A Real-Time Software Implementation of an OFDM Modem Suitable for Software Defined Radios

Antonio Lucio Cinquino

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science at
Concordia University
Montreal, Quebec, Canada

January 2004

Your file Votre référence
ISBN: 0-612-91013-X
Our file Notre référence
ISBN: 0-612-91013-X

ABSTRACT

A Real-Time Software Implementation of an OFDM Modem
Suitable for Software Defined Radios

Antonio Lucio Cinquino

This thesis describes a real-time, DSP implementation of a baseband OFDM system, which includes interleaving and forward error correction algorithms. Software modules representing discrete system blocks are created and sequentially called upon as needed. The software reconfigurable system is developed on a Texas Instruments TMS320C6201 Evaluation Module, which is based on a fixed-point processor. Different combinations of arithmetic precision and speed for the fixed-point operations were explored. The decisive combination was used to evaluate the accuracy of the system by plotting bit-error curves against an equivalent floating-point model.

System efficiency is employed from the design stage through the use of Look-Up tables as well as cyclic shift registers. Upon implementation, the system is built using settings optimized for speed. In addition, memory management is used to ensure that critical instructions are kept on the fast on-chip memory while different access times for retrieving data on external memory were evaluated for best results. Finally, each software module is profiled to investigate and remove unnecessary latency issues.

Processing efficiency is monitored by recording throughput results, which are used to calculate maximum bit-rates. Using these results, estimations are made for bit-rates achievable by higher performance processors. These estimations demonstrate that a highly flexible DSP platform can ultimately meet speed and flexibility requirements for Software Defined Radios.

ACKNOWLEDGEMENTS

I would like to express my immense gratitude to Dr. Yousef Shayan for giving me the chance to complete this thesis under his supervision. During the course of my studies I had the chance to get to know him quite well and very much appreciate his openness and genuine sincerity. I would like to thank him for his guidance, knowledgeable advice, and for always being available to lend an ear.

I want to thank my Mother for her unbounded love, infinite patience, and endless support. She has been my foundation during many difficult times in my life. I would have never finished if it were not for her.

And finally, I want to thank my sweetheart Grace whose love gives me the motivation and strength to never give up. Her warm smile is always enough to lift me my spirits. I am truly blessed and eternally grateful to have her in my life.

This Thesis is dedicated

in loving memory of

My Father and Sister, Maria-Franca

I hold you both forever dear in my heart

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

In recent years there has been explosive growth in the mobile communications industry. According to the International Telecommunication Union (ITU), the last 10 years has seen an increase from 34 million to 1.3 billion in mobile phone subscribers [1]. Research firm In-Stat/MDR predicts that by 2007, the subscription rate will reach 2 billion [2]. These statistics show that consumer and business demand for mobility has risen sharply, which indicates our growing dependency on wireless technology.

1.1 Brief Overview of Mobile Industry

In the early days of the mobile communications industry, first generation cellular systems were primarily used for voice communication [3]. The convenience of reaching anyone, anywhere, quickly increased the popularity of cellular telephones. As capacity limits drew near, the switch from analog to digital systems allowed for a considerable increase in capacity and cheaper service. This, in turn, gave rise to a massive jump in the number of mobile subscribers and therefore the ubiquity of the cellular telephone. Today, mobile phones are used for voice as well as data communication. Local wireless service providers offer messaging services, email and a connection to the Internet with limited browsing capabilities [3]. Data services are becoming increasingly important as businesses are starting to incorporate wireless technologies into their workforce in order to work more efficiently and have mobile access to important information. However given the inadequate processing capabilities of mobile devices as well as insufficient available bandwidth, the wireless data experience through a cellular communication system has not reached its full potential.

A threshold on capacity is drawing near for present day, second generation (2G and 2.5G) networks. The increase in subscription rate coupled with new data applications is causing the network crunch. This dilemma will become more pressing given the predicted subscriber growth rate as well as the need for higher bandwidth data services. Third generation (3G) networks promise to resolve the capacity issue while providing much faster transmission rates [3]. However, the slowdown in the technology industry has caused a major delay in the deployment of 3G networks. In addition data communications capabilities, although an improvement over previous networks, are still insufficient thus limiting once again the potential of mobile, wireless data communications. With the arrival of wireless Internet and videoconferencing, as well as multimedia mobile telephony, mobile communication systems have been given a lot of focus and R&D to deliver these bandwidth hungry applications.

1.2 Multipath Fading

A fundamental problem exists in building wireless networks that offer higher and higher bit-rates. The characteristics of the wireless channel are such that it adversely affects the transmitted signal because of multipath fading. Multipath fading is caused by reflections in the wireless environment such that a signal will interfere with a delayed version of itself [4]. A relationship exists such that, the higher the data rate the greater the deterioration of the signal due to multipath. A solution is needed such that wireless networks are able to lessen the effect of multipath fading in order to provide sufficiently high transmission rates to evoke the deployment of future wireless data applications.

1.3 Incompatibility of Wireless Networks

In addition to the problem of multipath fading, another factor limiting the development of high bandwidth wireless networks, is the interoperability of mobile devices across different networks either employing different platforms or different standards. Present day systems are primarily suited for one specific application. In the example above, cellular networks were designed to provide voice communications and have only recently been upgraded to handle slow-rate data services. Also, wireless local area networks (WLANs) were designed specifically to deliver short range, high-speed wireless data for applications such as the Internet. Therefore, the uniquely designed radio hardware makes it impossible for a mobile device conceived specifically for one cellular standard to work on another.

Today, a multi-mode cellular telephone needs specific hardware built into the phone in order to be used across the different air interface standards. For example a mobile phone built for a GSM network in North America, which uses 1900Mhz would not work on a European GSM network, which uses either 900Mhz or 1800Mhz unless the hardware to demodulate both frequency bands was built into the phone [5]. It goes without saying that the same GSM phone would not work on any CDMA network either. In the case of different wireless platforms a mobile phone would need additional hardware (either built in or as an add-on module) to take advantage of existing personal wireless networks such as Bluetooth or Wi-Fi (a.k.a. IEEE 802.11x).

These days, a typical scenario for a person wanting to satisfy all their communication needs may be to have a cellular telephone for mobile voice communication as well as slow speed data services, a personal digital assistant (PDA) with Wi-Fi built in for short range, high speed, wireless Internet, and a separate wireless handheld for instant corporate email access. The amount of hardware needed reveals a cumbersome and bulky solution to stay connected. As modern society becomes more data-centric the need will be for reconfigurable and highly flexible networks as well as mobile devices that work across a number of standards, interfaces, as well as execute many applications [6].

To recap the technical issues mentioned above: In the near future the mobile industry will be faced with delivering high speed wireless networks that provide faithful signal reception in a multipath environment in order to accommodate future high bandwidth data applications. In addition, a society growing increasingly dependant on wireless technology will create the need for an "all-in-one" device that can cater to the number of competing air-interface standards and different wireless platforms that exist.

1.4 OFDM as a solution to Multipath Fading

A possible solution to the multipath fading problem would be to investigate at the physical layer of a communication system. Within the physical layer, the type of modulation used has great bearing on the reliability of a wireless communication link. OFDM, which stands for Orthogonal Frequency Division Multiplexing, makes a suitable candidate as a modulation technique.

OFDM works by dividing a high rate datastream into several parallel datastreams and transmitting them on separate, orthogonal subcarriers [3]. By keeping orthogonality between subcarriers, they can be squeezed closer together than other multi-carrier systems without interference, thus making efficient use of the available frequency spectrum. Given that the data is sent in parallel, at a lower rate, it is less prone to multipath effects in the channel. In addition, when compared to a single-carrier system, a multi-carrier OFDM system involves less complexity to implement because it does not require equalization to combat delay spreads caused by multipath [4].

Its robustness to delay spreads, efficient use of the frequency spectrum, and relatively low implementation complexity compared to single carrier systems are all favourable properties [4]. Because of this, wireless technologies that are using OFDM are able to achieve very high speed at relatively lower costs. A perfect example would be IEEE 802.11x, WLAN standards that are being deployed as a solution for indoor mobile wireless networks. Airports, restaurants, and cafés have begun installing 'hotspots' to offer wireless services to their clientele.

Given the successful deployment of OFDM in other wireless networks, the mobile industry is seriously considering using OFDM as a standard in the fourth generation (4G) networks [7]. 4G networks are expected to deliver 10 to 100 Mbit/s transmission rates [8]. Networks such as 3G, which were primarily originated for circuit-voice and then adapted for data cannot as easily provide for applications needing high bandwidth. Instead, it is much more cost effective to provide OFDM based, 4G networks that can increase transmission speeds over 3G networks at lower prices. This makes OFDM well suited for high bandwidth applications such as the Internet or video conferencing in cellular networks [9].

1.5 Implementation of OFDM with digital signal processing

The principal of OFDM has been known for over 35 years. It was Bell Labs who patented the idea in 1966 [4]. However, implementation of parallel data streams relied on the use of multiple, analog modulators to generate the subcarriers. Using bulky and power hungry analog devices in such a technique would be impossible to realize in a small form factor like a mobile phone. It is only recently that processing power has become fast enough to implement OFDM systems both efficiently and practically. Implementation of OFDM systems today range from using application specific integrated circuits (ASICs), to field programmable arrays (FPGAs), to digital signal processors (DSPs) or a combination there of. Each digital signal processing technology listed above has its own benefits and disadvantages.

ASICs and FPGAs have the advantage of custom hardware designs that can greatly increase processing performance [10]. However, given the high cost of producing ASICs relative to other generic processors, the choice is increasingly unattractive in a mobile phone market where the cost of devices is dropping considerably. In addition, ASICs offer little or no programmability therefore it is much harder to design for expandability or upgrades in a particular product. FPGAs also perform very well, but overall, systems designed entirely of FPGAs may take up more board space than necessary given that specific hardware may need to be synthesized for certain processing tasks [11]. Also, although FPGAs are highly reconfigurable, the time it may take to synthesize hardware for a new algorithm may take an unacceptably long wait period.

DSPs are the most flexible of all present day processing technologies [12]. DSPs perform tasks by executing specifically coded software. High-level language code such as C or C++ representing different functions and applications can be easily compiled and run which makes the DSP platform highly scalable and modifiable [10]. This gives DSPs the advantage of catering to dynamic applications and different standards. A particular design is not bound to a unique hardware configuration therefore its capabilities are extended. Although resources are limited in a DSP architecture requiring many tasks to be completed in sequence, DSPs are being developed at ever increasing clock speeds [12]. With transistor technology becoming smaller and microprocessor computing becoming more efficient, DSPs can process increasingly complex tasks. In addition some DSP designs are including more and more parallelism in the form of on-chip accelerators, which can operate independently from the CPU to handle other tasks [13]. This makes DSPs a promising choice for use in future wireless applications.

1.6 Software Defined Radio

The issues involving the lack of interoperability between different radio types hinder the deployment of all-in-one devices that work equally well on any air interface standard [14]. As mentioned, present day, multi-mode devices need devoted hardware in order to work across different platforms. This increases the size of the device while also raising its cost, which counteracts the trend in the mobile industry of producing smaller and cheaper products. A true solution to bridge the divide amongst all radio types would be a highly scalable and reconfigurable wireless system.

Software Defined Radios (SDRs) are being researched as a solution to the aforementioned interoperability issues. SDR is expected by many to emerge as the dominant design in the commercial wireless marketplace [15]. Basically an SDR is defined as a device that can alter its performance characteristics by replacing its embedded software [16]. In other words, the platform is based on using generic hardware on both the transmitter and receiver end and the executing software determines radio functions. In this manner, low-level functions such as modulation type and frequency band selection can be determined via software. This allows for a highly expandable and reconfigurable architecture. A user can move across different wireless platforms and standards by simply downloading and installing the appropriate software module.

1.7 Literature Survey

The implementation of an OFDM system using commercially available DSPs has been explored in the past. In [17] the authors developed a wireless LAN system using the TI C6x platform. Using a flexible DSP based platform the project could be easily modified to comply with all three of the worlds' wireless LAN standards (HiperLAN/2, IEEE 802.11a and MMA HiSWANa) given their physical layer commonality. The paper describes a real-time DSP implementation for a software coded OFDM based system. A video based communication application is also developed to operate over the system. The DSP performance is evaluated and compared to a C++ simulation. The setup uses two personal computers, each containing a TMS320C6201 EVM (including a TMS320C6201 processor). Daughterboards were designed to perform transmit and receive tasks via a cable linking the two PC's. To evaluate system performance, a radio channel was modeled in the transmitter software however results were worse than expected therefore a simulated AWGN channel was used instead. The results show a sustained data rate of 1.7 Mb/s. A performance increase was estimated with the use of hardware acceleration in order to achieve a duty cycle better than 7%. The system bit error rate (BER) performance was fairly close the floating point simulation at lower signal-to-noise ratio values (e.g. 10 – 22 dB). However given dynamic range limitation of the DSP hardware the two curves diverge to a difference of as much as 2.5 dB at BER of 2×10^{-3} (i.e. 2 errors occurring in 1000 transmitted bits).

In [18], a coded OFDM system was developed also using the TMS3206201 processor for telemetry applications in the racing, automotive environment. Telemetry is basically the transmission of data containing instrument or sensor readings from a remote location. The automotive market, both racing and commercial, is seeing increasingly important uses for telemetry applications. The commercial market can use telemetry for remote diagnostics on a stalled vehicle whereas in the racing market, racecars can relay important statistics to their teammates to gain strategic advantages. The authors note the advantages of using DSPs related to size, weight and cost for telemetry applications in the racing environment. They also add that using DSPs reduce hardware complexity and easily manage possible reconfigurations. For example, tradeoffs such as BER performance, power consumption can easily be adjusted instantly with software. The amount of data that needs to be transmitted grows with the number of sensors on racecars. When communicating with a racecar wirelessly, a system has not only to contend with multipath propagation but Doppler shift effects as well. Coded OFDM was chosen as the modulation technique because of its greater resistance to these effects as well as its relatively high bandwidth that caters to the increasingly large data demands of telemetry. The numerical results gave a transmission rate of 256Kbit/s for baseband transmission. This was achieved while using less than half the CPU capacity. Field tests were performed on a commercial car as well as a fast sports car with good BER and no loss of data. The paper concluded that the performance obtained with a flexible, low cost and lightweight DSP platform can be an effective solution for telemetry applications.

In [8] an SDR prototype based on a multiprocessor architecture (MPA) was developed for a Japanese standard 2G mobile system called PHS, as well as the IEEE

802.11 Wireless LAN standard. The system is composed of a CPU (400MHz PowerPC), which handles higher layer protocols, four TMS3206201 DSPs, which handle all physical layer functions such as modulation and flexible-rate pre-/post processors for filtering tasks. The prototype, which consisted of two PC terminals communicating through a wireless link, was shown to successfully switch between these two standards using an over-the-air (OTA) download function. System reconfiguration was executed with a reconfiguration time of 10 seconds (8 seconds of which belong to rerouting in the FPGA). The paper demonstrates that SDR can be a viable solution to deliver cross platform systems for future mobile and private networks.

1.8 Motivation and Objectives

The motivation behind this research comes from the increasing popularity of SDR as a design platform for a reconfigurable and highly flexible architecture for wireless technologies. Studies have been made over a possible framework for reconfiguration of SDR equipment [16]. In addition a prototype SDR showing the successful switch between a WLAN and 2G architecture was implemented [8].

In order for these systems (including 4G mobile networks) to deliver reliable, high transmission rates it is predicted that they will incorporate OFDM as the modulation technique [9]. Standards bodies in America, Europe and abroad have helped turn OFDM into a world wide standard in the 5-Ghz band. OFDM is also widely used in other frequency bands for WLAN, Digital Video Broadcast (DVB), and Bluetooth, which demonstrate its wide acceptance thus far [4].

It was shown that OFDM has already been implemented using a DSP for applications such as telemetry [18] and wireless LANs [17]. The increase in processing power, and the benefit of a highly flexible platform creates an opportunity for DSPs to play a critical role within the implementation of OFDM in the development of these future systems.

Given the above statements this thesis will seek to establish the importance of a DSP platform in the design of future mobile networks based on Software Defined Radios [19]. It will stress the advantages of DSPs as a most likely solution to the high reconfigurability and flexibility requirements of SDRs. It will demonstrate the increased productivity achieved with DSPs through cost-effective development tools, code reuse, and a host of algorithm libraries. Performance issues will be addressed through maximizing the use of Look up Tables and cyclic registers to relieve the CPU of wasted cycles, software pipelining to maximize instruction parallelism, and memory management. The goal will be to measure the performance of an OFDM system implemented with a DSP and analyze its suitability for an SDR platform [19].

1.9 Organization of Thesis

This thesis will be divided into four main parts. Chapter 2 will detail the basics of OFDM and describe the theory of operation. Chapter 3 will begin with an introduction to the concept of a Software Defined Radio. This will be followed by a brief description of different signal processing technologies used to implement OFDM systems and evaluate their design suitability for Software Defined Radios. Chapter 4 will detail an OFDM system implemented with a Texas Instrument C6201 DSP processor. A description of each software module used to make up the system will be given. Chapter 5 will provide an analysis of the OFDM system based on the DSP platform. It will start with the optimization techniques used to maximize performance. System accuracy and performance results will also be presented. Performance increases using a more powerful processor are estimated. Finally, a summary and analysis of the results will be given.

2 OFDM Basics

OFDM is a multicarrier transmission technique that divides a single datastream into multiple lower rate datastreams to be sent off, in parallel, on a number of sub-carriers [20]. An interesting and fundamental feature of OFDM is that the sub-carriers are orthogonal. This avoids the use of a guard band and allows for the carriers to be placed very close together without them interfering with one another while still conserving bandwidth [4].

The use of lower rate streams increases the signal's tolerance against high delay spread values. In addition, dividing the datastream onto multiple sub-carriers with different frequencies augments its robustness to selective fading in the wireless channel. Single frequency systems may have the entire link lost due to narrowband interference however OFDM will have only part of the signal affected. The use of channel coding can help recover errors occurring on the affected sub-carriers.

OFDM has gained increasing popularity over the years and has been adopted as the transmission standard for indoor wireless systems using IEEE 802.11x and Europe's HiperLAN/2 [21]. Digital Audio and Video broadcast systems also use OFDM. The popularity of OFDM comes from certain key advantages summarized below [4]:

- OFDM is an efficient way to deal with multipath; for a given delay spread, the implementation complexity is significantly lower than that of a single carrier system with an equalizer.
- In relatively slow time-varying channels, it is possible to significantly enhance the capacity by adapting the data rate per subcarrier according to the signal-to-noise-ratio of that particular subcarrier.

- OFDM is robust against narrowband interference, because such interference affects only a small percentage of the subcarriers

The following sections will detail the concepts behind OFDM, which will explain why it has become the modulation format of choice for many wireless applications. The general concept will be described first, highlighting the major parts of a block diagram representing a basic OFDM transceiver. Subsequent sections of this chapter will detail certain blocks emphasizing key ideas and giving a more in-depth explanation.

2.1 Concept

The basic idea behind OFDM is to take a high rate datastream and to split it into a number of parallel, lower rate datastreams. The lower rate datastreams are transmitted simultaneously over separate subcarriers. If for example the original stream contains 32-bits transmitted in T seconds then every 4 bits would be transmitted in *T/8* seconds. However, in an OFDM system, if the datastream is divided into 8 parallel datastreams of 4-bits each (see Figure 1), then each parallel datastream would ***still*** take *T* seconds long. Therefore the duration of each bit has been extended. This extension in time is one of the principle properties against dispersion caused by multipath delay spread.

