16b,-q16b},{-q16a,-q16a},{-q16a,q16b},{-q16b,q16a},
       {-q16b,q16b},{-q16a,-q16a},{-q16b,-q16a},{-q16a,-q16b},
       {-q16b,-q16b}};
float QAM16_encoding[16][4] =
    {{0,0,0,0},{0,0,0,1},{0,0,1,0},{0,0,1,1},{0,1,0,0},{0,1,0,1},
       {0,1,1,0},{0,1,1,1},{1,0,0,0},{1,0,0,1},{1,0,1,0},{1,0,1,1},{1,1,0,0},{1,1,0,1},

       {1,1,1,0},{1,1,1,1}};
float QAM64_const[64][2] = {{q64a,q64a},{q64b,q64a},{q64a,q64b},{q64b,q64b},
       {q64d,q64a},{q64c,q64a},{q64d,q64b},{q64c,q64b},{q64a,q64d},
       {q64b,q64d},{q64a,q64c},{q64b,q64c},{q64d,q64d},{q64c,q64d},
        {q64d,q64c},{q64c,q64c},{q64a,-q64a},{q64a,-q64b},{q64b,-q64a},
        {q64b,-q64b},{q64a,-q64d},{q64a,-q64c},{q64b,-q64d},{q64b,-q64c},
        {q64d,-q64a},{q64d,-q64b},{q64c,-q64a},{q64c,-q64b},{q64d,-q64d},
        {q64d,-q64c},{q64c,-q64d},{q64c,-q64c},{-q64a,q64a},{-q64a,q64b},
        {-q64b,q64a},{-q64b,q64b},{-q64a,q64d},{-q64a,q64c},{-q64b,q64d},
        {-q64b,q64c},{-q64d,q64a},{-q64d,q64b},{-q64c,q64a},{-q64c,q64b},
        {-q64d,q64d},{-q64d,q64c},{-q64c,q64d},{-q64c,q64c},{-q64a,-q64a},
        {-q64b,-q64a},{-q64a,-q64b},{-q64b,-q64b},{-q64d,-q64a},{-q64c,-q64a},
        {-q64d,-q64b},{-q64c,-q64b},{-q64a,-q64d},{-q64b,-q64d},{-q64a,-q64c},
        {-q64b,-q64c},{-q64d,-q64d},{-q64c,-q64d},{-q64d,-q64c},{-q64c,-q64c}};
float QAM64_encoding[64][6] =
    {{0,0,0,0,0,0},{0,0,0,0,0,1},{0,0,0,0,1,0},{0,0,0,0,1,1},
        {0,0,0,1,0,0},{0,0,0,1,0,1},{0,0,0,1,1,0},{0,0,0,1,1,1},{0,0,1,0,0,0},
        {0,0,1,0,0,1},{0,0,1,0,1,0},{0,0,1,0,1,1},{0,0,1,1,0,0},{0,0,1,1,0,1},
        {0,0,1,1,1,0},{0,0,1,1,1,1},{0,1,0,0,0,0},{0,1,0,0,0,1},{0,1,0,0,1,0},
        {0,1,0,0,1,1},{0,1,0,1,0,0},{0,1,0,1,0,1},{0,1,0,1,1,0},{0,1,0,1,1,1},
        {0,1,1,0,0,0},{0,1,1,0,0,1},{0,1,1,0,1,0},{0,1,1,0,1,1},{0,1,1,1,0,0},
        {0,1,1,1,0,1},{0,1,1,1,1,0},{0,1,1,1,1,1},{1,0,0,0,0,0},{1,0,0,0,0,1},
        {1,0,0,0,1,0},{1,0,0,0,1,1},{1,0,0,1,0,0},{1,0,0,1,0,1},{1,0,0,1,1,0},
        {1,0,0,1,1,1},{1,0,1,0,0,0},{1,0,1,0,0,1},{1,0,1,0,1,0},{1,0,1,0,1,1},
        {1,0,1,1,0,0},{1,0,1,1,0,1},{1,0,1,1,1,0},{1,0,1,1,1,1},{1,1,0,0,0,0},
        {1,1,0,0,0,1},{1,1,0,0,1,0},{1,1,0,0,1,1},{1,1,0,1,0,0},{1,1,0,1,0,1},
        {1,1,0,1,1,0},{1,1,0,1,1,1},{1,1,1,0,0,0},{1,1,1,0,0,1},{1,1,1,0,1,0},
        {1,1,1,0,1,1},{1,1,1,1,0,0},{1,1,1,1,0,1},{1,1,1,1,1,0},{1,1,1,1,1,1}};
/************************************************************/

/***DEFINE BLOCK VARIABLES***/
float *mapper_const;
float *mapper_encoding;
int bits;
float block[64];
float diff[64];
/****************************/

/***MAPPER INITIALIZATION****/
void mapper_init(){
        int i;

        switch((int)QAM){

                case 2:

                        mapper_const = &BPSK_const[0][0];

                        mapper_encoding = &BPSK_encoding[0][0];

                        bits = 1;

                        break;

                case 4:

                        mapper_const = &QAM4_const[0][0];

                        mapper_encoding = &QAM4_encoding[0][0];

                        bits = 2;

                        break;

                case 16:

                        mapper_const = &QAM16_const[0][0];

                        mapper_encoding = &QAM16_encoding[0][0];

                        bits = 4;

                        break;

                case 64:

                        mapper_const = &QAM64_const[0][0];

                        mapper_encoding = &QAM64_encoding[0][0];

                        bits = 6;

                        break;

                default:

                        printf("ERROR: Please enter valid QAM parameter.\n");

                        exit(0);

                        break;

        }

}


/******MAPPER OPERATION******/

void mapper(queue_f *mapper_queue, queue_f *encoder_queue){

        int i;

        int j;


        for (i=0; i<bits; i++){

                block[i] = pop_Queue_f(encoder_queue);
        }

        for (i=0; i<(int)QAM; i++){
                if (bits > 1){
                        DSPF_sp_w_vec (block, &mapper_encoding[i*bits], -1, diff,
                            bits);
                }else{
                        diff[0] = block[0] - mapper_encoding[i];
                }
                if (DSPF_sp_vecsum_sq (diff, bits) == 0){
                        for (j=0; j<2; j++){
                                push_Queue_f(mapper_queue, (int)QAM_POWER*
                                    mapper_const[(i*2)+j]);
                        }
                        break;
                }
        }
}

#endif /*MAPPER_H_*/
```

A.1.6 ofdm.h

```
#ifndef OFDM_H_
#define OFDM_H_

#include "parameters.h"
#include "queue.h"
#include <math.h>

#define OFDM_2N 2*(int)OFDM_N

float w[OFDM_N];

/*****Generate Real and Imaginary Twiddle
```

```
Table of Size n/2 Complex Numbers*******/
gen_w_r2(float* w, int n)
{
        int i;
        float pi = 4.0*atan(1.0);
        float e = pi*2.0/n;
        for(i=0; i < ( n>>1 ); i++)
        {
                w[2*i] = cos(i*e);
                w[2*i+1] = sin(i*e);
        }
}

/********Bit Reversal************/
bit_rev(float* x, int n)
{
        int i, j, k;
        float rtemp, itemp;
        j = 0;
        for (i=1; i < (n-1); i++)
        {
                k = n >> 1;
                while(k <= j)
                {
                        j -= k;
                        k >>= 1;
                }
                j += k;
                if(i < j)
                {
                        rtemp = x[j*2];
                        x[j*2] = x[i*2];
                        x[i*2] = rtemp;
                        itemp = x[j*2+1];
                        x[j*2+1] = x[i*2+1];
                        x[i*2+1] = itemp;
                        }
                }
        }

/***OFDM INITIALIZATION****/
void ofdm_init(){
        gen_w_r2(w, OFDM_N); //Generate coefficient table
        bit_rev(w, OFDM_N>>1); //Bit Reversal required for use with DSPLIB routine
}

/***OFDM OPERATION****/
void ofdm(queue_f *ofdm_queue, queue_f *mapper_queue){
        int i;
        float ofdm_buffer[OFDM_2N];
        for (i=0; i<OFDM_2N; i++){
                ofdm_buffer[i] = pop_Queue_f(mapper_queue);
        }

        //iFFT
        bit_rev(ofdm_buffer, OFDM_N);
        DSPF_sp_icfftr2_dif(ofdm_buffer, w, OFDM_N);

        //First push cyclic prefix
        //(1/N) factor because of inverse FFT
        for (i=0; i<2*OFDM_CP; i++){
                push_Queue_f(ofdm_queue, (1.0/OFDM_N)*ofdm_buffer[OFDM_2N - (2*
                    OFDM_CP) + i]);
        }
        //Then push FFT chunk
        //(1/N) factor because of inverse FFT
        for (i=0; i<OFDM_2N; i++){
                push_Queue_f(ofdm_queue, (1.0/OFDM_N)*ofdm_buffer[i]);
        }
}

#endif /*OFDM_H_*/
```

A.2 RX Chain Code

A.2.1 rx_process.c

```
#include <stdio.h>
#include <c6x.h>
#include <csl_types.h>
#include <cslr_dspintc.h>
#include <soc_OMAPL138.h>
#include <cslr_tmr.h>
#include <cslr_syscfg0_OMAPL138.h>
#include <math.h>
#include "parameters.h"
#include "queue.h"
#include "iofdm.h"
#include "demapper.h"
#include "decoder.h"
#include "input_samples.h"

void Initialize_Buffers(void);
void Process(void);
void Decoder(void);
void Demapper(void);
void iOFDM(void);