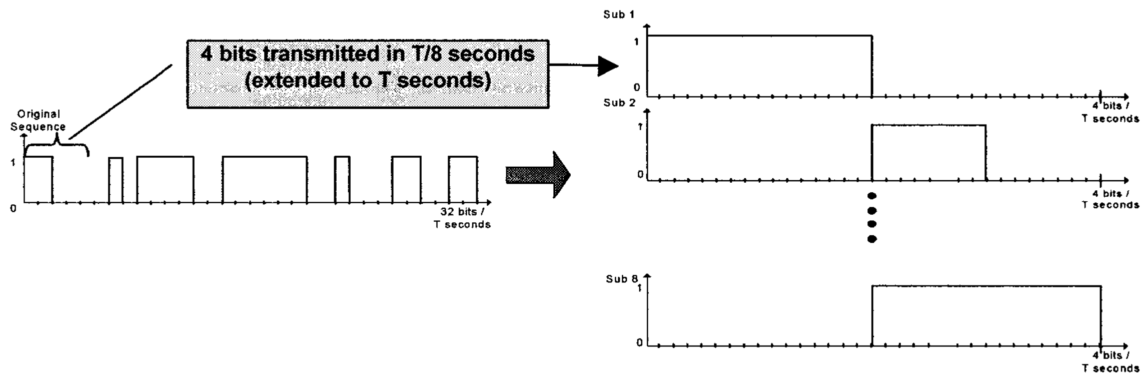


Figure 1: High Rate data stream divided into multiple low rate data streams

In using multiple sub-carriers, it is easy to think that more of the frequency spectrum is used to transmit the same amount of data as a single-carrier system. In fact, traditional multi-carrier systems (known as FDM systems) used adjacent sub-carriers that needed enough channel spacing (use of a guard band) to prevent overlap and hence avoid intercarrier interference (ICI) [4]. OFDM solves this problem by choosing adjacent subcarriers that are orthogonal. That is to say that the sub-carriers are related through a mathematical relationship. During demodulation of the OFDM signal, orthogonality of the subcarriers allows for the information encoded on each subcarrier to be extracted without interference from the adjacent subcarriers.

A basic OFDM system is shown in Figure 2. The system will take a stream of data bits (1's and 0's), channel code and interleave them to make the data more error resistant within the wireless channel. The bits are then mapped into complex signals using binary phase shift keying (BPSK), quadrature phase shift keying (QPSK), or quadrature amplitude modulation (QAM). Each complex signal now consists of a real and imaginary part. Pilots are also added to help the receiver synchronize the received data, which will have timing, and frequency offsets. The next step is to take a complex IFFT on the QAM input data. The IFFT operation is equivalent to modulating the complex data onto orthogonally generated subcarriers. The introduction of a guard time added before the final OFDM symbol helps to almost completely eliminate intersymbol interference (ISI). The baseband signal is then turned into an analog signal via a digital to analog converter (DAC). It is then upconverted to the desired transmitting frequency and sent off.

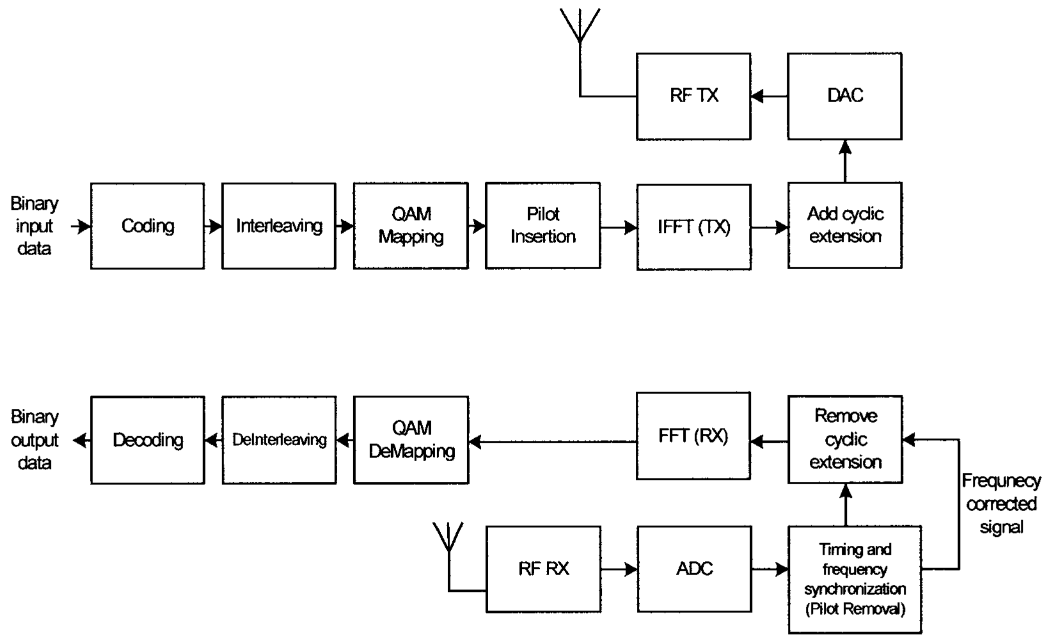


Figure 2: Block Diagram of an OFDM transceiver [4]

2.2 OFDM Symbol Generation

An OFDM symbol is the culmination of many operations performed in sequence as shown in the block diagram of Figure 2. Most operations prepare the data to combat against impairments caused by either the channel or system equipment and will be described shortly. However the main operation, considered the heart of an OFDM transmitter, is the generation of orthogonal subcarriers and modulation, which will be described first.

The generation of orthogonal carriers and modulation is performed through a mathematical operation called the Inverse Discrete Fourier Transform (IDFT) [22]. The Discrete Fourier Transform is performed on the receiver end to reverse the process. Most relate to the IDFT when considering that a complex sinusoid is formed through the summation of all its frequency components. By the same principal each OFDM signal is composed of the sum of subcarriers, which carry the QAM data. Eq. (2.1) mathematically describes an OFDM signal where d_i are the complex QAM symbols, N_s is the number of subcarriers, T is the symbol duration and f_c the carrier frequency [4].

$$s(t) = \mathrm{Re}\left\{ \sum_{i=-\frac{Ns}{2}}^{\frac{Ns}{2}-1} d_{i+Ns/2} \exp(j2\pi(f_c - \frac{i+0.5}{T})(t-t_s)) \right\}, \quad t_s \le t \le t_s + T \qquad (2.1)$$
$$s(t) = 0, \quad t < t_s \wedge t > t_s + T$$

Eq. (2.1) shows that the OFDM signal $s(t)$ is a time varying signal that is the result of the multiplication of complex QAM symbol with a complex exponential and summed over all subcarriers. The real and imaginary parts of the QAM symbol correspond to the in-phase and quadrature components respectively. This operation essentially describes the inverse Fourier transform of the QAM symbols, which act as the input frequency components. For implementation purposes with digital electronics, the variable t is replaced with a sample value n giving rise to time discrete equivalent expression in Eq. (2.2). This equation represents the Inverse Discrete Fourier Transform (IDFT) [4].

$$s(n) = \sum_{k=0}^{N_s-1} d_i \exp(j2\pi \frac{kn}{N}) \quad n = 0,1,....N-1 \qquad (2.2)$$

If the signal d_i is complex (i.e. consisting of real and imaginary parts), implementing Eq. (2.2), as is, would require N^2 complex multiplications for an N-point IDFT [23]. The value N is equivalent to the number of elements in the $s(n)$ sequence. When N becomes large, the complexity grows exponentially. The resulting operation would require an unacceptably long processing time for modern day applications. A practical algorithm titled, the Inverse Fast Fourier Transform (IFFT), efficiently implements Eq. (2.2) creating an enormous savings in computation time. The reduction comes from the realization that redundancy exists in the general IDFT equation. Specifically, the exponential portion shows complex conjugate symmetry as well as periodicity in *k* and *n* [24]. For practical systems the IFFT is used to generate the OFDM symbol and the FFT is used to recover the QAM data. The section below describes the IFFT/FFT algorithm in more detail.

2.2.1 IFFT and FFT Algorithm

The basic principal behind the Inverse Fast Fourier Transform (IFFT) and Fast Fourier Transform (FFT) algorithms is finding a method to more efficiently compute Eq. (2.3) and Eq. (2.4).

$$x[n] = \frac{1}{N}\sum_{n=0}^{N-1} X[k] W_N^{-kn}, \quad n = 0, 1, \ldots, N-1 \tag{2.3}$$

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \quad k = 0, 1, \ldots, N-1 \tag{2.4}$$

where $W_N = e^{-j(2\pi/N)}$. The reader should note that Eq. (2.3) and Eq. (2.2) are essentially equivalent. Since the IDFT and DFT equations differ in only the sign of the exponential portion and a scaling factor, the ideas presented in obtaining an algorithm for the FFT can be easily modified to fit the IFFT. Therefore, only the FFT algorithm will be described using Eq. (2.4).

A dramatic efficiency results from dividing the original time sequence *x[n]*, on the right-hand-side of Eq. (2.4) into successively smaller *x[n]*'s or reducing the frequency sequence *X[k]*, on the left-hand-side of Eq. (2.4) into successively smaller sequences and performing the DFT on either smaller sequence instead of the original [24]. Dividing the time sequence *x[n]* derives the Decimation in Time (DIT) FFT algorithm whereas dividing the frequency sequence *X[k]* derives the Decimation in Frequency (DIF) FFT algorithm. A DFT of each, smaller, sequence can still be taken because of the periodic properties of the exponential factor W_N. A simple substitution with a shorter index is all that is needed. Only the DIT algorithm will be explained further.

One essential criteria of the DIT algorithm is that the *N*-point sequence be a power of 2 (e.g. 2^v = 8-point sequence, where v = 3) [25]. The reason for this is so we can successively divide the sequence, *x[n]* in half and have an equal number of elements for each part. To summarize, if we had a sequence with *8* values and divided once, we would have two sequences of *4* values. Each 4-value sequence can be further divided into two 2-value sequences and so on. The more divisions we perform, the smaller the DFT's will be and hence fewer computations are required. However, this division operation cannot be done indefinitely as the sequence ultimately becomes an elementary operation and no longer a DFT (technically it is a 2-point DFT). Graphically this is termed a "butterfly" operation and is simply a 2-point transform where each of the two outputs is dependent on some combination operation of the inputs [23]. A butterfly flow graph is shown in Figure 3.

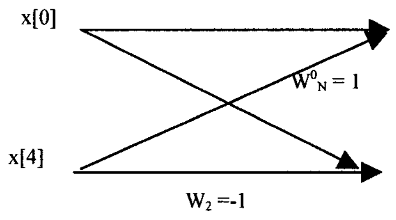
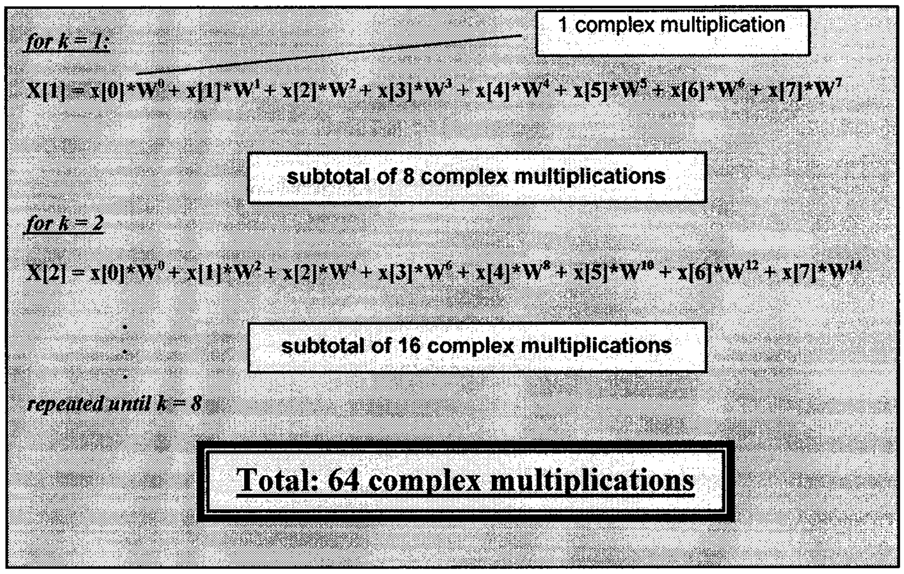


Figure 3: Flow graph of a 2-point DFT [24]

Using an 8-point DFT example (*N*=*8*) will help explain where the computational savings occur. We recall that an N-point DFT requires N^2 complex multiplications;

therefore 64 complex multiplications would be needed for our example. Figure 4 unrolls Eq. (2.4) for an 8-point DFT to demonstrate this.



for k = 1:

1 complex multiplication

X[1] = x[0]*W^0 + x[1]*W^1 + x[2]*W^2 + x[3]*W^3 + x[4]*W^4 + x[5]*W^5 + x[6]*W^6 + x[7]*W^7

subtotal of 8 complex multiplications

for k = 2

X[2] = x[0]*W^0 + x[1]*W^2 + x[2]*W^4 + x[3]*W^6 + x[4]*W^8 + x[5]*W^{10} + x[6]*W^{12} + x[7]*W^{14}

.
.
.

subtotal of 16 complex multiplications

repeated until k = 8

Total: 64 complex multiplications

Figure 4: Unrolling 8-point DFT [26]

The DIT algorithm starts by dividing the *x[n]* sequence into two sequences; one containing the even elements and the other with odd numbered points [24]. After the first division we will have an even-point N/2 sequence with *{x[0],x[2],x[4],x[6]}* as its elements and *{x[1],x[3],x[5],x[7]}* as the elements of the odd-point N/2 sequence. This same step is repeated over and over until each sequence will only have two elements per sequence. The pattern showing how the sequences are divided for an 8-point FFT are shown in Table 1.

Table 1: Input sequence x[n] after division into

Original Sequence	After Dividing into two N/2 sequences	After Dividing into four N/4 sequences
x[0] x[1] x[2] x[3] x[4] x[5] x[6] x[7]	{x[0], x[2], x[4], x[6]} {x[1], x[3], x[5], x[7]}	{x[0], x[4]} {x[2], x[6]} {x[1], x[5]} {x[3], x[7]}

We can demonstrate a reduction in computation even after the first division into two N/2 sequences simply by noting the number of computations needed for the DFT of each sequence. Our N-point DFT becomes two N/2-point DFT's as shown in Figure 5.

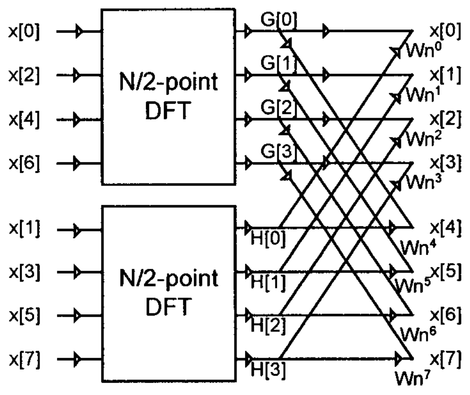


Figure 5: Flow graph for 8-point DFT [24]

Each DFT now requires *$(N/2)^2$ = 16* complex multiplications. Since there are two DFT's to compute, a total of *32* complex multiplications are needed which is half the original *64* complex multiplications. However an extra *N* multiplications are needed because of the extra multiplication by W_N^{kn} in the last stage of the flow graph bringing the total number of complex multiplications up to $N + 2(N/2)^2$ or *40* for this example. This represents a savings of *37.5%* over the original DFT after the first division. Further savings can be achieved by dividing the sequences again. If we continue until we are left only with butterfly operations, the number of complex multiplications reduces to $N/\log_2 N$ [25]. Table 2 compares the savings for a number of values of N.

Table 2: Comparison of DFT and FFT multiplication complexity [26]

N	DFT	FFT
32	1024	160
1024	1048576	10240
32768	$\sim 1 \times 10^9$	$\sim 0.5 \times 10^6$

The repeated transformation that takes place reduces all DFTs to a flow graph that requires only elementary operations as shown in Figure 6. The FFT algorithm simply looks at this flow graph and finds patterns to be replicated. In other words, the algorithm does not itself breakdown the sequence into smaller sequences. This was done once, by hand, and now we simply look at the final result and replicate that with an algorithm.

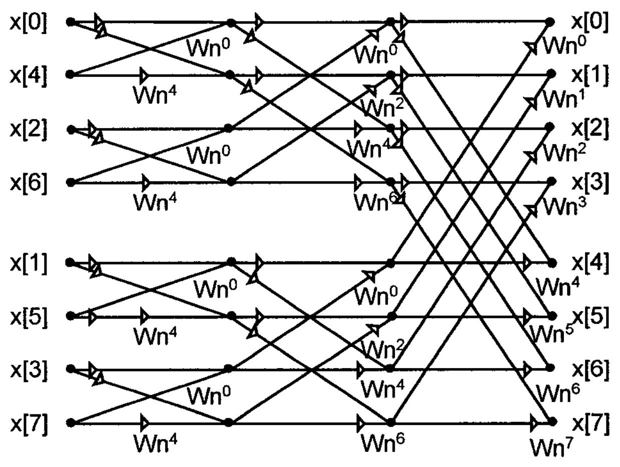


Figure 6: Reduced signal flow graph [24]

2.2.2 Bit Reversing

A basic relationship exists between the positions of the input and output sequences of the DIT and DIF algorithm. For the DIT, in order for the positions of each output element follow a linear order (top to bottom) then the positions of the inputs must have a corresponding bit reverse order as shown in Figure 7. The opposite is true for a DIF algorithm.

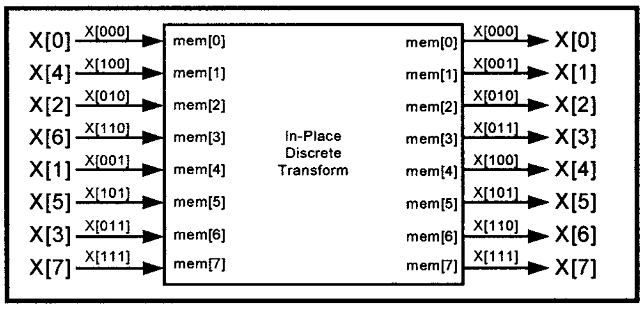


Figure 7: Bit Reversal of 8-Point DFT using DIT algorithm [27]

To explain bit reversing, it is convenient to represent the position number of the input and output sequences with binary digits. For example, for an 8-point FFT, *x[1]* can be represented as *x[001]* and *x[4]* as *x[100]*. Figure 7 shows that if memory location, *mem[1]* contains data in position *x[4]* at the input, then the output would have memory location *mem[1]* populated with data from memory location *x[1]*. Note that *x[4]* = *x[100]* has a binary value that is the reverse order of *x[1]* = *x[001]*. In general $x[b_2b_1b_0]$ has become *x[b0b1b2]*. The same applies to all input positions when compared to the output.

2.2.3 Orthogonality

As mentioned, the strength of an OFDM system lies in transforming a high rate datastream into multiple lower rate datastreams in order to achieve greater robustness to a multipath wireless channel. Spectral efficiency is maintained through the omission of a guard band between subcarriers. In order to transmit multiple datastreams and still conserve bandwidth, the subcarriers carrying each datastream need to be orthogonal to each other [4]. This allows subcarriers to lie close to one another within a given spectrum without interfering with one another (i.e. cause ICI). Figure 8 shows the spectrum of OFDM sub carriers. Note that the max of each sub carrier occurs at the zeroes of all other sub carriers.

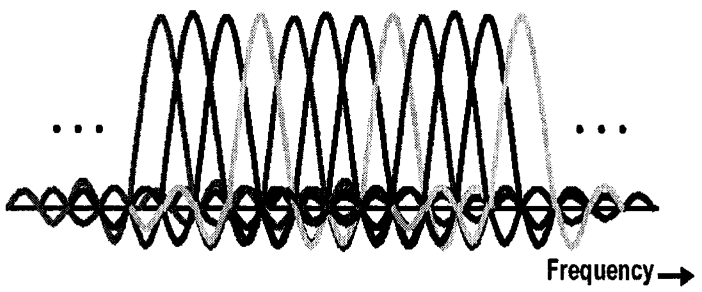


Figure 8: Spectrum of Orthogonal subcarriers [28]

The complex IDFT, by definition produces a complex time signal that is the summation of multiple sine and cosine carriers, which are orthogonal to each other, which is why the IDFT is used to modulate the complex QAM data. The IDFT operation creates sub carriers, which have an integer number of cycles within the OFDM symbol duration and are spaced out by exactly one integer cycle between adjacent sub carriers, which defines orthogonality [4].

2.3 Error Coding and Decoding

The first block in our basic OFDM system involves error coding. Error coding is used to improve communication performance within a communication system by reducing the effects of channel impairments such as noise, interference and fading [28]. This is accomplished by transforming the transmitted signal to give it certain properties that allow for the receiver to more easily distinguish it from other intended and non-intended signals.

At the baseband level, coding is achieved by developing structured sequences from the original, inputted, binary data. The structured sequence contains extra bits providing some redundancy to the original sequence [29]. The resulting datastream has better "distance properties". In other words, when comparing two sequences before and after coding is applied, the coded sequences look more "unalike" or have more differing bits than the unencoded sequence. The decoder in the receiver is then able to detect and/or correct a certain number of errors to retrieve the original data correctly.

The price paid for channel coding is bandwidth because of the added, redundant bits. However, the increase in bandwidth is a desirable tradeoff to either improved error performance or improved power efficiency depending on system criteria. So, in fact, channel coding can provide better system performance at less cost when compared to other methods such as higher power transmitters or the use of additional antennas [28].

Within an OFDM link, ISI is avoided by separating the datastream into a number of parallel streams transmitted on different carriers. However another problem is presented in that each of these subcarriers may arrive at different times and with different amplitudes caused by multipath fading. Some subcarriers may be more adversely affected than others, perhaps lost completely by deep fades. Therefore, these "weak" subcarriers largely determine the system error performance. To counteract this, channel coding is used across the carriers to correct some of the errors [4].

2.3.1 Convolutional Encoder.

A number of forward error correction (FEC) coding schemes exist, each having their own performance characteristics, implementation complexity, and coding gain. Convolutional codes are often chosen for many applications because they offer better performance for the same complexity implementation of other schemes [30].

Convolutional coding can be described in a number of ways. A convenient convolutional encoder representation is the connection representation shown in [28].

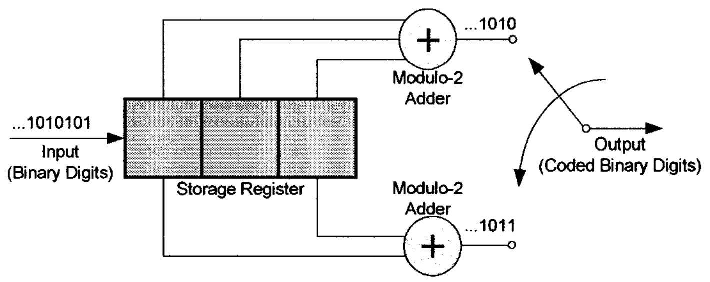


Figure 9: Connection representation of convolutional encoder [28]

The encoder works by passing a continuous stream of input bits, *n*, through a memory device such as a storage register [30]. The register length, referred to, as the constraint length is an important parameter as it has bearing on the distance properties of the resulting coded sequence. The constraint length of the register in the figure is $K = 3$. The number of output bits, *k*, for every input bit is determined by the amount of modulo-2 adders. In this example there are two thus making the code rate, *k/n*, equal to ½. Each modulo-2 adder is connected to specific cells of the shift register, which define the connection vector. Connection vectors of many codes have been tested to find those, which produce the best distance properties. The output bits are produced by the modulo-2 addition of only the cells selected by the connection vector.

Once the modulo-2 addition of the selected cells is completed, two output bits (one output from each modulo-2 adder) are produced. The shift register then inputs a new bit from the left-hand side and the right most bit is discarded. Therefore, each output bit is not only dependent on the present k input bits, but the past $K-1$ input bits as well [30].

2.3.2 Viterbi Decoder

A Viterbi decoding algorithm is used on the receiver end to decode the encoded bits from the convolutional encoder. The Viterbi decoding algorithm was discovered and analyzed by Viterbi in 1967 [28]. It is essentially based on the principal of *maximum likelihood decoding*. That is to say that the decoder will make a decision about the input sequence based on statistical knowledge of all possible input sequences. For example, for a convolutional encoder which has memory (i.e. the received bits are related to prior and present bits) the decoder will choose the most likely sequence that has the same pattern as the received sequence. Decision theory is described in greater detail in Appendix B of [28].

What makes the Viterbi algorithm attractive is that it performs the decision process with less computational complexity than a brute force approach. This is done by calculating a measure of distance or similarity, called a metric, between the received sequence and all other sequences at time t and eliminating all those that couldn't possibly be candidates. The process is then repeated from bit to bit until only the most likely candidate is left. Since only the "surviving" sequences are used, this removes the need for the received sequence to be compared with all other possible sequences saving unnecessary computations [31].

The trellis diagram is a manageable way to display the decoding procedure for a convolutional encoder. It provides the dimension of time to show all possible paths taken for each subsequently inputted bit. In addition, by exploiting the repetitive nature of the convolutional encoder after time t equal to the constraint length K, the number of branches that need to be drawn is kept relatively low. The number of rows in the trellis diagram is equal to the number of states taken from the last K-1 shift register elements. For $K= 3$, the number of states would be represented by $2^{K-1} = 4$ states. Figure 10 shows a trellis diagram of the received sequence ***U=11 01 01*** which was encoded using the specifications of the encoder in Figure 9. The trellis assumes that the convolutional encoder starts with zeroes in the storage register.

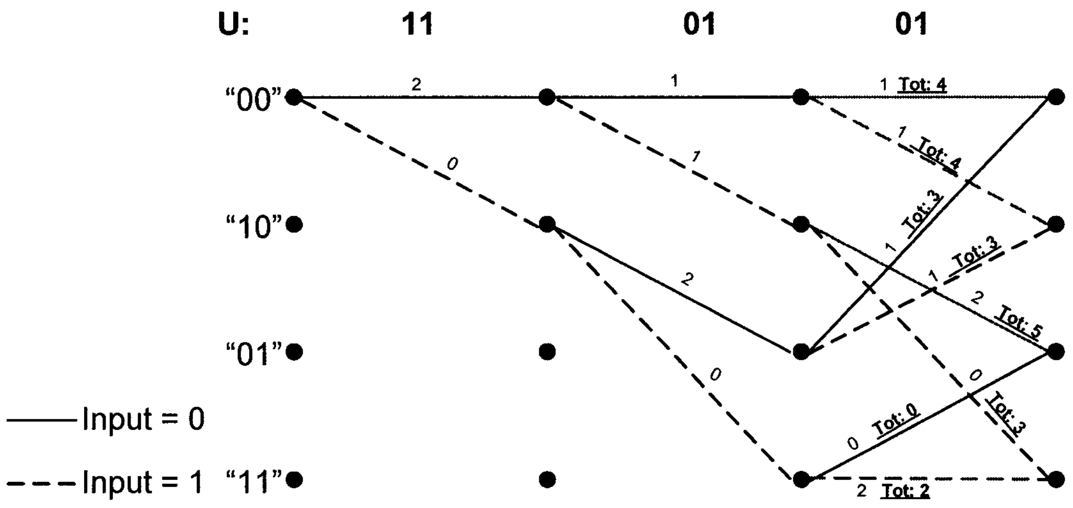


Figure 10: Trellis diagram of received sequence U = 11 01 01 [28]

As mentioned the Viterbi algorithm saves computation time by eliminating paths that are not likely candidates. This is performed at a node where two paths merge. The algorithm then tallies up all metric values accumulated at that merged point and discards the path with the higher metric value as shown by the red lines in Figure 10. Once the trellis grows large enough (usually several times the size of *K*), we can begin the traceback by choosing which of the four states has the lowest accumulated error. If we follow the path of the "survivor" back though the trellis we decode the most likely sequence, which can correct some errors that may have occurred in the channel [28].

2.3.3 Soft Decision Decoding

When a demodulator makes a decision between a binary "1" or "0", this is often called a hard-decision. However, quantizing the distance between these two values into higher levels is also possible and adds to system performance. Therefore, converting the received value to a quantized soft-value provides the decoder with more information. Instead of feeding the decoder a final decision, it gives a measure of confidence based on the quantized value. Figure 11 compares an 8-level soft decision and a 2-level hard decision using maximum likelihood probability distribution function plots. Note that a "000" is strong indication of a "0" and a "111" is a strong indication of "1".

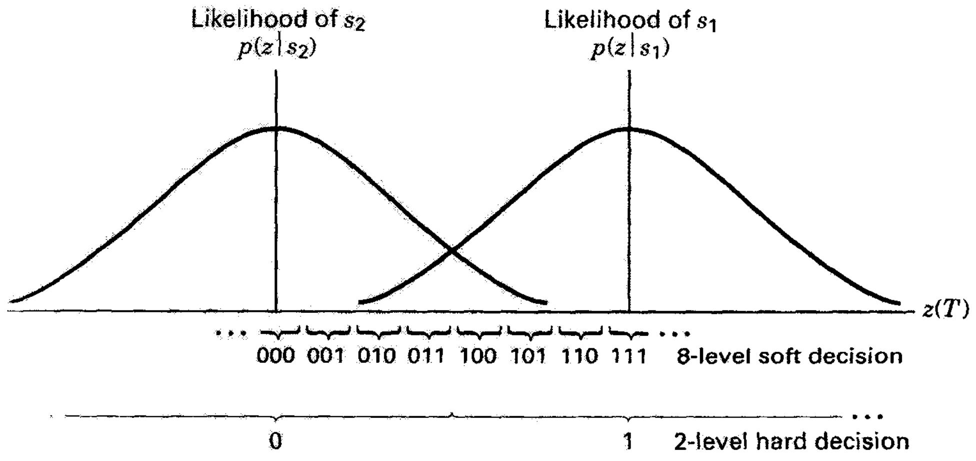


Figure 11: Hard and soft decoding decisions [28]

2.4 Interleaver and De-Interleaver

The second block in the transmitter portion of our OFDM system involves interleaving. An interleaver is a simple way of randomizing bits in a predetermined manner to help the recovery of information in the presence of burst errors. Some radio channels create deep fades in signal amplitudes at distinct frequencies resulting in frequency selective fading [4]. As a result certain subcarriers are more adversely affected than others, which cause groups of adjacent bits to be corrupted instead of errors occurring in a random fashion.

An interleaver spaces out adjacent bits so that when deep fades occur during transmission, errors actually occur at different symbols as shown in Figure 12. Therefore instead of, for example, four errors occurring in a row within a symbol, the errors might be dispersed amongst four symbols with one error in each. Most forward error correction (FEC) codes can handle one or two errors per symbol so the dispersion of errors by the interleaver aids in the recovery of data.

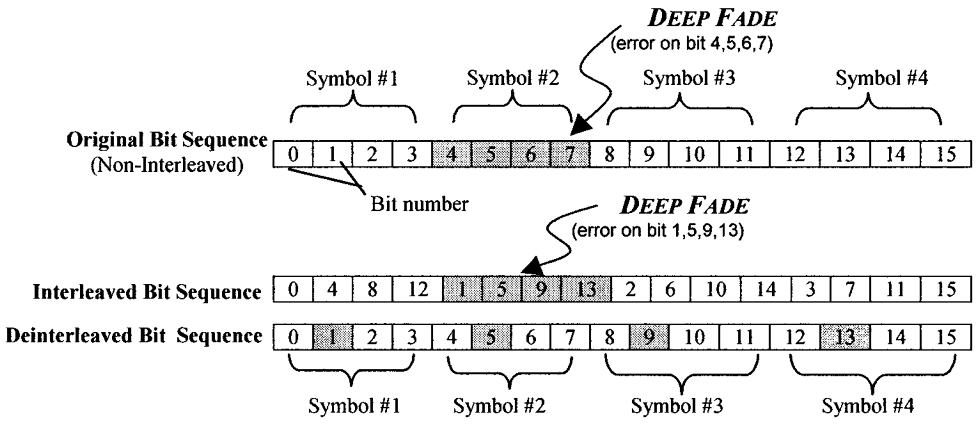


Figure 12: Affect of burst error on Non-Interleaved and Interleaved Data

A block interleaver is a common way of implementing interleaving. A block interleaver consists of a square matrix having dimensions equal to the length of the symbol or frame. The matrix acts as a sort of buffer where input symbols populate the columns of the matrix and once it is full, the data is read out row by row [4]. The reverse operation takes place on the receiver end to place the bits back in the original sequence before the decoder is applied. Figure 13 graphically describes block interleaving.

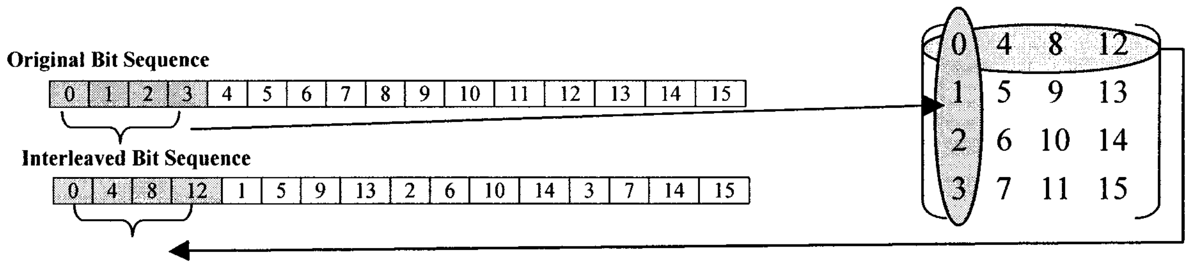


Figure 13: Block Interleaver

2.5 Quadrature Amplitude Modulation (QAM)

The block following the Interleaver block on the transmitter portion of our diagram has been titled "QAM" which stands for Quadrature Amplitude Modulation. The purpose of this block is to characterize the digital data into a format that will help decrease the bandwidth requirements [32]. QAM works well with OFDM and is most often chosen as the modulation format. The complex QAM signals, which represent frequency components act as inputs to the complex IFFT function. In practice, many OFDM systems normally choose between BPSK, QPSK, or QAM [33]. QAM has the advantage of offering the highest bit rate whereas the other formats would be used for fallback bit rates in noisier channels because of better distance properties. QAM can be considered as an extension of BPSK and QPSK therefore it will be the only one described.

2.5.1 Mapping

The concept of QAM is easily understood as a combination of amplitude shift keying (ASK) and phase shift keying (PSK) [34]. Within a QAM system, two independent values are created from the binary data such that one is amplitude modulated with the cosine of the carrier frequency and other with sine. Therefore QAM can be best described as two-dimensional signaling [32]. A constellation can be formed on a Cartesian plane where the x-axis represents the in-phase value and the y-axis represents the quadrature value. Figure 14 shows an example of a 16-QAM constellation used in the IEEE 802.11 standard.

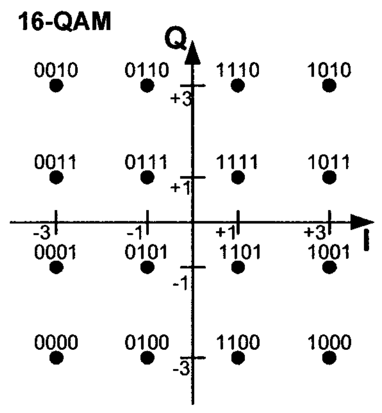


Figure 14: Constellation for 16-QAM [33]

Often, the QAM symbols making up the rectangular constellation are Gray encoded. The reason for Gray encoding is to provide further robustness against bit errors. This is done by ensuring that adjacent symbols have bits that differ in only one position. Consequently, if a symbol error should occur whereby the demodulator selects an adjacent symbol to that of the intended symbol, only one bit error will have occurred [32]. The constellation for the 16-QAM plot in Figure 14 shows Gray coding. Note how each group of 4 bits differs in only one bit location from the adjacent groups

The advantage of QAM over simply amplitude modulating the binary data is that it represents a method of reducing the bandwidth needed for data transmission. QAM belongs to the family of M-ary signaling whereby each symbol represents more than one bit. In this manner each symbol can transmit $k = log_2M$ bits. As an example each 16-

QAM (*M =16*) symbol represents four (*k = 4*) bits. The incoming bitstream is divided into groups of bits, in this case four bits for 16-QAM, and is mapped to the corresponding 16-QAM value taken from the constellation.

Higher QAM formats such as 64-QAM and 256-QAM represent further bandwidth decreases. However, under a fixed power constraint, the system would suffer from having QAM symbols spaced too close together [4]. This results in a larger signal to noise ratio (SNR) to achieve the same bit error rate (BER) for higher M-QAM values. Figure 15 below shows theoretical BER curves for various M-QAM values for an AWGN channel [35]. Note that a horizontal line drawn at a BER of 10^{-6} would show an SNR loss of about 4.5 dB between 16-QAM and 64-QAM.