/******Initialize Queues******/
queue_f decoder_queue = {0,0,"Decoder_Queue"};
queue_f demapper_queue = {0,0,"Demapper_Queue"};
queue_f iofdm_queue = {0,0,"iOFDM_Queue"};

int inp_buf_ptr = 0;
float channel[OFDM_2N];

/*********MAIN FUNCTION**************/
void main (void)
{
        Initialize_Buffers();

    while(1){
        Process();
    }
}

/*********INITIALIZATION*************/
void Initialize_Buffers(void)
{
        int i;
        int j;
        int n;

        /***Get Input Samples***/
        for (i = 0; i<INPUTSAMPLESLENGTH; i++){
                input[2*i] = input_samples_real[i];
                input[(2*i)+1] = input_samples_imag[i];
        }

        decoder_lb_init();
        decoder_cv_init();
        demapper_init();
        iofdm_init(channel);
}

/*********PROCESS*************/
void Process(void){
        if ((ENCODER_V==0 && demapper_queue.count>=ENCODERN) ||
                (ENCODER_V==1 && demapper_queue.count >= ((int)ENCODER_CV_TRIALS*(
                    int)ENCODER_CV_P))){
                Decoder();
        }else{
                if (iofdm_queue.count >= 2){
                        Demapper();
                }else{
                    iOFDM();
                }
            }
        }

/*********DECODER BLOCK*************/
void Decoder(void)
{
        if((int)ENCODER_V == 0){
                decoder_lb (&demapper_queue, &decoder_queue, rec_data, &rec_data_ptr
                    );
        }else{
                decoder_cv (&demapper_queue, &decoder_queue, rec_data, &rec_data_ptr
                    );
        }
}

/*********DE-MAPPER BLOCK*************/
void Demapper(void)
{
        demapper(&iofdm_queue, &demapper_queue);
}

/*********iOFDM BLOCK*************/
void iOFDM(void)
{
        iofdm(&inp_buf_ptr, &iofdm_queue, channel);
}
```

A.2.2 parameters.h

```
#ifndef PARAMETERS_H_
#define PARAMETERS_H_

#include "input_samples.h"
```

```
#include <math.h>
#include "dsplib674x.h"

/*****************USER INTERFACE*****************/
#define ENCODER_V 0          //0 for LB, 1 for CV
#define ENCODER_LB_N 48
#define ENCODER_LB_K 24
#define ENCODER_CV_K 3
#define ENCODER_CV_P 2
#define ENCODER_CV_TRIALS 8
#define QAM 64
#define OFDM_N 128
#define OFDM_CP 32
/************************************************/

#define ENCODER_LB_N_K (((int)ENCODER_LB_N)-((int)ENCODER_LB_K))
#define ENCODER_LB_CODE_SIZE (int)ENCODER_LB_K*(int)ENCODER_LB_N_K
#define ENCODER_LB_ICODE_SIZE ((int)ENCODERN_K*(int)ENCODERK)
#define ENCODER_LB_PAR_CHECK_SIZE ((int)ENCODERN_K*(int)ENCODERN)
#define ENCODER_LB_ERROR_SIZE ((int)ENCODERN*(int)ENCODERN)
#define ENCODER_LB_SYND_SIZE ((int)ENCODERN*(int)ENCODERN_K)
#define ENCODER_CV_STATES_BITS 2
#define ENCODER_CV_STATES 4
#define MAXQUEUESIZE 4000
#define OFDM_2N 2*(int)OFDM_N
#define PI 3.14159265358979

float input[INPUTLENGTH];