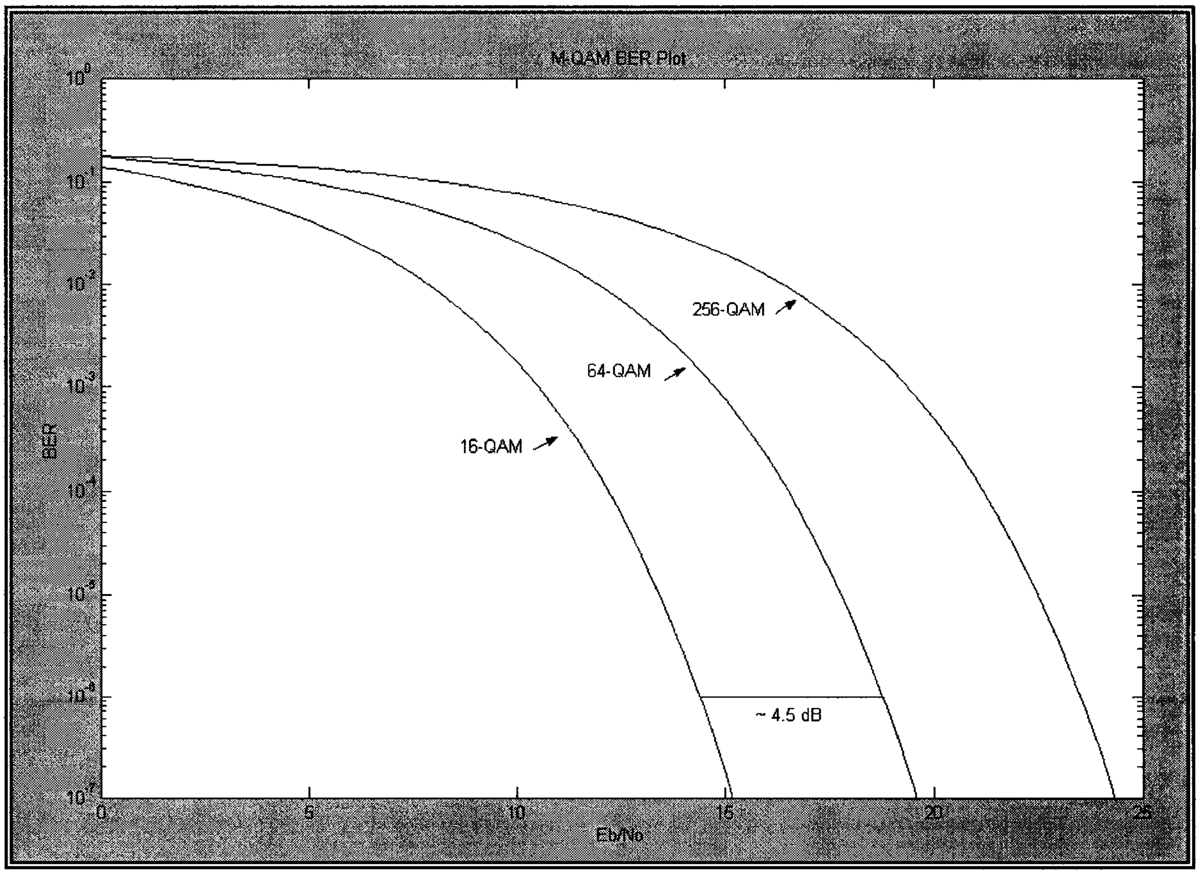


Figure 15: Various QAM BER curves

2.5.2 Demapping

Demapping of the QAM symbols takes place on the receiver end after the OFDM symbol recovers the QAM data from the FFT operation. Demapping is not as straight forward as the mapping operation because of impairments in the wireless channel that will cause the amplitude values of the cosine and sine functions to vary. Therefore, upon reception and demodulation of the analog waveform, the QAM symbols are not discrete values as shown in Figure 14. Instead, the existence of noise causes the constellation points to be spread as in Figure 16.

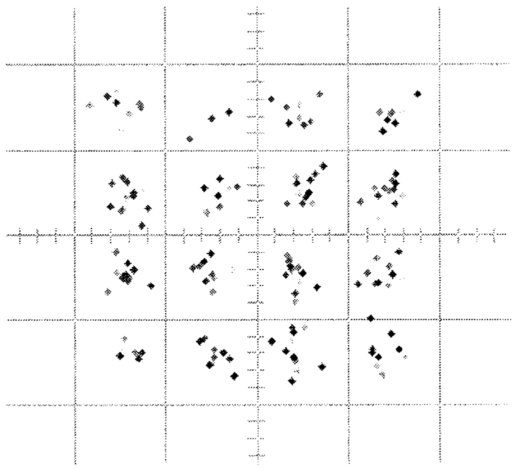


Figure 16: Scatter plot of 16-QAM constellation

Therefore the demapping procedure must establish "decision regions" in order to associate the received values with an actual QAM value. However errors may occur, such that the received value spread so far as to fall into an adjacent region. The error correction scheme used in the OFDM system is then responsible for correcting these errors.

2.6 Pilot Symbols

Before the QAM symbols are inputted to the IFFT block, an extra step is taken to ensure proper demodulation on the receiver end. Transmitting information on orthogonal carriers creates a strong point for OFDM by conserving bandwidth. However this same principal also creates a problem for OFDM transmission. OFDM is particularly sensitive to any frequency or timing offsets, which cause orthogonality to be lost [36]. For example, a frequency offset of any subcarrier would then cause the maximums of every other subcarrier to occur at a point which does not correspond to the zeros of other subcarrier thus increasing the chance of ICI. Frequency and timing offsets occur simply because there is no way of guaranteeing that the transmitter and receiver clocks are completely in sync.

Pilot insertion is performed on the transmitter side in order for synchronization with the transmitted signal to be performed at the receiver. Pilots are symbols or subcarriers containing "extra" information, which the receiver has a priori knowledge of and can be used as a training sequence. They are used to correct timing and frequency offsets of the transmitted signal. These offsets introduce ISI and ICI in the signal, which would ultimately cause irrecoverable errors [36]. An OFDM symbol can have certain subcarriers reserved for use as pilots. On the receiver end, this information is extracted and used to determine symbol boundaries and optimal timing for demodulation.

2.7 Guard Time and Cyclic Extension

OFDM's robustness to multipath delay is one of its main advantages over other modulation formats. However, wireless channels still introduce multipath delays, which cause certain parts of the OFDM symbol to arrive later than others. When multipath components of one symbol arrive at the same time as another symbol, the energy of each can overlap and add together to introduce intersymbol interference (ISI). ISI can be almost completely eliminated with the proper use of a guard time [37]. Therefore, after the OFDM symbol is generated, a guard time is used to extend the duration of an OFDM symbol. This ensures that any subcarrier that arrives with delay will not interfere with those of another symbol. If the guard time is larger than the expected delay then ISI will not occur due to the delay spread [38]. The reason for this is to give enough time for all the multipath components to "die out" before demodulation of the adjacent symbol is started. The demodulation period is time over which the FFT interval is taken in order to retrieve the QAM symbols.

The format and value for the guard time cannot be arbitrarily chosen. For example extending the symbol duration with no signal would cause ICI because the FFT interval of certain delayed components of a symbol would then not contain an integer number of cycles as shown in Figure 17.

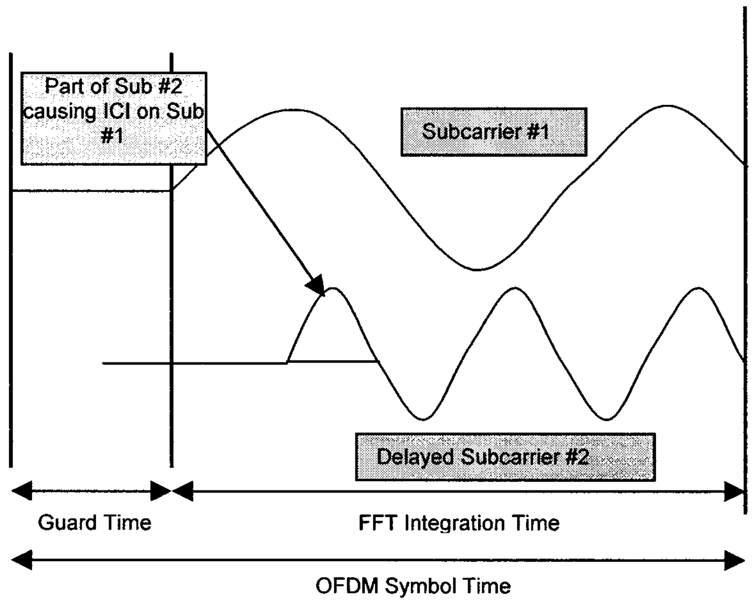


Figure 17: Effect of multipath with zero signal in the guard time [4]

During the FFT integration time, "Subcarrier 1" would be demodulated properly, as there is no delay, however intercarrier interference would be caused by "Subcarrier 2" because the OFDM symbol does not contain an integer number of cycles within the FFT integration time. To avoid ICI, the guard time should be composed of a cyclic extension of the OFDM symbol. In other words the guard time simply consists of a continuation of the signal. In this manner, orthogonality is maintained which is essential for the FFT to properly retrieve the QAM symbols on each subcarrier. As long as the delayed signal arrives within the length of the guard time, the FFT integration time will always consist of an integer number of cycles. This would eliminate ICI from “Subcarrier 1” onto “Subcarrier 2” and vice versa [4].

3 Signal Processing Technologies used in Software Defined Radios

With advancements in computing power, the trend in transceiver design has been to perform more and more tasks using digital signal processing technologies [14]. Certain radio modules traditionally reserved for analog devices are now being processed digitally. A perfect example was described in Chapter 2 where the IFFT operation replaces a bank of analog modulators to generate the subcarriers for an OFDM system. With this continuing trend, the term Software Defined Radio (SDR) has been coined in the wireless telecommunications industry [39]. SDRs offer multiple advantages in building networks that are very flexible and reduce the expense of bulky analog equipment that consumes more power. An ideal SDR architecture consists of a simple analog subsystem and a complex digital subsystem [40]. Specifically, within an SDR architecture, the baseband processing portion of the digital subsystem has most to gain from advancements in digital signal processing technology.

Many digital signal-processing technologies exist today to implement the baseband processing portion of digital subsystem in SDRs. However each solution may have certain advantages and disadvantages over others. The choices available run from custom hardware such as application specific integrated circuits (ASICs), to reconfigurable hardware such as field programmable gate arrays (FPGAs) to reprogrammable platforms such as digital signal processors (DSPs) and general-purpose processors (GPPs) [10].

Given the choices and the respective advantages that each hold makes it harder for a designer to pick the right technology. A designer may not only have performance issues to contend with. Time to market, scalability and adaptability are all other factors to consider. The general criteria that need to be addressed are outlined below [41]:

- **Programmability**: the ability to reconfigure a device to perform the desired functions for all of the target standards
- **Level of integration**: the ability to integrate several functions into a single device, thus reducing the size and hardware complexity of the system
- **Development cycle**: the time it takes to develop, implement, and test a function with a specific device
- **Performance**: the ability of a device to perform the function within the required time
- **Power**: the power utilization of the device when performing the required function

As mentioned in Chapter 1 the need for future networks that can reliably deliver high data transmission rates as well as offer interoperability and seamless switching between different wireless networks using different protocols and standards will become increasingly important. This makes the use of OFDM as a modulation format and the idea of an SDR concept ideal for future networks. The emergence of a single platform that can cater to all the criteria listed is still out of reach for present day designs. However, it is believed that programmable DSPs offer advantages over other signal processing technologies that create an opportunity to be a key technology in the development of future wireless networks that require a great amount of flexibility such as SDRs.

This chapter will begin by introducing the concept of Software Defined Radio (SDR) including a description of the proposed architecture and key components. Later sections will detail the different digital signal processing technologies that can be potentially used to help implement SDRs. Advantages and disadvantages of each will be given with special attention on the use of programmable DSPs.

3.1 Software Defined Radio

Historically, radio design in early cellular systems such as the first generation (1G) networks traded bandwidth for system complexity resulting in relatively low system capacity. As the industry moved to second generation (2G), digital networks, more baseband processing was performed in order to increase the number of users on a single carrier. For example, both TDMA and CDMA multiplex a number of users in the digital part of the system at baseband prior to the modulator [40]. Third generation (3G) and fourth generation (4G) networks, require even more processing power to provide increased system capacity. With more processing taking place in the digital part of the system, less equipment is needed in the analog subsystem. This is one aspect in forming the concept of an SDR however easy reconfiguration of the digital subsystem is another. The SDR Forum, a non-profit organization created to help develop and promote SDR technologies uses the following definition [39].

> "Software Defined Radio (SDR) is a collection of hardware and software technologies that enable reconfigurable system architectures for wireless

networks and user terminals. SDR provides an efficient and comparatively inexpensive solution to the problem of building multi-mode, multi-band, multi-functional wireless devices that can be enhanced using software upgrades. As such, SDR can really be considered an enabling technology that is applicable across a wide range of areas within the wireless industry.“

The general idea is to design devices such as handhelds, and radio equipment included in the wireless network infrastructure that can be dynamically programmed and reprogrammed with software. The hardware goes unchanged yet it has the ability to be reconfigured to perform different functions at different times. For example, a software download could upgrade the deployed members from a simple air interface based on, QPSK, to an improved air interface with a new high data rate 16-QAM mode [10]. This provides a huge advantage over existing architectures and the benefits extend from consumers to wireless service providers. Examples are [15]:

- For subscribers- easier international roaming, improved and more flexible services, increased personalization and choice.
- For mobile network operators – the potential to rapidly develop and introduce new, personalized, and customized services; a tool for increased customer retention and added value services.
- For handset and base station manufacturers – the promise of new scale economies, increased production flexibility, and improved products .

The wireless industry has been increasing effort into developing SDR technologies and turning the idea into reality. The motivation comes from being able to deploy cost effective networks that are easily upgradeable with the possibility of little or no hardware redesign effort.

3.1.1 Basic Architecture

An ideal software radio architecture consists of a complex digital subsystem and a simple analog system. The goal is to limit the analog functions to those that can only be performed with analog components such as the antenna, RF filtering, RF combination, receive and transmit amplification. The digital subsystem has two major responsibilities [40]: 1) carrier separation and up/down frequency conversion to baseband and 2) all baseband processing such as error control coding and modulation functions. It should have the added characteristic that certain functions are reprogrammable depending on system application. Figure 18 shows an ideal software radio architecture including application software.

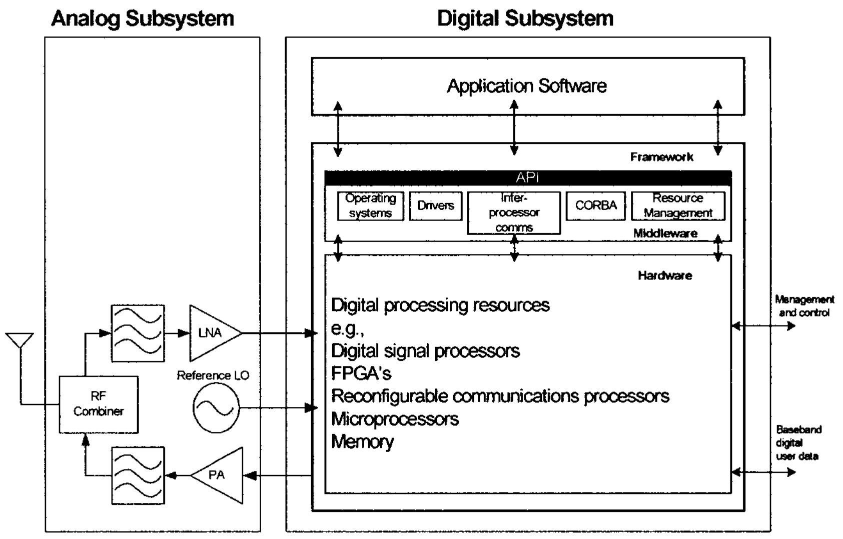


Figure 18: Ideal Software Radio Architecture [40]

Software for the ideal architecture is layered so that it is detached from the hardware level. It is the task of the middleware layer (e.g. operating system, resource management) to interface the application software and hardware layer. This removes low-level programming from application development and allows software designers to develop more complex and powerful applications [40].

As can be seen from Figure 18 the architecture of a software defined radio is composed of many components, however, only the hardware resources in the digital subsystem will be discussed further.

3.1.2 Digital Subsystem Hardware

The greatest advancements in technology have to take place in the digital subsystem side in order to make the concept of a cost effective SDR solution viable. We would ideally like to choose a technology that is powerful enough to interface with minimal analog equipment as well as be programmable enough to easily interface with high-level application software [40].

As mentioned above, the software defined radio digital subsystem consists of digital frequency conversion and baseband processing. The idea behind digital frequency conversion is to directly sample the operating frequency of the carrier. The challenge of interfacing with the analog portion is to be able to sample higher and higher frequencies therefore eliminating several down and up converters for intermediate frequency (IF) conversions [10]. However given that even a good wideband front-end receiver can take a 3G system carrier signal of 900MHz and convert it to a digital IF of 70 MHz, it would still take about 4300 MIPS (millions of instructions per second) to digitally sample and process. This requires a dedicated, high end DSP alone for just the task of frequency sampling and conversion to baseband. Therefore fundamental limitations prevent the use of digital processors to tackle the task of frequency conversion. It would be better left to digital up converters (DUCs) and digital down converters (DDCs) to perform frequency conversion much more efficiently and still have enough programming capability to be considered for SDR platform [40].

Of equal importance in system design within an SDR is the baseband processing portion of the digital subsystem. The design is directly dependant on the air interface standard it must cater to whether it is for the mobile industry or any other wireless communication system application. For example, different standards may implement different modulation schemes or error correction code techniques. An SDR platform will be able to use software in the application layer to dynamically change functions. Therefore the need will be for digital processing to be powerful enough to handle the higher transmission rates as well as provide the flexibility needed to quickly switch to different air-standards and applications [10].

Many digital signal-processing technologies exist today to implement this portion of an SDR platform. The following section details the most likely candidates.

3.2 Comparison of Signal Processing Technologies for Baseband Processing

Signal processing technologies that are candidates for SDR baseband processing range from application specific integrated circuits (ASIC's), to reconfigurable hardware such as field programmable gate arrays (FPGA's) to reprogrammable platforms such as digital signal processors (DSP's) and general-purpose processors (GPP's). There are trade offs with the digital processor solutions that are available. The following section will detail the different signal processing technologies available to implement the baseband digital subsystem portion of an SDR platform.

3.2.1 ASIC and GPP Solutions

We can place ASICs and GPPs on opposite ends of the selection spectrum [14]. GPPs are designed once and are produced in mass volume to offer cheap processing power. General-purpose processors such as those found in personal computers (PCs) are designed to run many different types of applications such as image processing, math intensive software, or recreational arcade games. Therefore the design is not intended to run any application extremely well as each type of application would have special requirements for efficiency. This results in GPP's consuming many more cycles than competing technologies to implement functions needed in an SDR. ASICs on the other hand are designed to run a particular application with great efficiency in both cycle and power consumption. However, ASICs are fixed designs that cannot be altered to perform anything other than the originally intended application. This directly contradicts the idea of an SDR that must be designed with the capability to be configured to run many different applications. The design time needed for producing an ASIC chip used for a single application raises the cost per unit giving ASICs a price disadvantage as well. Therefore these two technologies are not generally suited as a baseband processing design solution for an SDR platform.

3.2.2 FPGA Solutions

The benefit of using an FPGA solution over GPPs and DSPs is its potential for much greater performance. Hardware can be specifically synthesized to execute processing tasks within an application therefore performing them in parallel. However the number of hardware resources used as well as the maximum clock frequency is dependent on the efficiency of the synthesis tools used in the design. Therefore, actual clock frequency may be significantly less than the theoretical maximum [40]. Also, if hardware needs to be synthesized for each application, a disadvantage might result in having a system with an unnecessarily large amount of resources. This may increase power consumption from one hardware profile to the next making the power requirements for a battery powered handheld unreliable [14].

FPGAs are not as easily and efficiently programmable as DSPs. The same library of algorithms that exist for C programmed code is not available for code written in VHDL for FPGAs therefore making a product's time to market lengthier. When compared to DSPs this can decrease the cost effectiveness of using FPGAs over the design cycle [40].

A fundamental difference between DSPs and FPGA for use in SDRs is reconfiguration time. A DSP simply needs to run new code to change the function of the hardware, however FPGAs are not dynamically reconfigurable. A mobile device using an FPGA may need to come off-line in order for the FPGA to be reconfigured to change its function. This can take up to fractions of a second, which can contribute to many lost frames in a communication link resulting in unacceptable quality of service [40].

Given the above, FPGAs have still found their way into SDR designs because of their advantageous processing power over DSPs. However, their use will be limited to processing tasks that require intensive computational loads that need not dynamically change during a link call such as filtering and waveform shaping of the IF.

3.2.3 DSP solutions

The advantage of DSPs over GPPs is that they are designed to carry out specific instructions very efficiently [12]. One of the most common examples is that most DSPs can perform a multiply and accumulate (MAC) operation in one cycle time whereas GPPs normally require several cycles for the same task. This proves to be advantages in communications related signal processing where MACs are frequently and repetitively used in algorithm development. In addition, the hardware architecture of most recent DSPs is tailored to handle other communication-related processing more efficiently.

The advantage of DSPs over FPGAs is that they are easily and efficiently programmable [14]. Early DSPs required the knowledge of a different assembly language for each processor type. However today's DSPs allow programmers to code in a high level language such as C or C++ and have the compiler produce executable code. This allows for the production of high quality code that can be reused in other system design. It also helps to reduce the amount of development time needed in system design using a DSP. Another advantage to programmable DSPs is that they fully comply with the software requirement of the SDR concept making the design completely flexible [40]. Therefore, DSPs could easily handle changes to baseband processing tasks as the changes could simply be programmed in a high level language and executed when needed.

DSPs have a performance disadvantage when compared to FPGAs as the amount of operations needed for some of the processing tasks can outnumber the available resources therefore delaying the overall system. However, higher performance DSPs are widely available in the market today. The performance of DSPs has been increasing at a rate similar to Moore's Law, which predicts the doubling of computing power every 18 months [40]. For communication related processing a good measure of DSP performance is to consider how many million multiply and accumulate per second (MMACS) it can perform. Figure 19 shows a plot of MMACs over time for Texas Instrument processors.

Therefore, although DSPs may not compete with the processing power of FPGAs in certain applications, it has shown exponential increase over the years. It is this rate of increase that has pushed the concept of an SDR from paper to implementation. It can be predicted that this trend will continue for the next ten years thus potentially reaching a performance level high enough to handle the demanding baseband processing tasks of future SDR designs.

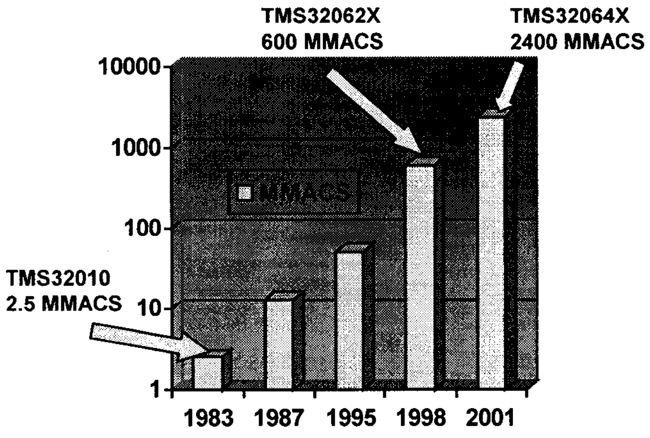


Figure 19: DSP performance increase. [40]

Performance disadvantages may be also addressed in other ways. The trend for some present day DSPs has been to include application specific coprocessors that handle computationally intensive tasks such as error correction coding, which would normally have been processed by the DSP core [13]. These coprocessors are designed with enough programmability to be used with different standards therefore keeping with the advantage of a flexible and programmable platform. In addition designers can place multiple DSPs on a board and use a divide and conquer approach to handle very computationally intense tasks. In summary the advantages of using DSPs are [42]:

- Reduced development time and quick prototyping. Quick time to market.
- Flexibility. It can quickly adapt to changing or different standards as it needs only a software change.

- Ideal for multi-mode Basebands where multiple standards are supported by the same device.

The above advantages make DSPs very attractive for future, mobile wireless systems where each system may need to cater to a multitude of standards and interfaces. Therefore, DSPs might be best suited to areas that need to be easily programmable to ensure high adaptability and make a prime candidate for use within an SDR platform.

The DSP platform chosen for the implementation presented in this thesis is provided by Texas Instruments. The development tools consist of a complete software suite with debugging capabilities as well as a fully functional evaluation module containing a high performance DSP and external memory. Further details are provided in the subsequent chapters.

4 Software Implementation of an OFDM System

The advantages of using DSP processors over other signal processing technologies were given in the previous chapter. What follows is a description of an OFDM, baseband system implemented using a Texas Instruments (TI) DSP platform. A brief introduction to the software suite and hardware used will be given. The software structure will be described while detailing the modular, code blocks that implement the system.

4.1 Software Platform

A host of DSP platforms exist on the commercial market. Companies such as Motorola, Analog Devices and Texas Instruments offer design kits, which include evaluation boards and accompanying design software. Of these, Texas Instruments (TI) offers the fastest and most powerful DSP processors under the C6000 platform [43]. In March of 2003 they announced the introduction of a 720MHz processor and two months later they unveiled an updated roadmap of the C6000 platform, which includes a 1.1GHz processor for mid-2004. Table 3 demonstrates a performance summary of competing platforms.

Table 3: Performance Comparison of DSP industry leaders [40]

Company	**Texas Instruments**	**Texas Instruments**	**Analog Devices**	**MSC8102**
Processor	***C6416 600 MHz***	***C6416 1100 MHz***	***TigerSHARC, 180 MHz***	***MSC8102 300 MHz***
8-bit MMACS	4800	8800	2880	--
16-bit MMACS	2400	4400	1440	4800
MIPS	4800	8800	1080	7200
Coprocessors	TCP (12604 MIPS)+ VCP (2581 MIPS)	TCP (23107 MIPS)+ VCP (4732 MIPS)	None	EFCOP (1200 MMACS)

TI offers a software suite titled Code Composer Studio which facilitates the design process by offering compile, testing and debugging features all under the same software tool. In addition, a DSP library containing many algorithms commonly used in industry are available for designers to use and include in custom designs [44].

The following describes the TI hardware and software used for the implementation of the OFDM system in greater detail.

4.1.1 TI Code Composer Studio

Code composer studio (CCS) conveniently ties the DSP hardware with the design of any project. It includes all the software tools needed to take a design project from beginning to end. It supports all phases of the development cycle shown in Figure 20.

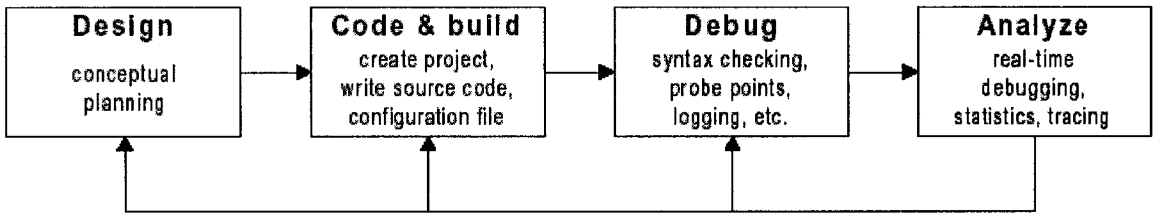

Design
conceptual planning
Code & build
create project, write source code, configuration file
Debug
syntax checking, probe points, logging, etc.
Analyze
real-time debugging, statistics, tracing

Figure 20: Phases of development cycle [45]

Once the conceptual planning of the design is finalized, the code generation tools included in the suite allow the designer to efficiently code and build the project. CCS provides the convenience of programming in a high level language such as C or C++ to ease the design flow. A typical software development flow using the code generation tools is shown in Figure 21. Path “A” defines a design flow using C code whereas Path “B” is typical of those using assembly code.

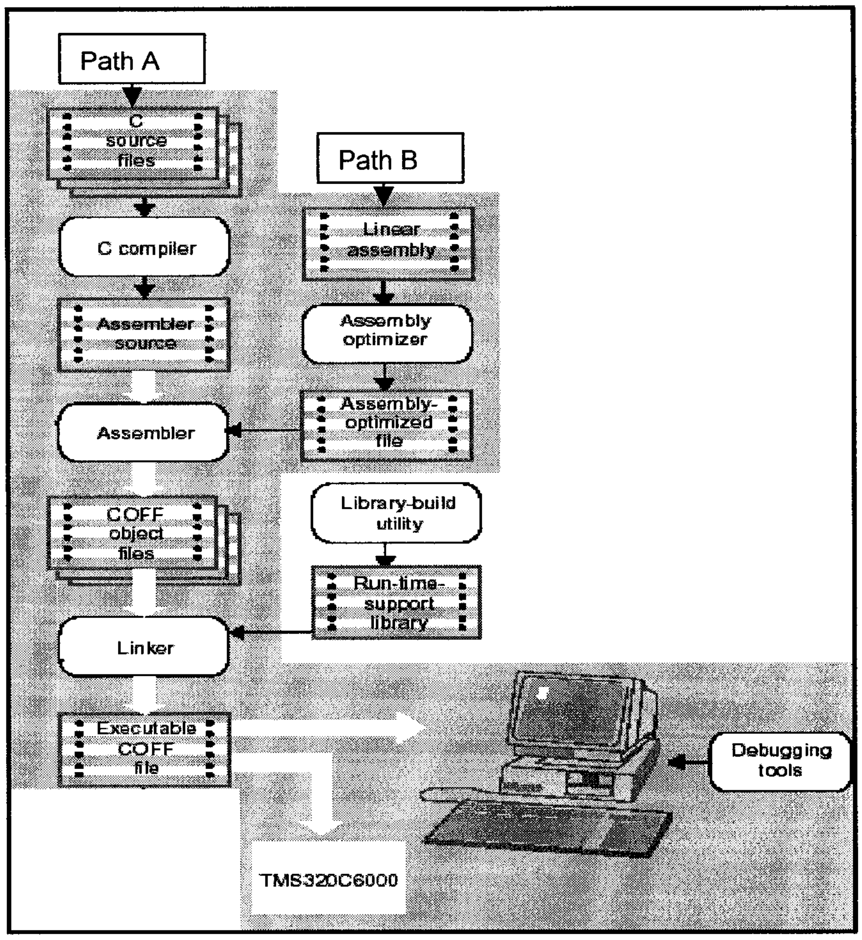


Figure 21: Software development flow [45]

The C compiler accepts C source code and produces assembly language source code. To reduce overall execution time, the compiler can operate in one of four optimization modes. For example the compiler can unravel loops to reduce the overall number of branches to speed code execution. A great amount of engineering has gone into producing a compiler that can rival the performance of assembly code, which has traditionally been far superior [40]. Table 4 demonstrates the efficiency of some algorithms compiled with CCS when compared to assembly.

Table 4: Comparison of CCS compiler assembly language programming [40]

Algorithm	Use	Assembly cycles	C Cycles	Percent Efficiency
Codebook Search	***CSELP vocoder***	977	976	102
10-Tap FIR Filter	***VSELP vocoder***	238	280	85
16-Tap FIR Filter	***Filter***	43	38	113

The assembler then translates assembly language source files into machine language object files. The machine language is based on common object file format (COFF). In Path "B", the assembly optimizer allows you to write linear assembly code without the need to focus on the pipeline structure or register assignment. The linker combines object files into a single executable object module. While creating the executable module, it performs relocation and resolves external references. The linker accepts relocatable COFF object files and object libraries as input. The run-time-support libraries contain ANSI standard run-time-support functions, compiler-utility functions, floating-point arithmetic functions, and I/O functions that are supported by the C compiler. As an option, the designer can use the library-build utility to build his/her own customized run-time-support library. Finally, the executable COFF file is downloaded to the DSP target for execution or loaded on the CCS simulator for debugging [45].

The integrated design environment (IDE) includes a compiler and debugger, which help to run and test programs without having to jump between applications [44]. Debugging is accomplished through the addition of probe points or break points as well as recording logs of outputs.

Within the analysis portion of the design cycle, CCS offers profiling options to evaluate the speed of an application and target the code that has been programmed inefficiently. In addition, with the use of DSP/BIOS an application can be debugged in real-time to make sure it meets real-time deadlines. With DSP/BIOS a designer can monitor a system's performance without affecting the target DSP. In addition, DSP/BIOS offers a graphical interface to implement projects using multi-threading. In addition it can plot the CPU load graph, for scheduling functions and monitoring CPU usage respectively. Both tools can help to maximize DSP resource usage [46].

4.1.2 TI Evaluation Module

The hardware chosen for implementing the OFDM system is a TI Evaluation Module (EVM), model number TMS320C6201. The full-sized EVM board measures approximately 4.2 inches wide by 12.28 inches long and is designed to be used in a PC's PCI expansion slot. A diagram of the EVM board is shown in Figure 22.

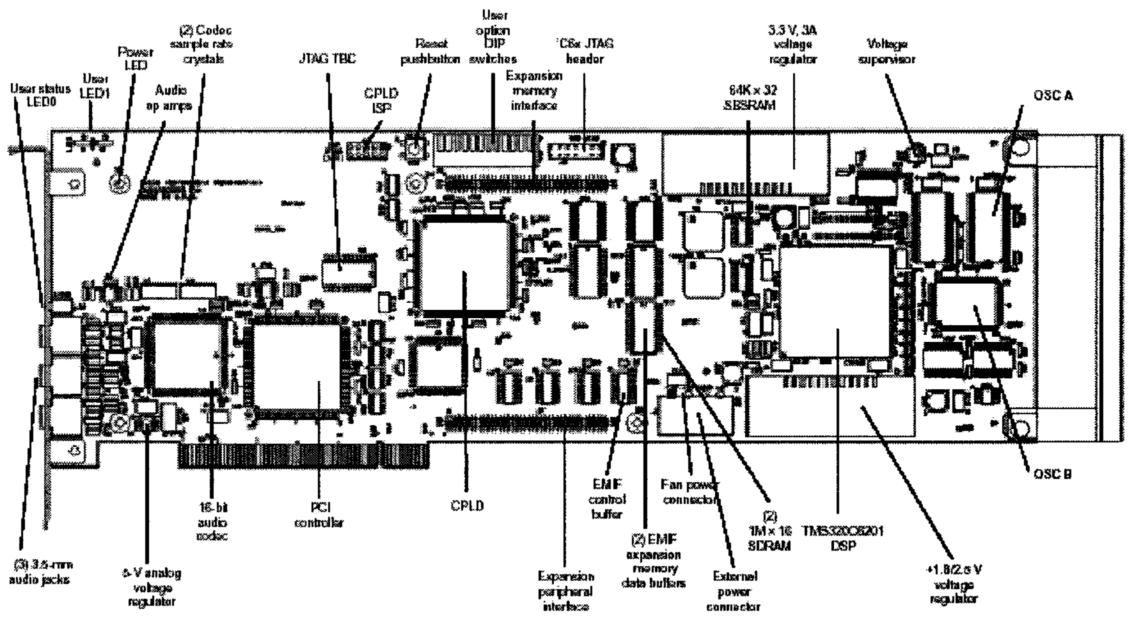


Figure 22: TMS320C6201 EVM diagram [47]

Functionally, the board allows high-speed verification of C6000 code with the included EVM specific Code Composer debugger. The EVM board features a PCI interface, SBSRAM and SDRAM. Connectors on the EVM board provide a DSP external memory interface (EMIF) and peripheral signals that enable it to be expanded with custom or third-party daughterboards [47].

The board includes a C6201 processor, which is a fixed-point processor. It is capable of 1600 MIPS at 200MHz using a 5ns cycle time, however only a maximum clock rate of 160 MHz is possible with the EVM. It is able to execute up to eight 32-bit instructions every cycle. The core CPU consists of 32 general-purpose registers of 32-bit word length and eight functional units, which are 6 multipliers and 2 ALUs. The CPU contains TI's own VelociTI architecture which makes the C6000 DSPs the first off-the-shelf DSPs to use an enhancement of traditional VLIW (Very Long Instruction Word) to achieve high performance through increased instruction-level parallelism. A traditional VLIW architecture consists of multiple execution units running in parallel that perform multiple instructions during a single clock cycle [47]. Other on board features such as memory and peripherals are summarized in Table 5.

Table 5: Features of TMS320C6201 EVM board [47]

Memory and Peripheral features
Glueless external memory interface to synchronous memories such as SBSRAM and SDRAM
Glueless external memory interface to asynchronous memories such as SRAM and EPROM
4-channel direct memory access (DMA)
Host port interface (HPI) with dedicated auxiliary DMA channel providing access to entire processor memory space
Two multichannel buffered serial ports (McBSPs) for direct interfacing to telecommunication, audio, and other serial devices
Two general-purpose timers
Multiply-by-4 phase locked loop (PLL) and multiply-by-1 PLL-bypass options
1M-bit on-chip memory (2K × 256 bits of program memory/64K bytes of data memory)

4.2 Fixed Point Processing

Designs based on fixed point processing offer the advantage of greater speed in computation, lower power consumption, and lower production cost over floating-point based designs. This is mainly because of the lower number of bits per word representing each data type. For example, some computers reserve 1 byte of memory for an integer in a fixed-point architecture whereas 2 to 4 bytes of memory can be reserved for a floating-point constant in a floating-point architecture [48]. However, the trade-off is a lack of precision in results obtained from fixed-point processors.

Most system designs start off as floating-point models but then go through a longer design procedure to convert to fixed-point design. The extra time spent in the design stage pays off with the savings in the production stage [49]. This is due partly to the extra on chip memory needed for representing floating point values.

Any system using fixed-point processors must consider the limitations of fixed-point arithmetic. The consequences can be significant precision losses otherwise. The main points to consider when designing such a system are: quantization error and saturation error.

4.2.1 Quantization Error

A fixed-point processor has a dynamic range that is far less than that of a floating-point processor. Given the limited range of fixed-point processors, they mostly perform mathematical operations on numbers representing positive or negative integers. Therefore, the precision is limited to whole, integer numbers because no bits are allocated to representing a decimal value as in floating point numbers. This leads to rounding error otherwise known as quantization error. Quantization error occurs when values lying within a given range are all assigned the same value [49]. Therefore, the system is unable to distinguish between these values and so precision suffers.

Some DSPs offer a method for fixed, fractional number representation where parts of the bits are used to assign a decimal value. This is known as Q15 or Q31 representation depending on how many bits are used to represent the decimal part [50]. However this is done at the expense of a further loss in dynamic range. This method may be better suited for signal representation of low voltage values. Otherwise a designer may use scaling to make use of the entire dynamic range. If the values used in operations all lie in close proximity to each other scaling allows for a small number to be represented by a large number thus spacing the number of values between the original two values. This scaled representation of the original numbers puts precision back into the operation. However scaling should be used cautiously as it can lead to the second main source of error, which is discussed next.

4.2.2 Saturation Error

Saturation errors are the outcome of computations that give results exceeding the dynamic range of the hardware. For example if two integer numbers, represented by 16 bits each, are multiplied and the result exceeds the maximum value that can be represented by 16 bits then overflow will occur. Once the maximum value is reached the hardware simply cycles over from the start therefore the resulting value can be completely different from the actual value.

To avoid overflow, some algorithms include special conditions that will always give the maximum value regardless of by how much it is exceeded. This is called saturation and is analogous to "clipping" of analog signals in an RF transmitter. If saturation is not acceptable then the algorithm has to avoid overflow altogether by scaling the two values to a lower value prior to the operation thereby ensuring the result will not exceed the maximum value.

4.2.3 Effects of error on system performance

It is clear that a system design with fixed-point processors has to balance the effects of quantization error and saturation error to obtain acceptable system performance. The solution of scaling values upward to minimize quantization error may cause excessive saturation errors and vice versa. The OFDM system designed in this chapter makes sure to take these factors into account in order to yield acceptable results. The methods used to minimize errors due to finite word lengths of a fixed-point processor are further described in later sections. However, even with good design practice, a system based on a fixed-point processor can never be as accurate as its floating-point equivalent. In [51] the effects of quantization on an OFDM system was investigated. It was found that the fixed-point model had a BER plot that was higher than its fixed-point equivalent model. Figure 23 shows the results.