/*****************USER INTERFACE*****************/
int gen_poly_matrix[ENCODER_CV_P][ENCODER_CV_K] = {{0,1,1},{1,1,1}};
float code_matrix[ENCODER_LB_CODE_SIZE]
    ={1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,
       1,0,1,0,1,0,1,1,0,0,1,0,0,1,1,0,1,0,0,0,0,1,1,0,
       0,0,0,0,1,0,1,1,1,0,0,0,1,1,1,0,1,0,1,0,0,0,1,1,
       0,1,1,0,1,1,1,1,0,0,1,0,1,0,1,0,1,0,1,0,0,0,1,0,
        1,0,0,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,0,1,0,1,1,
        1,0,1,0,1,0,0,0,0,0,1,0,1,1,0,0,1,0,1,0,1,0,1,0,
        1,1,1,0,0,1,1,0,1,0,1,0,1,0,0,0,1,0,1,0,1,0,1,0,
        0,0,1,0,1,0,1,0,0,0,1,1,1,0,1,0,1,1,0,0,0,0,1,1,
        1,0,0,0,1,0,1,1,1,0,1,0,0,1,1,0,1,0,1,0,1,1,1,0,
        0,0,1,0,1,1,1,0,1,0,0,0,1,1,1,0,1,0,1,1,1,0,1,0,
        1,0,0,1,1,0,1,0,0,0,1,0,1,1,1,0,1,0,1,0,1,0,1,0,
        1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,0,1,0,1,0,1,0,1,0,
        0,1,1,0,1,1,1,0,1,0,1,1,1,0,0,1,0,1,0,0,1,1,1,0,
        1,0,0,1,1,0,1,1,1,0,0,0,0,0,1,0,1,0,1,0,1,0,1,0,
        1,0,1,0,0,0,0,1,1,0,1,0,1,1,1,0,1,1,1,0,0,0,1,1,
        0,0,1,0,0,1,1,1,1,0,1,0,1,0,1,0,1,0,0,0,1,1,1,0,
        1,0,1,0,1,0,1,0,0,0,1,0,1,1,1,1,1,0,1,0,1,0,1,0,
        1,1,1,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,1,0,
        0,0,0,0,1,1,0,0,1,0,1,1,1,0,1,0,0,0,1,0,1,0,1,1,
        1,0,1,0,1,0,1,1,1,0,1,0,0,1,1,0,1,1,0,1,1,0,1,0,
        0,0,0,0,1,1,1,1,0,1,1,0,1,0,1,1,0,0,1,1,1,1,1,1,
        1,0,1,0,0,1,1,1,1,0,1,0,1,0,0,0,1,0,0,0,1,0,1,0,
        1,0,1,0,1,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,1,0,1,0,
        1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1};
/************************************************/

#endif /*PARAMETERS_H_*/
```

A.2.3 queue.h

Same as TX chain.

A.2.4 decoder.h

```
#ifndef DECODER_H_
#define DECODER_H_

#include "parameters.h"
#include "queue.h"
```

```
struct pred
    {
    int states[2];
    int states_output_parity[2][ENCODER_CV_P];
    int states_output_bit[2];
    };
typedef struct pred PREDECESSORS;

int CV_STATES_4[4][2] = {{0,0},{0,1},{1,0},{1,1}};
float icode_matrix[ENCODER_LB_ICODE_SIZE];
float par_check_matrix[ENCODER_LB_PAR_CHECK_SIZE];
float error_matrix[ENCODER_LB_ERROR_SIZE];
float syndrome_matrix[ENCODER_LB_SYND_SIZE];
float syndrome_matrix_T[ENCODER_LB_SYND_SIZE];
PREDECESSORS preds[ENCODER_CV_STATES];

/***SYNDROME DECODING INITIALIZATION***/
void decoder_lb_init(void){
        int i;
        int j;
        int e;
        float sb;

        //Initialize Parity Check Matrix and Syndromes
        DSPF_sp_mat_trans (code_matrix, (int)ENCODER_LB_K, (int)ENCODER_LB_N_K,
            icode_matrix);

        //Then compute parity check matrix
        for (i = 0; i<(int)ENCODER_LB_N_K; i++){
                for(j = 0; j<(int)ENCODER_LB_N; j++){
                        if (j<(int)ENCODER_LB_K){
                                par_check_matrix[(i*(int)ENCODER_LB_N)+j] =
                                    icode_matrix[(i*(int)ENCODER_LB_K)+j];
                        }
                        else{
```

```
                        if (i+((int)ENCODER_LB_K)==j){
                                par_check_matrix[(i*(int)ENCODER_LB_N)+j] = 1;
                        }
                        else {
                                par_check_matrix[(i*(int)ENCODER_LB_N)+j] = 0;
                        }
                }
        }
    }

    //Find Syndromes
    for (i = 0; i<(int)ENCODER_LB_N; i++){
        for (j = 0; j<(int)ENCODER_LB_N; j++){
            if (i==j){
                error_matrix[(i*(int)ENCODER_LB_N)+j] = 1;
            }
            else{
                error_matrix[(i*(int)ENCODER_LB_N)+j] = 0;
            }
        }
    }
    for (e = 0; e<(int)ENCODER_LB_N; e++){
        for (i = 0; i<(int)ENCODER_LB_N_K; i++){
            sb = 0;
            for (j = 0; j<(int)ENCODER_LB_N; j++){
                sb = (int)((sb + (par_check_matrix[(i*(int)
                    ENCODER_LB_N)+j]*error_matrix[(e*(int)
                    ENCODER_LB_N)+j])))%2;
            }
            syndrome_matrix[(e*(int)ENCODER_LB_N_K)+i] = sb;
        }
    }
}

/***SYNDROME DECODING OPERATION***/
```

```
void decoder_lb (queue_f *demapper_queue, queue_f *decoder_queue, float *rec_data,
     int *rec_data_ptr){
       int i;
       int j;
       int e;
       float rec[ENCODER_LB_N];
       float syndrome[ENCODER_LB_N_K];
       float sb;
       int counter;
       int found_syndrome = 0;
       float synd_temp[ENCODER_LB_N_K];

       for (i = 0; i<(int)ENCODER_LB_N; i++){
               rec[i] = pop_Queue_f(demapper_queue);
       }

       DSPF_sp_mat_mul (par_check_matrix, (int)ENCODER_LB_N_K, (int)ENCODER_LB_N,
           rec, 1, syndrome);
       for (i=0; i<(int)ENCODER_LB_N_K; i++){
               syndrome[i] = (float)(((int)syndrome[i])%2);
       }

       if (DSPF_sp_vecsum_sq(syndrome, (int)ENCODER_LB_N_K) == 0){
               for (i = 0; i<(int)ENCODER_LB_K; i++){
                       push_Queue_f(decoder_queue,rec[i]);
               }
       }else{
               error_counter = error_counter + 1;
               for (e = 0; e<(int)ENCODER_LB_N; e++){
                       DSPF_sp_w_vec (syndrome, &syndrome_matrix[e*(int)
                           ENCODER_LB_N_K], -1, synd_temp, (int)ENCODER_LB_N_K);
                       if (DSPF_sp_vecsum_sq(synd_temp, (int)ENCODER_LB_N_K) == 0){
                               found_syndrome = 1;
                               for (j = 0; j<(int)ENCODER_LB_K; j++){
                                       push_Queue_f(decoder_queue,(float)((int)(rec[j
                                           ]+error_matrix[(e*(int)ENCODER_LB_N)+j])
```

```
                                                    %2));
                                        }
                                        break;
                                }
                        }
                        if (found_syndrome == 0){
                                for (j = 0; j<(int)ENCODER_LB_K; j++){
                                        push_Queue_f(decoder_queue,0);
                                }
                        }
                }
        }

/***VITERBI DECODING INITIALIZATION***/
void decoder_cv_init(void){
        int i;
        int j;
        int p;
        int s;
        int ob;
        int sb;
        int ns;
        int parity[ENCODER_CV_P];
        int new_state;
        int k_buf[ENCODER_CV_K];

        //Initialize preds
        for (s=0; s<ENCODER_CV_STATES; s++){
                        for(ob=0; ob<2; ob++){
                                //Find next state
                                ns = ob;
                                for (i=1; i<(int)ENCODER_CV_STATES_BITS; i++){
                                        ns = ns + (CV_STATES_4[s][i]*pow(2,((int)
                                            ENCODER_CV_STATES_BITS)-i));
                                }
```

```
                    preds[ns].states[count[ns]] = s;
                    preds[ns].states_output_bit[count[ns]] = ob;

                    for (sb=0; sb<ENCODER_CV_STATES_BITS; sb++){
                        k_buf[sb] = CV_STATES_4[s][sb];
                    }
                    k_buf[((int)ENCODER_CV_K)-1] = ob;

                    //Find outputed parity bits
                    for (p = 0; p<ENCODER_CV_P; p++){
                        preds[ns].states_output_parity[count[ns]][p] =
                            0;
                        for (j = 0; j<ENCODER_CV_K; j++){
                            preds[ns].states_output_parity[count[ns
                                ]][p] = (preds[ns].
                                states_output_parity[count[ns]][p]
                                + (gen_poly_matrix[p][j]*k_buf[j]))
                                %2;
                        }
                    }
                }
            }
}

/***VITERBI DECODING OPERATION***/
void decoder_cv (queue_f *demapper_queue, queue_f *decoder_queue, float *rec_data,
     int *rec_data_ptr){
        int Predecessor[ENCODER_CV_STATES][ENCODER_CV_TRIALS];
        int PM[ENCODER_CV_STATES][ENCODER_CV_TRIALS];
        int t;
        int i;
        int s;
        int p;
        int bm0;        //branch metric 0
        int bm1;        //branch metric 1
        int pm0;        //path metric 0
int pm1;        //path metric 1
int min_pm;
int best_state;
float rec_buf[ENCODER_CV_STATES_BITS];
float out_buf[ENCODER_CV_TRIALS];

for (t=0; t<ENCODER_CV_TRIALS; t++){
        for (i=0; i<ENCODER_CV_P; i++){
                rec_buf[i] = pop_Queue_f(demapper_queue);
        }
        for (s=0; s<ENCODER_CV_STATES; s++){
                //Compute branch metrics.
                bm0 = 0;
                for (p = 0; p<ENCODER_CV_P; p++){
                        bm0 = bm0 + (((int)rec_buf[p] + preds[s].
                            states_output_parity[0][p])%2);
                }
                bm1 = 0;
                for (p = 0; p<ENCODER_CV_P; p++){
                        bm1 = bm1 + (((int)rec_buf[p] + preds[s].
                            states_output_parity[1][p])%2);
                }

                if (t==0){
                        pm0 = bm0;
                        pm1 = bm1;
                }else{
                        pm0 = PM[preds[s].states[0]][t-1] + bm0;
                        pm1 = PM[preds[s].states[1]][t-1] + bm1;
                }