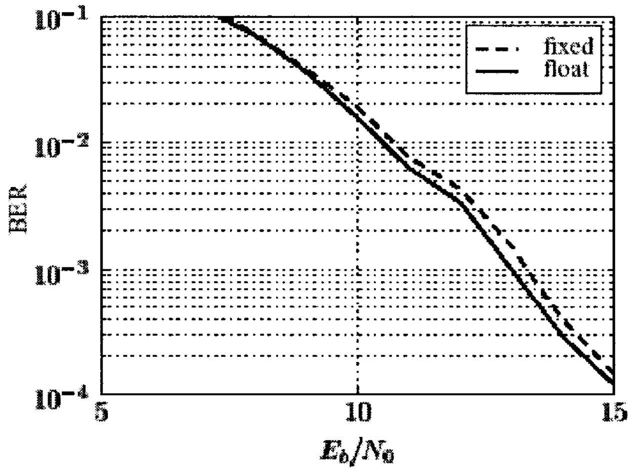


Figure 23: BER Comparison of a fixed-point and floating-point model [51]

4.3 Software Structure

The following section will detail the software structure of the OFDM system implemented on the TI EVM board. It will start with a general overview of the implementation, noting the tools used and the methodology. Later sections will detail each software module that represents the different blocks of the OFDM block diagram. A description of the functions will be given as well as methodology to implement the mathematical models.

4.3.1 Overview

The OFDM system is constructed with modularity such that each module in the system represents a particular block of the diagram in Figure 2. The module consists of one or more functions to implement it. The entire block diagram was implemented in C code. As mentioned, the compiler in the Code Composer Suite is used to convert the C code to assembly language. The build options were chosen to compile for increased speed at the expense of code size. Once the executable COFF file is created, it is downloaded to EVM hardware for execution [45]. The modules of the system are executed sequentially such that the data is completely processed by one module before moving to the next. Each function is listed in the ***main()*** function in the order in which it is executed.

The block diagram implemented is shown in Figure 24. Under the name of each block, in bullet format, are the functions that make up each particular block. What follows is a description of each module in the OFDM system.

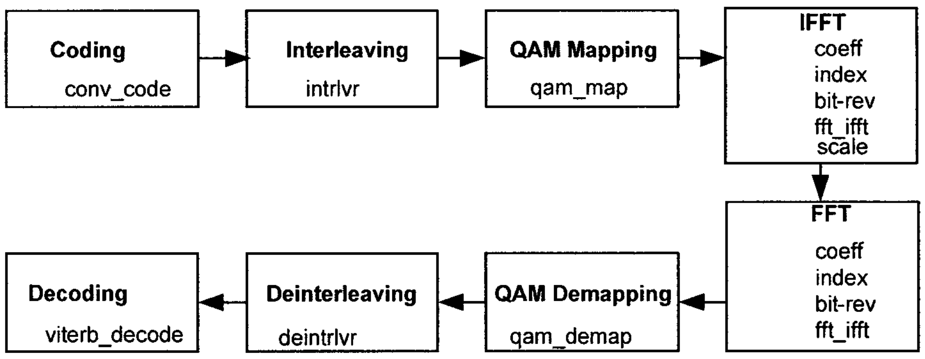


Figure 24: Block Diagram of implemented OFDM system

4.3.2 Coding

Error control coding was performed with the use of a convolutional encoder. The convolutional encoder is executed by calling the function ***conv_code*** from ***main()***. The module takes in as input, the 128 elements of an array titled *data []*. The generation of *data []* is further described in section 5.1.1. The elements of *data []* are stepped into an array *shift_reg[],* which represents a shift register of constraint length *K = 3* as shown in Figure 25.

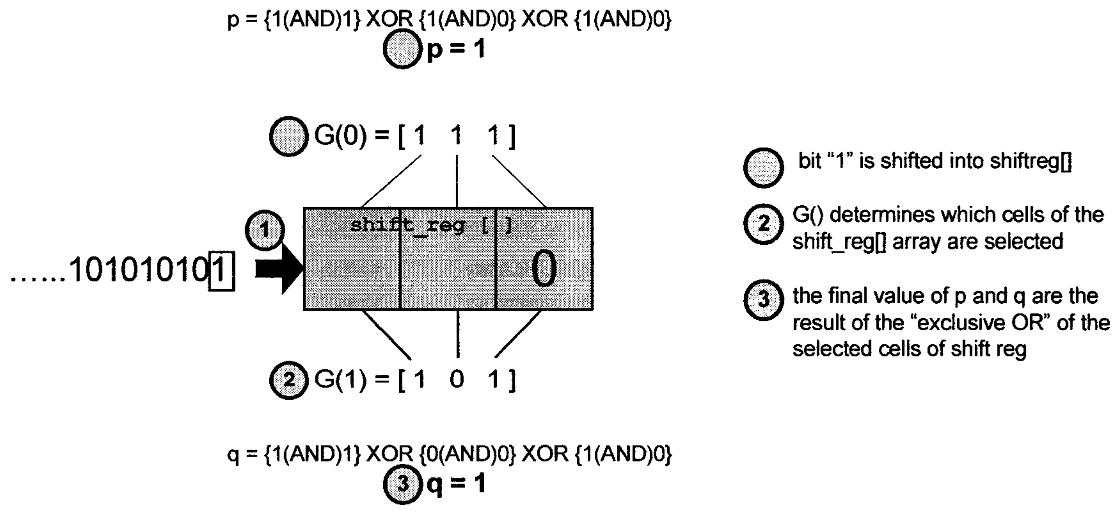

p = {1(AND)1} XOR {1(AND)0} XOR {1(AND)0}
p = 1
G(0) = [1 1 1]
......10101010[1]
1
shift_reg [ ]
0
2 G(1) = [1 0 1]
q = {1(AND)1} XOR {0(AND)0} XOR {1(AND)0}
3 q = 1
bit "1" is shifted into shiftreg[]
2 G() determines which cells of the shift_reg[] array are selected
3 the final value of p and q are the result of the "exclusive OR" of the selected cells of shift reg

Figure 25: Implementation of convolutional encoder

There are two connection vectors making the code rate ½. The connection vectors are defined by a two dimensional array, *g[m][K]*, where *m* = 2. The two-dimensional array contains the connection vector *G(0) = {1,1,1}* in the first row and connection vector *G(1) = {1,0,1}* in the second as taken from [28]. The algorithm loops the data into the *shift_reg[]* array one bit at a time. After the first shift, two coded bits are created for the single inputted bit. To create the output bits, the algorithm performs a modulo-two addition of only the contents of the *shift_reg[]* selected by each connection vector. The result for one of the coded, output bits is stored in integer variable *p* and the second in integer variable *q*.

The "selection" operation is performed by the logical "AND'ing" of each *shift_reg[]* array element with the corresponding connection vector element. The result is then stored in *p* for the first output bit and *q* for the second. The process is repeated for each element of the *shift_reg[]* array. With each iteration, the present value of *p* and *q* is exclusive OR'ed with the past value of *p* and *q* respectively. The "exclusive OR" operation is equivalent to modulo-2 addition as show in Table 6.

Table 6: Equivalence of "Exclusive OR" and Modulo-2 truth tables [48]

Bit Number 1 (b1)	*Bit Number 2 (b2)*	*Exclusive OR*	*Modulo-2 Addition*
0	0	**0**	**0**
0	1	**1**	**1**
1	0	**1**	**1**
1	1	**0**	**0**

The algorithm loops until all *K* elements of the *shift_reg[]* array have been operated upon. The final values of *p* and *q* make up the two coded bits for the bit that was input at the beginning of the cycle. The algorithm then inputs a new bit and the operations described above are repeated. The new, coded data stream is contained in an array titled *coded_data []*, which has grown from 128 to 256 bits.

The shift register is operated upon as a cyclic shift register to speed up the algorithm. Instead of feeding a new bit in from the left and right shifting all other bits to discard the rightmost bit, a pointer value keeps track of the bit positions. In this manner, only the bit that is no longer needed is over written with the newest, inputted bit. The pointer keeps track of the other bit positions so that they are operated on by the appropriate connection vector. This saves the task of copying or reallocating data.

4.3.3 Interleaving

The interleaver is implemented using the block matrix method. The interleaver module is executed by calling the ***interlvr*** function from ***main()***. A two-dimensional, square array, *interlv[n,n],* is declared where the row and column dimensions are equivalent to the size of the square root of the *coded_data* array as shown in Figure 26.

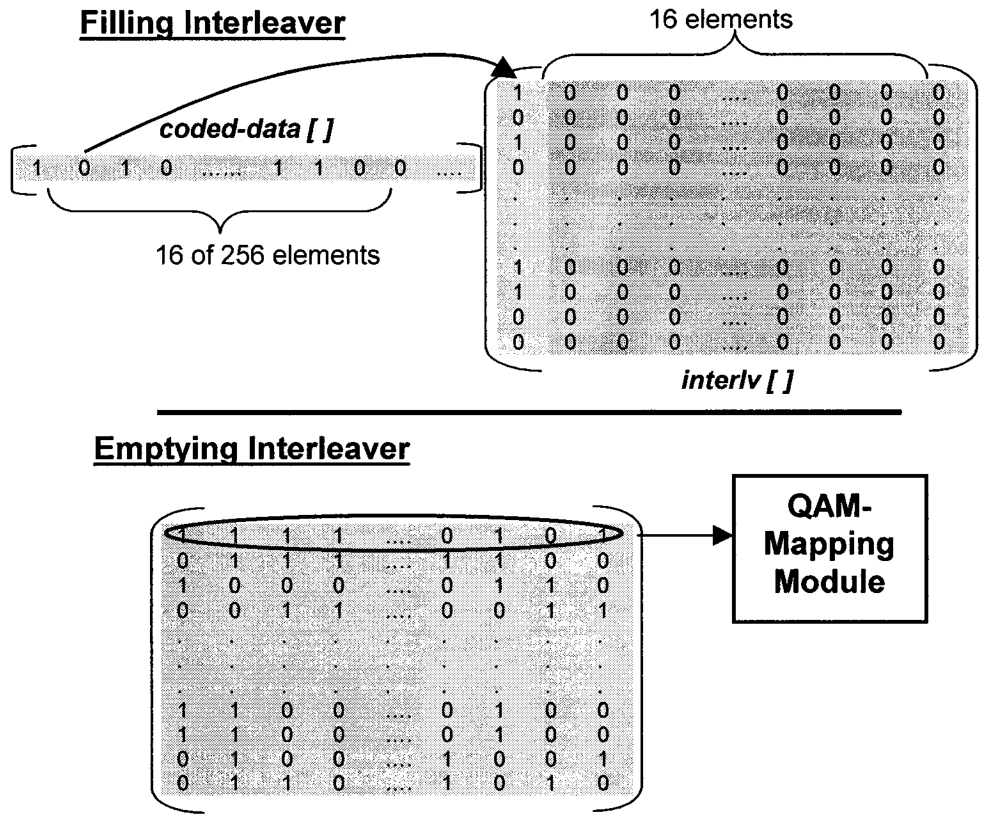


Figure 26: Graphical representation of Interleaver module

The interleaver works by reading the *coded_data* bit stream into the *interlv[n,n]* array in a vertical manner until all the columns are filled. Once this task is complete, the first row and then subsequent rows are read out as the new, interleaved bitstream, which is sent on to the QAM-mapping module for processing.

4.3.4 QAM Mapping

The QAM mapping module is executed by calling the ***qam_map*** function from ***main()***. It takes in as input, the interleaved, coded bit sequence and maps the bits to a complex QAM symbol. Each QAM symbol will consist of two values representing the in-phase and quadrature magnitudes. The gray encoded, 16-QAM constellation chosen for this system is shown in Figure 27.

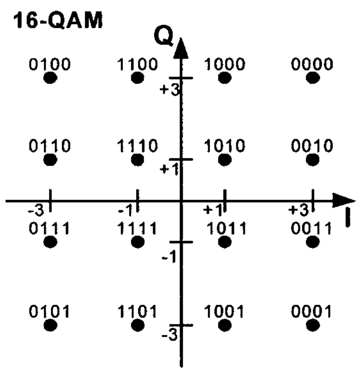


Figure 27: 16-QAM constellation used in OFDM system

The QAM symbols are stored in a look up table (LUT) therefore no calculation to determine the value is needed. An efficient method to implement QAM mapping is to use the decimal equivalent value of the bit group as an index to the location of the QAM values in the LUT. Figure 28 graphically demonstrates the QAM mapping procedure.

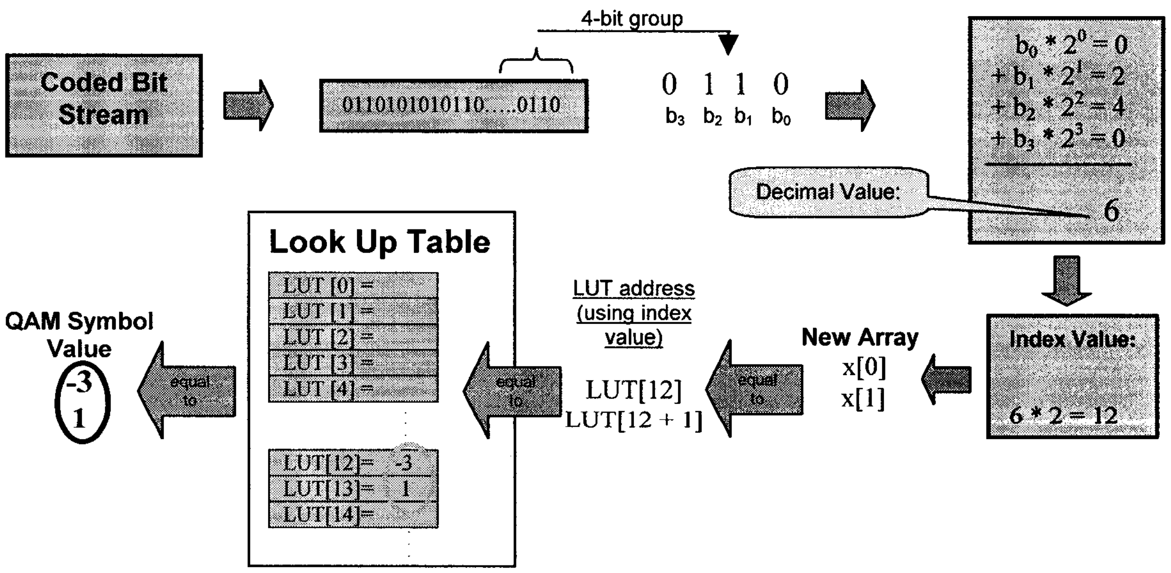


Figure 28: Implementation of QAM mapping using an LUT

The first step is to divide the incoming sequence into groups of bits. Using a 16-QAM constellation requires that each group consists of 4 bits. The 4-bit group is then converted to its decimal equivalent by multiplying each bit value by 2^a, where "*a*" is the bit position number. The decimal value is used in determining the "address" to locate the proper QAM symbol in the LUT. Since there are two values that make up each QAM symbol, the decimal value is multiplied by 2 to get the starting index value corresponding to the in-phase QAM value. The quadrature value is found simply by adding one to the index value. For example the 16-QAM constellation from Figure 27 has values (-3,1), corresponding to the 4-bit group "0110". The decimal value was found to be 6 and the index value 12, therefore -3 and 1 would be in the 12th and 13th memory location in the LUT array. The LUT array is formed by simply organizing the Gray encoded constellation of Figure 27 and storing the decimal equivalent of the 4-bit values in an array titled, *qam_lut []*, in linear order starting from decimal value *0* to decimal value *15*.

For each bit-group that is mapped, the complex QAM values are placed in a new array, *x [ ]*, which will bring the element count back to 128 down from 256 in the original *coded_data []* bit stream.

The entire QAM constellation can be defined as a combination of values taken from the following group of numbers: {-3, -1, 1, 3}. These values are relatively close when compared to the entire dynamic range associated with a 16-bit word length. In fact, using such low values would cause large quantization errors to occur in the IFFT module that follows. Therefore a scaling factor to "spread out" the QAM values should be used. Ideally, we would like to choose the largest integer values possible. For a 16-bit processor we can represent values between –32768 and 32768 so that our QAM values would be {-32766, -10922, 10922, 32766}. Note that 10922 is exactly one third of 32766 to keep the same proportions as 1 and 3. However this would cause saturation errors in the IFFT module. The actual values chosen to represent the QAM symbols are {-510, -170, 170, 510}. These values are approximately 1/64th the maximum integer value to prevent overflow. A scaling by 64 was used which directly relates to the number complex pairs in the *x[]* array processed by the IFFT and FFT modules. System performance and accuracy is described further in Chapter 5.

4.3.5 Inverse Fast Fourier Transform

The IFFT module [50] is composed of five functions. They are titled, in the order in which they are called from ***main()***, ***coeff***, ***index***, ***ifft_fft***, ***bitrev***, ***scaling***. The first function, *coeff*, generates the twiddle factors, W_N, used by the *ifft_fft* and the second function, *index*, generates the index values used by the complex *bitrev* function. These two functions (*coeff* and *index*) are executed at compile time and do not use up any cycle time while the OFDM system is running. The *ifft_ftt* performs a 64-point complex IFFT on the 128 element *x []* array, using the DIF algorithm. It is then followed by *bitrev* to bit reverse the output values and place them in linear (normal) order. The following describes each function in further detail.

4.3.5.1 Twiddle Factor Generation

As mentioned this program calculates the twiddle factors to be used in the *ifft_fft* function. Since the size of the IFFT is known beforehand ($nx = 64$ for this system) we know exactly how many twiddle factors must be computed. These are stored in an array *w[]* until needed by the IFFT/FFT function. The program computes the operation shown in Eq. (4.1).

$$W_N^{nk} = e^{-j2\pi nk/N} = \cos(2\pi nk/N) + \sin(2\pi nk/N) \tag{4.1}$$

The program separates the calculation of the exponential, *e*, into its cosine and sine constituents. Therefore it should be noted that two memory locations (e.g. *w[0]* and *w[1]*) are needed to make up any W_N value. The even *w[]*'s will contain the cosine values while the odd w[]'s will contain the sine values.

The program contains two separate "*for*" loops, which calculate a different set of twiddle factors for the *ifft_fft* function depending on which is needed. Only one loop is active at during the function call and is chosen by evaluating the "inverse" expression. If "*inverse*" is set to true (i.e. = 1) then twiddle factors for the IFFT are computed otherwise twiddle factors for the FFT are computed.

Within the computation of the twiddle factors, a call to the *cosine* and *sine* functions is made. This is a time consuming and inefficient operation. However, a call to the *coeff* function is only needed once so that the sine and cosine values are calculated before the OFDM system needs to be run. It is irrelevant to the timing and performance of the system therefore a reduction in the complexity of calculating the cosine sand sine values is not needed.

In addition, given that this is a fixed-point processor, each *w[]* value is scaled (i.e. multiplied by 2^{15}). The reason for this is precision related as discussed in section 4.2. Without the scaling factor, the result of the *sine* or *cosine* function would be a relatively small number and would have all the precision lying in the decimal value. Given that the decimal value is lost in fixed-point processing gives rise to unacceptable rounding errors.

4.3.5.2 IFFT/FFT

Executing the *ifft_fft* function performs the complex IFFT. The algorithm was coded and implemented as decimation in frequency (DIF). It takes in as inputs, 128 complex data points stored in array x in the form *x [real1, imgnry1, real2, imgnry2 ...]*. It then calls on the corresponding twiddle factor values stored in array *w[]* depending on the value of inverse as shown in Figure 29.