                //Find predecessor
                if (pm0<pm1){
                        Predecessor[s][t] = 0;
                        PM[s][t] = pm0;
                }else{
                        Predecessor[s][t] = 1;
                        PM[s][t] = pm1;
                    }
                }
            }

            min_pm = PM[0][((int)ENCODER_CV_TRIALS)-1];
            best_state = 0;
            for (s=1; s<ENCODER_CV_STATES; s++){
                    if (PM[s][((int)ENCODER_CV_TRIALS)-1] < min_pm){
                            min_pm = PM[s][((int)ENCODER_CV_TRIALS)-1];
                            best_state = s;
                    }
            }

            for (t=((int)ENCODER_CV_TRIALS)-1; t>=0; t--){
                    out_buf[t] = preds[best_state].states_output_bit[Predecessor[
                        best_state][t]];
                    best_state = preds[best_state].states[Predecessor[best_state][t]];
            }

            for(t=0; t<ENCODER_CV_TRIALS; t++){
                    push_Queue_f(decoder_queue,out_buf[t]);
            }
}

#endif /*DECODER_H_*/
```

A.2.5 demapper.h

```
#ifndef DEMAPPER_H_
#define DEMAPPER_H_

#include "parameters.h"
#include "queue.h"
```

```
#define SQ2 1.4142135623
#define q4a SQ2/2
#define q16a SQ2/6
#define q16b SQ2/2
#define q64a SQ2/14
#define q64b (3*SQ2)/14
#define q64c (5*SQ2)/14
#define q64d SQ2/2

float BPSK_const[2][2] = {{1,0},{-1,0}};
float BPSK_encoding[2][1] = {{1},{0}};
float QAM4_const[4][2] = {{q4a,q4a},{-q4a,q4a},{-q4a,-q4a},{q4a,-q4a}};
float QAM4_encoding[4][2] = {{0,0},{1,0},{0,1},{1,1}};
float QAM16_const[16][2] = {{q16a,q16a},{q16b,q16a},{q16a,q16b},
       {q16b,q16b},{q16a,-q16a},{q16a,-q16b},{q16b,-q16a},
       {q16b,-q16b},{-q16a,-q16a},{-q16a,q16b},{-q16b,q16a},
       {-q16b,q16b},{-q16a,-q16a},{-q16b,-q16a},{-q16a,-q16b},
       {-q16b,-q16b}};
float QAM16_encoding[16][4] =
    {{0,0,0,0},{0,0,0,1},{0,0,1,0},{0,0,1,1},{0,1,0,0},{0,1,0,1},
       {0,1,1,0},{0,1,1,1},{1,0,0,0},{1,0,0,1},{1,0,1,0},{1,0,1,1},{1,1,0,0},{1,1,0,1},

       {1,1,1,0},{1,1,1,1}};
float QAM64_const[64][2] = {{q64a,q64a},{q64b,q64a},{q64a,q64b},{q64b,q64b},
       {q64d,q64a},{q64c,q64a},{q64d,q64b},{q64c,q64b},{q64a,q64d},
       {q64b,q64d},{q64a,q64c},{q64b,q64c},{q64d,q64d},{q64c,q64d},
       {q64d,q64c},{q64c,q64c},{q64a,-q64a},{q64a,-q64b},{q64b,-q64a},
       {q64b,-q64b},{q64a,-q64d},{q64a,-q64c},{q64b,-q64d},{q64b,-q64c},
       {q64d,-q64a},{q64d,-q64b},{q64c,-q64a},{q64c,-q64b},{q64d,-q64d},
       {q64d,-q64c},{q64c,-q64d},{q64c,-q64c},{-q64a,q64a},{-q64a,q64b},
       {-q64b,q64a},{-q64b,q64b},{-q64a,q64d},{-q64a,q64c},{-q64b,q64d},
       {-q64b,q64c},{-q64d,q64a},{-q64d,q64b},{-q64c,q64a},{-q64c,q64b},
       {-q64d,q64d},{-q64d,q64c},{-q64c,q64d},{-q64c,q64c},{-q64a,-q64a},
       {-q64b,-q64a},{-q64a,-q64b},{-q64b,-q64b},{-q64d,-q64a},{-q64c,-q64a},
       {-q64d,-q64b},{-q64c,-q64b},{-q64a,-q64d},{-q64b,-q64d},{-q64a,-q64c},
        {-q64b,-q64c},{-q64d,-q64d},{-q64c,-q64d},{-q64d,-q64c},{-q64c,-q64c}};
float QAM64_encoding[64][6] =
    {{0,0,0,0,0,0},{0,0,0,0,0,1},{0,0,0,0,1,0},{0,0,0,0,1,1},
        {0,0,0,1,0,0},{0,0,0,1,0,1},{0,0,0,1,1,0},{0,0,0,1,1,1},{0,0,1,0,0,0},
        {0,0,1,0,0,1},{0,0,1,0,1,0},{0,0,1,0,1,1},{0,0,1,1,0,0},{0,0,1,1,0,1},
        {0,0,1,1,1,0},{0,0,1,1,1,1},{0,1,0,0,0,0},{0,1,0,0,0,1},{0,1,0,0,1,0},
        {0,1,0,0,1,1},{0,1,0,1,0,0},{0,1,0,1,0,1},{0,1,0,1,1,0},{0,1,0,1,1,1},
        {0,1,1,0,0,0},{0,1,1,0,0,1},{0,1,1,0,1,0},{0,1,1,0,1,1},{0,1,1,1,0,0},
        {0,1,1,1,0,1},{0,1,1,1,1,0},{0,1,1,1,1,1},{1,0,0,0,0,0},{1,0,0,0,0,1},
        {1,0,0,0,1,0},{1,0,0,0,1,1},{1,0,0,1,0,0},{1,0,0,1,0,1},{1,0,0,1,1,0},
        {1,0,0,1,1,1},{1,0,1,0,0,0},{1,0,1,0,0,1},{1,0,1,0,1,0},{1,0,1,0,1,1},
        {1,0,1,1,0,0},{1,0,1,1,0,1},{1,0,1,1,1,0},{1,0,1,1,1,1},{1,1,0,0,0,0},
        {1,1,0,0,0,1},{1,1,0,0,1,0},{1,1,0,0,1,1},{1,1,0,1,0,0},{1,1,0,1,0,1},
        {1,1,0,1,1,0},{1,1,0,1,1,1},{1,1,1,0,0,0},{1,1,1,0,0,1},{1,1,1,0,1,0},
        {1,1,1,0,1,1},{1,1,1,1,0,0},{1,1,1,1,0,1},{1,1,1,1,1,0},{1,1,1,1,1,1}};

/***DEFINE BLOCK VARIABLES***/
float *mapper_const;
float *mapper_encoding;
int bits;
float block[64];
float diff[64];
/****************************/

/***DE-MAPPER INITIALIZATION****/
void demapper_init(){
        int i;
        switch((int)QAM){
                case 2:
                        mapper_const = &BPSK_const[0][0];
                        mapper_encoding = &BPSK_encoding[0][0];
                        bits = 1;
                        break;
                case 4:
                        mapper_const = &QAM4_const[0][0];
                        mapper_encoding = &QAM4_encoding[0][0];
                        bits = 2;
                        break;
                case 16:
                        mapper_const = &QAM16_const[0][0];
                        mapper_encoding = &QAM16_encoding[0][0];
                        bits = 4;
                        break;
                case 64:
                        mapper_const = &QAM64_const[0][0];
                        mapper_encoding = &QAM64_encoding[0][0];
                        bits = 6;
                        break;
                default:
                        printf("ERROR: Please enter valid QAM parameter.\n");
                        exit(0);
                        break;
        }
}

/******DE-MAPPER OPERATION******/
void demapper(queue_f *iofdm_queue, queue_f *demapper_queue){
        int i;
        int j;
        float rec[2];
        float diff[2];
        float d_min;
        float d_temp;
        int QAM_const = 0;
        rec[0] = pop_Queue_f(iofdm_queue);
        rec[1] = pop_Queue_f(iofdm_queue);

        DSPF_sp_w_vec (rec, &mapper_const[0], -1, diff, 2);
        d_min = DSPF_sp_vecsum_sq (diff, 2);
        for (i = 1; i<(int)QAM; i++){
                DSPF_sp_w_vec (rec, &mapper_const[i*2], -1, diff, 2);
                d_temp = DSPF_sp_vecsum_sq (diff, 2);
                if (d_temp < d_min){
                        QAM_const = i;
                        d_min = d_temp;
                }
        }

        for (j = 0; j<bits; j++){
                push_Queue_f(demapper_queue,mapper_encoding[(QAM_const*bits)+j]);
        }

}

#endif /*DEMAPPER_H_*/
```

A.2.6 iofdm.h

```
#ifndef IOFDM_H_
#define IOFDM_H_

#include "parameters.h"
#include "queue.h"

float w[OFDM_N];

/*****Generate Real and Imaginary Twiddle
Table of Size n/2 Complex Numbers*******/
gen_w_r2(float* w, int n)
{
        int i;
        float pi = 4.0*atan(1.0);
        float e = pi*2.0/n;
        for(i=0; i < ( n>>1 ); i++)
        {
                w[2*i] = cos(i*e);
                w[2*i+1] = sin(i*e);
        }
}