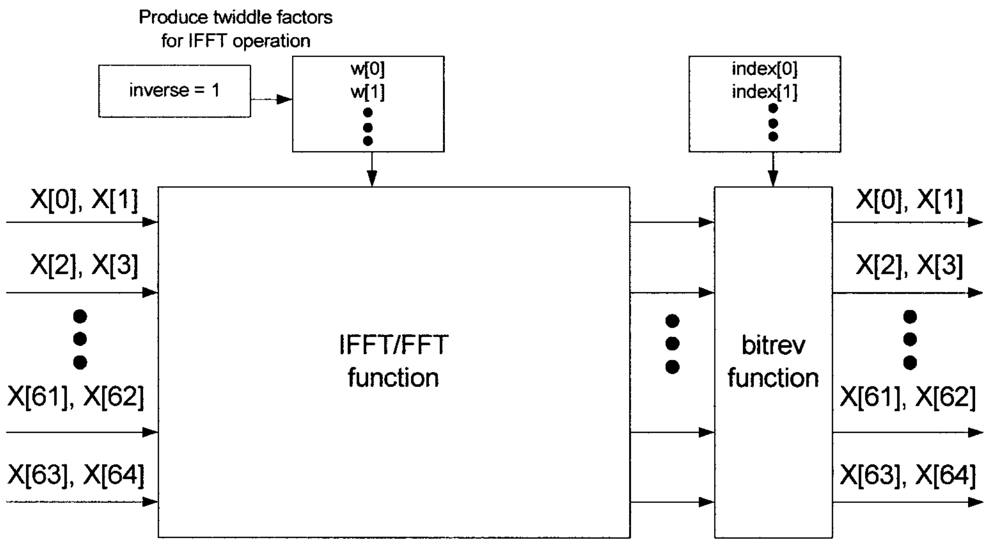


Figure 29: IFFT/FFT Module

The algorithm essentially replicates a flow graph similar to that of Figure 6 in section 2.2.1 except there would be $\log_2(64) = 6$ stages instead of 3. From the value of *n*, it determines the number of stages and simply performs the elementary operations at each node. When multiplying by the twiddle factors at each node, a bit wise shift with the value 15 is performed. This is equivalent to dividing by 2^{15} to remove the scaling added by the twiddle factors and return the result to its proper value.

The main points to note in creating the algorithm are [24]:

- the input sequence is in numerical order

- the output sequence is not in numerical order, they are in bit reversed order thus requiring random access memory to more efficiently implement the algorithm
- the branches and nodes consist of elementary operations such as adds, subtracts and multiplication by twiddle factors (i.e. W_N)
- the twiddle factors can be calculated in advance to save processing time
- performing the algorithm while following the flow graph order results in "in-place computation" where the results from stage "m" replace those of the previous stage "m-1" thus requiring less memory capacity.

4.3.5.3 Bit Reversal

Two functions are called upon to execute the task of bit reversal: *index* and *bitrev*. Bit reversal of the output of the *ifft_fft* function is necessary because it performs either the IFFT or FFT using the DIF algorithm. Note that bit reversal of the input would have been necessary if the algorithm was DIT. The method used in our algorithm is to create a lookup table that holds the normal order position for every bit-reverse position value entered. The tree diagram shown in Figure 30 on the next page helps to explain the iterative approach.

Given the binary value of any array position that is bit-reversed (e.g. *x[1,0,1,0]* = *x[m3,m2,m1,m0]*), the value is sorted by first looking at the least significant bit, *m0*. If it is "0", it is placed in the top half otherwise it is placed in the bottom half [27]. The process is repeated for each bit, thus adding new levels (or branches) to the tree. Once the tree is formed we can feed in an entire bit-reversed data sequence and the path through the tree determines the normal order position, (e.g. *x[0,1,0,1]* = *x[n2,n1,n0]*). This is repeated for every data element in sequence until the entire array is sorted. The tree can be formed once, in advance, so that it can act as an index look up table.

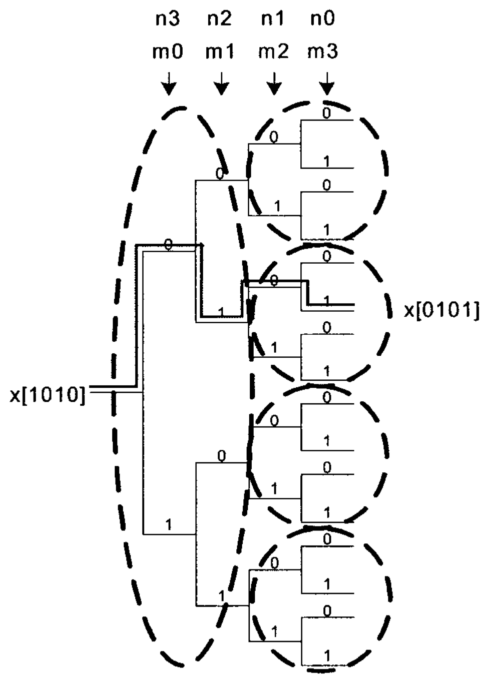


Figure 30: Implementation of index algorithm for bit reversal [27]

This is essentially the task of our *index* program. It begins by taking in as an input, the number of elements in an array, and tallies how many binary bits are needed to represent each number. For example, the array in Figure 30 has 4 binary bits (= $log_2 16$) representing a total of 16 possible elements. This also determines the number of levels in the tree (i.e. 4 in this case). This routine requires *N* cycles for an N-point FFT to complete. This is faster than a routine that would swap values, which would require $Nlog_2N$ cycles. However, the same amount of memory used to store the data array is needed for the array in the index look up table, which can be significant for large *N* values [27]. A solution can be achieved by noting that large tree diagrams are essentially made up of smaller tree diagrams that repeat such as in Figure 30. Note that the entire tree is made up of a combination of smaller, identical trees, which are circled. Therefore only an index lookup table for the small tree need be made and called on more times to form a sorting routine for large N values. The same N cycles are needed, however, the memory requirements are far less because the maximum size for the lookup table is x*sqrt(N) (where x is the square root of the FFT radix value). Calls to the *index* routine are made as needed by the *bitrev* function to perform the bit-reversing task.

4.3.5.4 Scaling

Executing the *scale* function performs scaling of the output data. It is performed after bit reversing. The scaling operation simply divides each element of the output sequence by the value of variable *n* as shown in the coefficient of Eq. (2.3) in section 2.2.1.

4.3.6 Fast Fourier Transform

The FFT module is the first module executed on receiver side of the OFDM block diagram. It reverses the IFFT operation performed just before transmission. It is essentially the same as the IFFT module as almost all the functions in the IFFT are called upon to perform the FFT. The difference being that the variable *inverse* is set to "0" before the call to the *coeff* function in order to generate the correct twiddle factors. In addition, scaling is not needed; therefore the *scale* function is not called.

4.3.7 QAM Demapping

The QAM demapping module is executed by calling the ***qam_demap*** function from ***main()***. Two QAM demapping modules were tested. One was programmed for hard decision and the other for soft-decision. Either of the two functions takes in as input the complex QAM values contained in the *x []* array that were recovered from executing the FFT module. However, because of impairments caused by the simulated Gaussian channel, the amplitudes of the QAM data have been affected and do not equate to the discrete values derived from the QAM mapping module. Therefore, decision regions are used to represent the discrete QAM values. The decision regions are created by dividing the original Cartesian plane of Figure 14 into 16 hard decision regions as shown in Figure 31. Note that the regions lying on the perimeter extend to infinity.

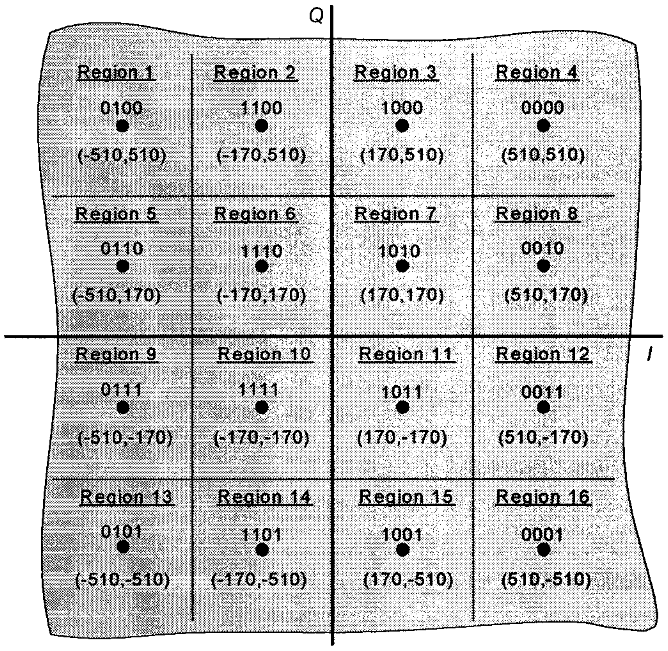


Figure 31: Division Cartesian plane into QAM regions

A received QAM value is evaluated to see what region it falls into and is then assigned the 4-bit group associated with that region. For example, any value falling within region 7 would be assigned QAM value 170,170. The corresponding four bit binary group "1010" then replaces the QAM value. This process would be sufficient for a hard decision receiver, however, the Viterbi decoder used in this system takes 3-bit soft-values as inputs, therefore each of the 16 regions need to be represented by the soft decision equivalent value. Recall from section 2.3.3 that a 3-bit value of "111" indicates a strong possibility of having received a binary "1" and a 3-bit value of "000" indicates a strong possibility of having received a binary "0". Since each region has a 4-bit group, all binary values can be converted to their 3-bit equivalent values. For example, region 7 from Figure 31 which was originally "1-0-1-0" would become "111-000-111-000". For simplicity we will use octal equivalent values to represent each 3-bit group giving us "7-0-7-0". Note however that since we are still only using a unique bit sequence to represent each of the 16 regions, therefore the system still works using hard decisions in this case.

To achieve a signal to noise ratio gain from using the soft decision capabilities of the Viterbi decoder, each region should be further divided into soft decision regions. We can use the uniform quantizer of Figure 32, as a basis for our QAM regions.

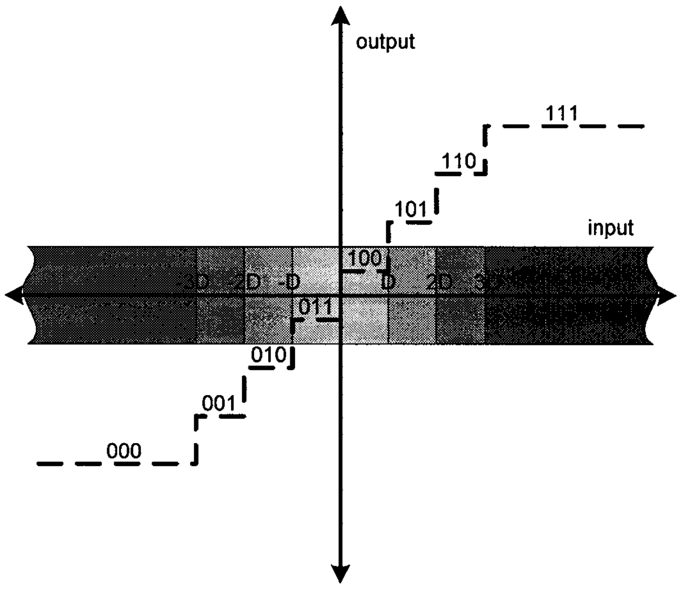


Figure 32: Uniform quantizer

The uniform quantizer shown in Figure 32 is created for a binary system. We divide the regions on opposite sides of the threshold line into equal distances "D" and assign a 3-bit value for each soft decision region. For example, a value falling within "D" and "2D" would be assigned the 3-bit value "1-0-1". Note, however, that the region starting at –3D extends to negative infinity and the region starting at 3D extends to positive infinity. This idea can be extended for use in the 16-QAM constellation used in the OFDM system. However, a 16-QAM constellation works in two dimensions and multiple regions lie next to one another in one dimension. Regions 5,6, 7, and 8 from Figure 31 would be divided as shown in Figure 33.

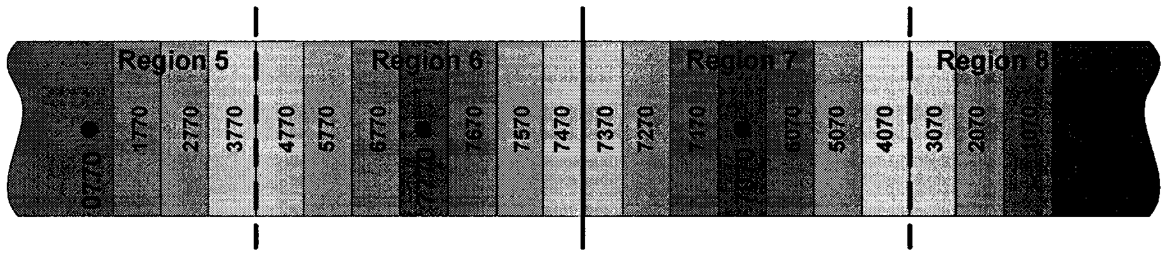


Figure 33: Soft Decision regions between Regions 5,6,7, and 8

Since the QAM symbols are Gray encoded, we know that as we move in one direction from one region to the next, only one bit in the 4-bit group will change. For example, moving from region 7 to region 8 in Figure 31 causes only the first bit of the 4-bit group to change from "1" to "0". Therefore, even though 4 bits are used for every region, we are only concerned with the one that is changing from one region to the next. If we place soft decision regions between regions 7 and 8 we will see a more gradual transition from value "7-0-7-0" to "0-0-7-0" as shown in Figure 33. Therefore a lower measure of confidence is given for values at the border between two regions (indicated by a lighter blue color) compared to values falling close to the center of the region.

Extending this idea into two-dimensions gives us the Cartesian plane represented in Figure 34. Because each region can be surrounded by at most four other regions, each region must be divided into soft decision regions for each of its four sides. Figure 34 shows the layout of the soft decision regions for all regions within the 16-QAM constellation. The blow out portion of region 11 shows the soft decision regions numbered 1 through 13.

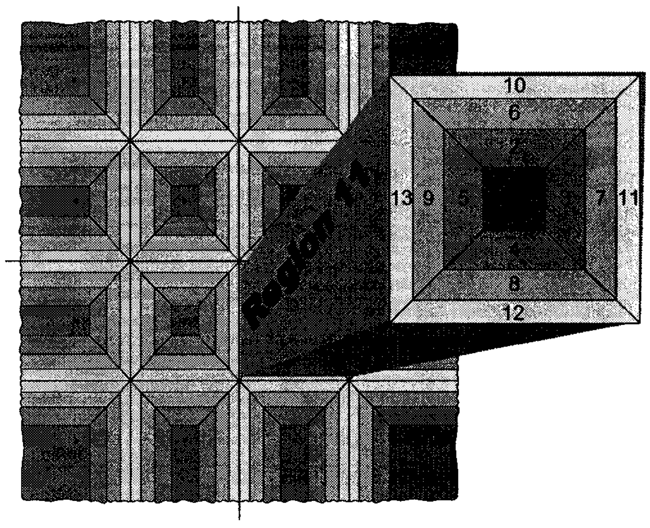


Figure 34: Example of soft decision region assignment for Region 11

A matter of consideration should be given to the distance "D" used to divide the regions into soft decision regions. A uniform distance "D" as used in the uniform quantizer of Figure 32 would not translate to a uniform area for each soft decision region as shown in Figure 35.

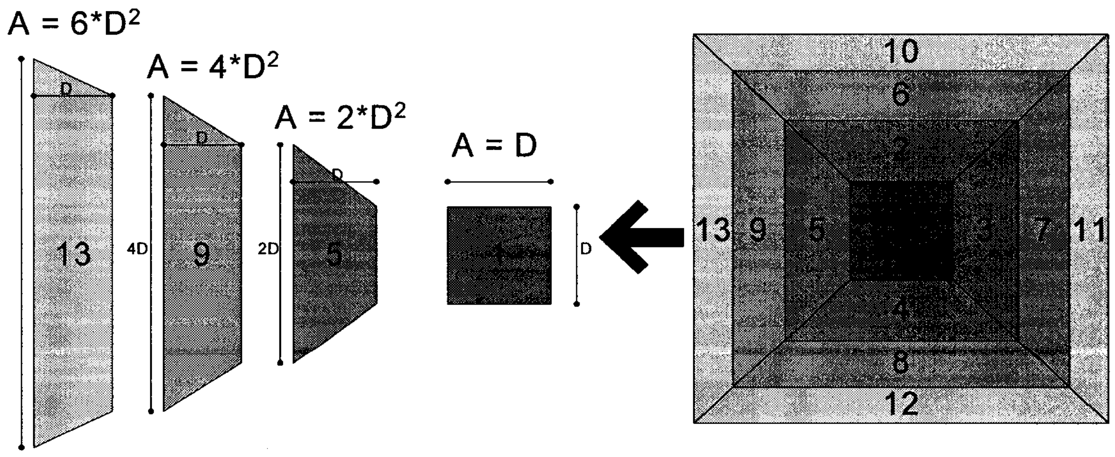


Figure 35: Effect of uniform distance "D" on area

Note that soft decision region 13 is six times as big as soft decision region 1. To divide the regions so that all soft decision regions are equal in area, we must vary the value of D as shown in Table 7.

Table 7: Varying value of "D" for uniform area

Soft Decision region	Old distance	New distance	New Area
1	D	d	d^2
2,3,4,5	D	$\frac{d}{\sqrt{2}}$	$2*\left(\frac{d}{\sqrt{2}}\right)^2 = d^2$
6,7,8,9	D	$\frac{d}{\sqrt{4}}$	$4*\left(\frac{d}{\sqrt{4}}\right)^2 = d^2$
10,11,12,13	D	$\frac{d}{\sqrt{6}}$	$6*\left(\frac{d}{\sqrt{6}}\right)^2 = d^2$

Both uniform distance and uniform area were tested in Chapter 5 to see its effects on BER performance. All hard and soft decision regions are defined within the *qam_demap* function. The algorithm is programmed to search for the soft decision region containing the received value. The first step in the soft-decision QAM demapper module is to locate which region the received symbols lie in much like the hard decision module. Since each region has equal chance of receiving a symbol, the regions can be searched in any order. Once the hard region is found, the algorithm then tests to see which soft decision region contains the received symbol. The algorithm searches if the value lies in the middle soft region and works its way outward in a circular fashion as shown by the numerical order of Figure 34.

In a situation where the signal to noise ratio is high, there is a high probability of finding the QAM symbols to lie in the first soft decision region ending the search and moving on to the next received value. However the time to find the correct soft decision region will be extended if the value lies closer to the borders between regions, which would happen more frequently with a lower signal to noise ratio. Therefore there may be a throughput speed difference at different signal to noise ratios. The two extremes were tested and the bit rate was measured in Chapter 5.

Once the correct soft decision region is found, the soft-decision bit sequence that is contained in that region (such as those shown in Figure 33) is read and copied into an array titled *outstrm []*. The *outstrm []* array will contain 256 elements when all the elements of the FFT module are processed.

4.3.8 De-Interleaving

The de-interleaver is implemented using the block matrix method as well. The de-interleaver module is executed by calling the ***deinterlvr*** function from ***main()***. A two-dimensional, square array, *deinterlv[n,n],* is declared where the row and column dimensions are equal to the square root of the size of the *outstrm* array as shown in Figure 36.