/********Bit Reversal************/
bit_rev(float* x, int n)
{
        int i, j, k;
        float rtemp, itemp;
        j = 0;
        for (i=1; i < (n-1); i++)
        {
                k = n >> 1;
                while(k <= j)
                {
                        j -= k;
                        k >>= 1;
                }
                j += k;
                if(i < j)
                {
                        rtemp = x[j*2];
                        x[j*2] = x[i*2];
                        x[i*2] = rtemp;
                        itemp = x[j*2+1];
                        x[j*2+1] = x[i*2+1];
                        x[i*2+1] = itemp;
                        }
        }
}

/***iOFDM INITIALIZATION****/
void iofdm_init(float *channel){
        int i;

        gen_w_r2(w, OFDM_N); // Generate coefficient table
        bit_rev(w, OFDM_N>>1);
        //Initialize channel (arbitrary for now)
        for (i=0; i<OFDM_N; i++){
                channel[(2*i)] = 1;
                channel[(2*i)+1] = 0;
        }
}

/***iOFDM OPERATION****/
void iofdm(int *inp_buf_ptr, queue_f *iofdm_queue, float *channel){
        int i;
        float temp1, temp2, temp3;
        float iofdm_buffer[OFDM_2N];
        int cp = (int)OFDM_CP;
        int t;

        //Skip Cyclic Prefix
        t = ((*inp_buf_ptr)+(2*cp))%((int)INPUTLENGTH);
        *inp_buf_ptr = t;

        //Extract OFDM chunk
        for (i=0; i<OFDM_2N; i++){
                iofdm_buffer[i] = input[*inp_buf_ptr];
                *inp_buf_ptr = (*inp_buf_ptr+1)%((int)INPUTLENGTH);
        }

        //FFT
        DSPF_sp_cfftr2_dit(iofdm_buffer, w, OFDM_N);
        bit_rev(iofdm_buffer, OFDM_N);

        //Y=HX so divide Y by H to get X
        for (i=0; i<OFDM_N; i++){
                temp1 = iofdm_buffer[2*i]*channel[2*i] + iofdm_buffer[(2*i)+1]*
                    channel[(2*i)+1];
                temp2 = iofdm_buffer[(2*i)+1]*channel[2*i] - iofdm_buffer[2*i]*
                    channel[(2*i)+1];
```

```
            temp3 = DSPF_sp_vecsum_sq (&channel[2*i], 2);
            push_Queue_f(iofdm_queue, temp1/temp3); //real
            push_Queue_f(iofdm_queue, temp2/temp3); //imag
      }
}

#endif /*IOFDM_H_*/
```

Bibliography

[1] H. Balakrishnan. *Linear Block Codes: Encoding and Syndrome Decoding.* MIT 6.02 Lecture Notes, Feb. 2012.

[2] H. Balakrishnan. *Convolutional Codes: Construction and Encoding.* MIT 6.02 Lecture Notes, Oct. 2011.

[3] H. Balakrishnan. *Viterbi Decoding of Convolutional Codes.* MIT 6.02 Lecture Notes, Oct. 2011.

[4] M. Bansal, J. Mehlman, S. Katti and P. Levis. OpenRadio: A Programmable Wireless Dataplane. In *HotSDN*, 2012.

[5] Bluespec. *http://www.bluespec.com.* Bluespec Inc.

[6] USRP. `http://www.ettus.com`. Ettus Inc.

[7] FlexRadio. `http://www.flexradio.com`. FlexRadio Systems.

[8] R. Gallager. *Principles of digital communication.* Cambridge University Press, 2008.

[9] GENESIS. `http://www.genesisradio.com.au`. GenesisRadio.

[10] GNURadio Software Radio. `http://gnuradio.org/trac`.

[11] A. Goldsmith. *Wireless Communications.* Cambridge University Press, 2005.

[12] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proc. OSDI*, October 2010.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campusnetworks. In *SIGCOMM Comput. Commun. Rev.*, 38(2):6974, 2008.

[14] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ANCS*, Oct. 2010.

[15] A. Oppenheim. and R. Schafer. *Discrete-time Signal Processing.* Pearson, 2010.

[16] D. Reay. *Digital Signal Processing and Applications with the OMAP-L138 eXperimenter.* Hoboken, N.J.: Wiley, 2012.

[17] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Volker. SORA: High Performance Software Radio Using General Purpose Multi-core Processors. In *NSDI*, 2009.

[18] Texas Instruments, "OMAP-L138 C6-Integra DSP+ARM Processor", OMAP-L138 datasheet, Jun. 2009 [Revised Oct. 2011].

[19] Texas Instruments, "TMS320C67x DSP Library Programmers Reference Guide", TMS320C6000 DSP Library, Jan. 2010.

[20] D. Tse and P. Vishwanath. *Fundamentals of Wireless Communications.* Cambridge University Press, 2005.

[21] Vanu software radio. `http://www.vanu.com`. Vanu, Inc.

[22] Rice university wireless open-access research platform (WARP). `http://warp.rice.edu`.