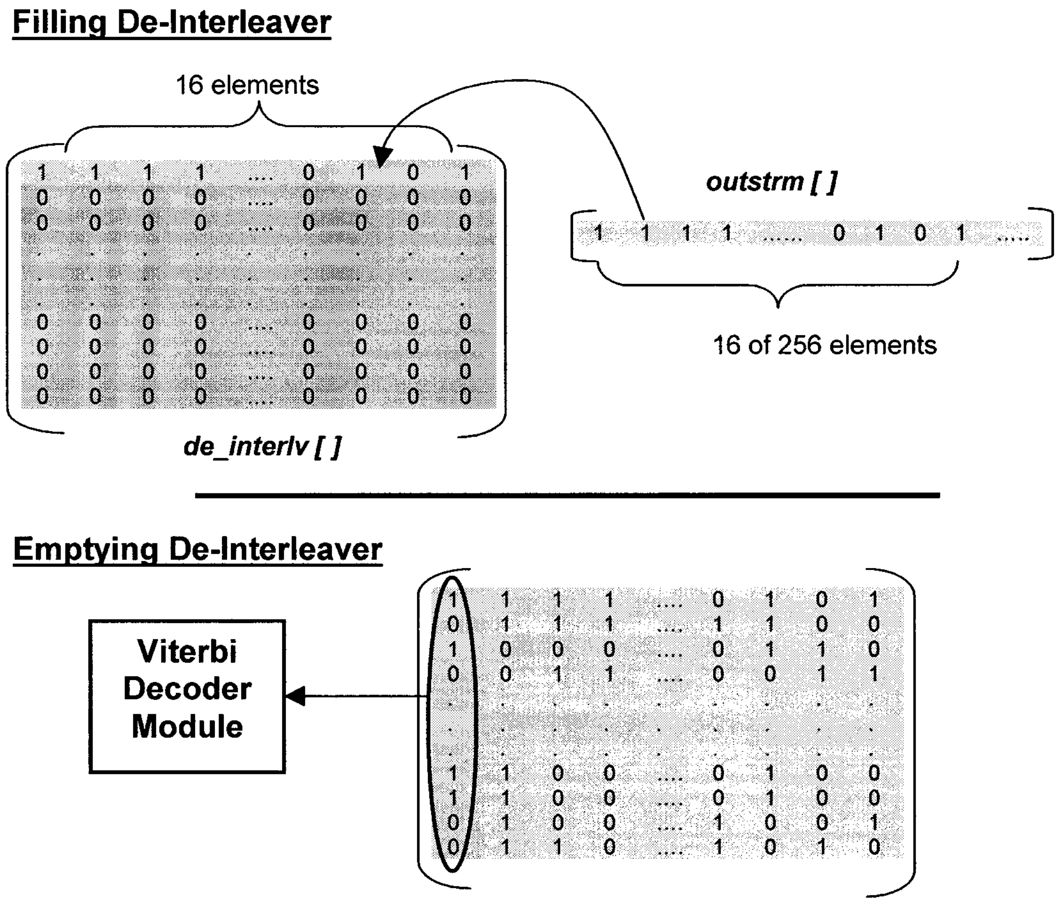


Figure 36: Graphical representation of De-Interleaver module

The de-interleaver works by reading the *outstrm* bit stream into the *de_interlv[n,n]* array in a horizontal manner until all the rows are filled. Once this task is complete, the first column and then subsequent columns are read out as the new deinterleaved bitstream, which is sent through the next Viterbi module for decoding.

4.3.9 Decoding

The Viterbi decoder is implemented by calling on the ***viterbi_decode*** function [52] from ***main ()***. The convolutional encoder described in section 4.3.2 was set up to encode with a constraint length of *K=3* and a code rate of *n=1/2*. Therefore this knowledge can save processing time storing comparison data for the decoder in LUTs.

Three matrices are formulated and stored in two-dimensional arrays that will be used as LUTs for the decoding algorithm. They are: *input[] []*, *nextstate[] []*, and *output[] []* . The matrices are filled in with data representing the encoder as a state diagram where the *K-1* rightmost stages of the shift register determine the state. Therefore every state will be made up of two binary values. The state diagram is reproduced in Figure 37.

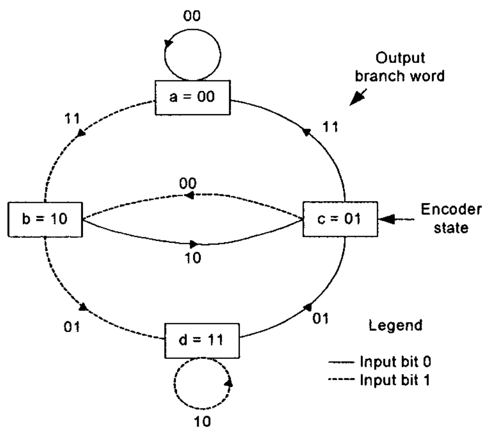


Figure 37: State Diagram for K=3 [28]

The contents of each matrix are shown in Figure 38. With the exception of the Input matrix, all contents are represented in the decimal equivalent of the binary value. For example, with respect to the Next State matrix, a Next State value of 2 (i.e. binary value of "10") is placed in the first row (represented by a Current State value of "0" or binary value "00") and the second column (represented by an Input value of binary "1"). The other matrices are filled in with the same rules for the rows and columns using the state diagram of Figure 37.

Input
col: Next State
row: Current State

	0	1	2	3
0	0		1	
1	0		1	
2		0		1
3		0		1

Next State
col: Input
row: Current State

	0	1
0	0	2
1	0	2
2	1	3
3	1	3

Output
col: Input
row: Current State

	0	1
0	0	3
1	3	0
2	2	1
3	1	2

Figure 38: LUTs, Input, Next State and Output for Viterbi Decoder Algorithm

The first step in the decoding process is to take in as input, the soft-decision values stored in the de-interleaver. The *viterbi_decode* function then implements the Viterbi decoder algorithm by comparing the received, encoded bits with all possible transmitted combinations and choosing the most likely candidates through the lowest branch metric value calculations. It then stores the lowest branch metric values in the *accum_err_metric[]* array. The algorithm keeps track of the most likely paths by building a trellis represented by the array called *state_history [].* Once the trellis grows to a predetermined value of *K*5*, the first bit can be decoded. The trace back portion of the algorithm consists of forming a *state_sequence []* array from the information contained in *state_history []* and using the earliest two values to determine the most likely transmitted bit. Both the *state_history* and *state_sequence* array are operated on as a cyclic buffer much like the shift register in section 4.3.2. In the same manner, pointers are used to overwrite the oldest values with the newest values instead of wasting cycles to shift all elements of the arrays to the right in order to input the newest value from the left.

Let us take an example of the decoding algorithm using an arbitrary received value of "***7-0***" at some time ($t = i$) and fill in the *accum_err_metric* and *state_history* array using information from the trellis diagram. The algorithm will perform an operation equivalent to the leftmost trellis diagram of Figure 39.

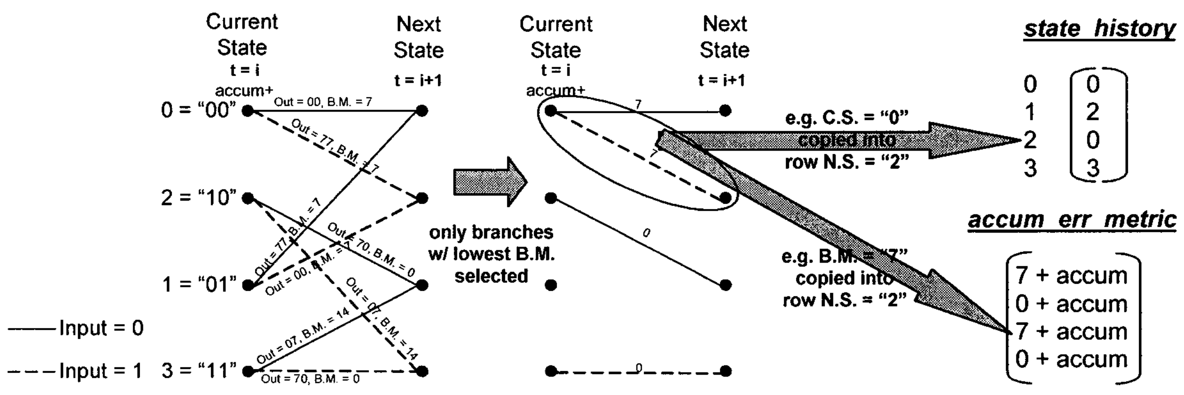


Figure 39: Translation from Trellis Diagram to arrays in Viterbi algorithm

It will use the information of the Output array to compare each element (eight in all) with the received encoded bit "*7-0*". The branch metric is computed by performing an absolute value subtraction between the received value and each Output array element. The final results are stored in the *accum_err_metric* array in the row corresponding to the next state value. For example "*7-0*" compared to "*7-7*" would yield a branch metric of "*7*". When two different current states arrive at the same next state (at $t = i+1$), only the path with the lowest branch metric is kept as shown in the second trellis diagram of Figure 39. Note that in the above example, the next state value "00" has two paths arriving with the same branch metric value (i.e. 7). We assume the branch with the lowest "accum" value is chosen. In the case of a tie, an arbitrary rule is used to select any of the two paths. This information is represented in the *accum_err_metric* array where "accum" is the value of the accumulated error from all times prior to $t = i$. The *state_history* array contains the value of the current state in the row corresponding to the next state (the next state value is extracted from the next state array). Therefore one column of the state history array holds all the information to describe the trellis diagram transitioning from $(t = i)$ to $(t = i + 1)$.

The algorithm then moves on to the next received value and performs the same operations. Once the *state_history* has been filled, the *state_sequence* array can also be filled in by using the row value of the lowest accumulated error in the *accum_err_metric* array as shown Figure 40.

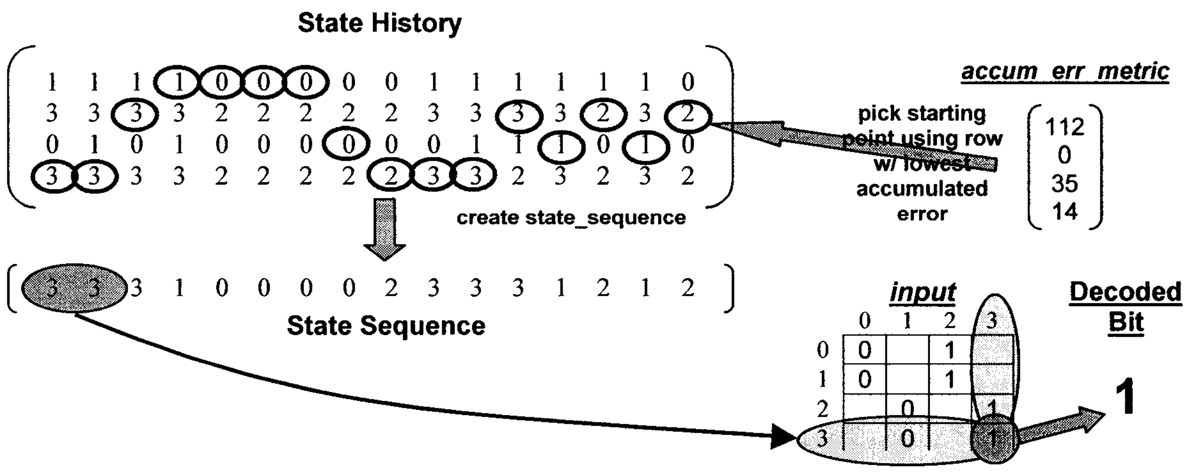


Figure 40: Implementation of decoder algorithm

This is the starting point of the *state_history* array. The *state_sequence* array is filled in with the path taken while working backwards through the *state_history* array. The last two values of state_sequence (i.e. 3,3) are used as the current (row) and next state (column) values respectively to decode the first bit using the Input matrix.

Once the first bit is decoded, the algorithm loops again to evaluate the next encoded bit and repeats the decoding procedure. The decoded sequence is stored in *decoder_output_matrix []* for comparison with the original inputted sequence.

5 Test Results and Analysis

The following sections will give details on the techniques used to test the system for accuracy and throughput. In addition estimates are made for increased performance with a faster processor. Finally an analysis will be given on the use of DSPs as a basis for implementing baseband processing of SDRs.

5.1 Testing

The OFDM system was tested for accuracy and performance. In order to conclude the system has sufficient accuracy, the outputs of each function module were monitored for correctness using a known input bit pattern. Once each module was functioning correctly the system was left to run with a random input pattern of one million bits. The output was compared with a delayed version of the input for any errors. Once it was determined the system functioned without error in a noiseless system, an AWGN channel was added to obtain a bit error rate plot. A fading channel was not chosen because the performance of OFDM in a fading channel is well known. Therefore an AWGN channel was chosen simply to demonstrate the system's correctness and accuracy.

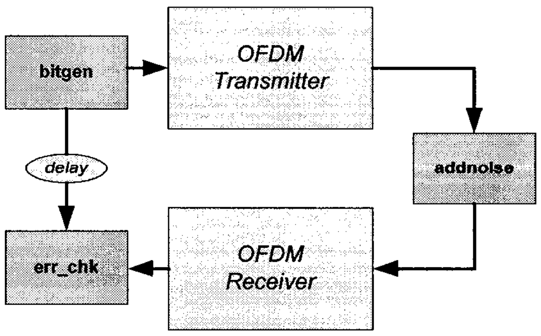


Figure 41: OFDM system with testing modules added

The following sections give a description of the three added software modules (*bitgen*, *addnoise*, and *err_chk*) responsible for testing the OFDM system. The modules are shown in Figure 41.

5.1.1 Bit generation

The EVM is supplied with a number of interfacing options as shown in section 4.1.2 to provide input data. However, a more straightforward and practical approach to test the functionality and accuracy of the system is to program a random bit generator on board as this provides greater control for debugging. The random bit generator is executed by calling the function ***bitgen*** from ***main()***.

The function *bitgen* contains the C function *rand(),* which is used to generate a random sequence. When called upon, *rand( )* will generate a random integer number that is uniformly distributed between 0 and 32767 [48]. Note that 32767 is the maximum integer value that can be represented in a 16-bit fixed point processor. In order to create a bit generator with values "0" or "1" a simple integer division by 16384 is used. Integer division will always truncate the result therefore any random number generated by *rand( )* that falls between 0 and 16384 will have an integer division result truncated to "0" and any number between 16384 and 32767 will have an integer division result truncated to "1". Given that *rand( )* produces numbers that are uniformly distributed between the entire dynamic range (0 to 32767), "0" and "1" have an equal probability of occurrence [48].

A matter of consideration is the "seed" value of the *rand( )* function. The seed value is a number on which the random number sequence is based. Therefore, if the seed value remains the same (the default value is 0) then a random sequence generated on two separate program runs will be identical. In this case, a "pseudorandom" sequence is a more appropriate name. The advantage of this might be for debugging purposes. In order to truly randomize the bits a seed change should be initiated at the beginning of each program run. This can be done with the function *srand(A)* where "*A*" is the seed value. Using the command line "*srand(time(NULL))*" returns an integer number representation of the present time within the dynamic range. Given that the time is always varying, this will give a different seed value each time, resulting in a more realistic random sequence.

When executed, the *bitgen* function produces sixteen values to be stored in an array called *data [ ]* which represents the initial bit stream. With the same function call, the *data [ ]* values are copied to another array called *buffer [ ]* which is twice the size of *data [ ]*. Copying to *buffer []* alternates between the first and second half of the array with each function call while overwriting the previous values. The *buffer [ ]* array is needed as a delay to compare transmitted and received sequences for the error count. Error counting is further explained in the *errchk[ ]* function in section.

5.1.2 Noise Generation in Simulated Channel

A noise generator was programmed using code to simulate an Additive White Gaussian Noise channel (AWGN). The purpose was to evaluate the system's performance in a noisy channel with and without forward error correction.

The noise generator is executed by calling on the ***addnoise*** function from ***main()***. The C language cannot generate Gaussian random variables (RVs), only uniform RV's, therefore a work around is needed.

In order to generate a Gaussian RV, we must first realize that a mathematical formula exists between Uniform, Rayleigh, and Gaussian RVs. Eq. (5.1) and Eq. (5.2) demonstrates the mathematical relationships

$$R = \sigma * \sqrt{2.0 * \ln\left(\frac{1}{1-U_1}\right)} \tag{5.1}$$

$$G = \mu + R * \cos(2 * \pi * U_2) \tag{5.2}$$

where R is the Rayleigh RV, U_1 and U_2 are uniform RVs, G is a Gaussian RV, σ is the variance, and μ is the mean. Using these formulas, we can now write a program to generate an AWGN channel. First, two uniform, floating point, random variables, U_1 and U_2 are generated between decimal value 0 and ($1 - 1\text{x}10^{-6} \approx 0.999999$). This is performed using *rand( )* function and then dividing the resulting RV by 32768. The first uniform random variable is used to generate the Rayleigh random variable and the second uniform random variable along with the recently generated Rayleigh RV is used to generate the Gaussian RV as in Eq. (5.2) [52].

The Gaussian RV is stored in variable "scalegauss" and is simply added to the first element of array *x []* after it has been IFFT'd (i.e. after calling on the IFFT module). The procedure is repeated *N=128* times to generate one Gaussian RV for each element of array *x []* until all the elements have had "noise" added to them.

A note should be given as to how the Gaussian RV is scaled in proportion to the value of the elements in the array *x [].* Recall that the QAM values were scaled in section 4.3.4 in order to avoid loss in precision by having values to close together. Since our QAM values have been scaled higher, our noise values need to be scaled as well.

Before *addnoise* is called a program called *avg_nrg* is called in order to calculate the average energy of array *x []* which contains the IFFT'd QAM data. The average energy, which is the same as the variance σ, is calculated by taking the square of each element in array *x []* and then summing them over all "*N*" elements. The sum is then divided by "*N*" and is then further divided by *2* because *x [ ]* is made up of real and imaginary values, therefore we are finding the average energy of each complex pair.

Once the average energy is found it is stored in the variable *sig_es*. The *addnoise* function uses this value to properly scale the variance in proportion to the received signal value. The scaling is performed through "σ" in Eq. (5.1) which is found through Eq. (5.3) and stored in *scalesigma.* It is then used to find the scaled, Rayleigh random variable and finally the Gaussian random variable, which is then added to the *x []* array. In Eq. (5.3) *sig_es* is the variance, *sn_ratio* is the linearized signal to noise ratio.

$$scalesigma = \sqrt{\frac{sig\_es}{2 * sn\_ratio}} \tag{5.3}$$

5.1.3 Error Count

Error counting is executed by calling the ***err_chk*** function from ***main()***. The last step before a new set of data is passed through the OFDM system is to compare the output of the Viterbi decoder with the input that is stored in the *buffer* array. Because 15 bits must pass through the *state_history* array before the first bit can be decoded, a buffer array is needed at the input to serve as a system delay for bit comparison purposes.

A variable called *count* is assigned the task of keeping an error count for each of the 128 data points processed for every program run. The value of the variable is then passed on to *main()* and added to the variable *total*, which keeps a total error count for the programmed number of loops. The value of *total* is kept and divided by the total number of processed bits to obtain an error plot as described in the next section.

5.2 Error Performance Results

Error performance results were obtained with and without forward error correction. In order to test the system, the value for the variable *es_ovr_n0* which represents signal to noise ratio in decibels (dB) was varied from 0 to 14 dB for the system without FEC and 0 to 9 dB for the system with FEC as shown in Figure 42. The system was run long enough to generate approximately 100 errors at each SNR value. For example a BER of 10^{-2} would require approximately 10^4 or 10 000 generated bits to pass through the system in order generate about 100 errors.

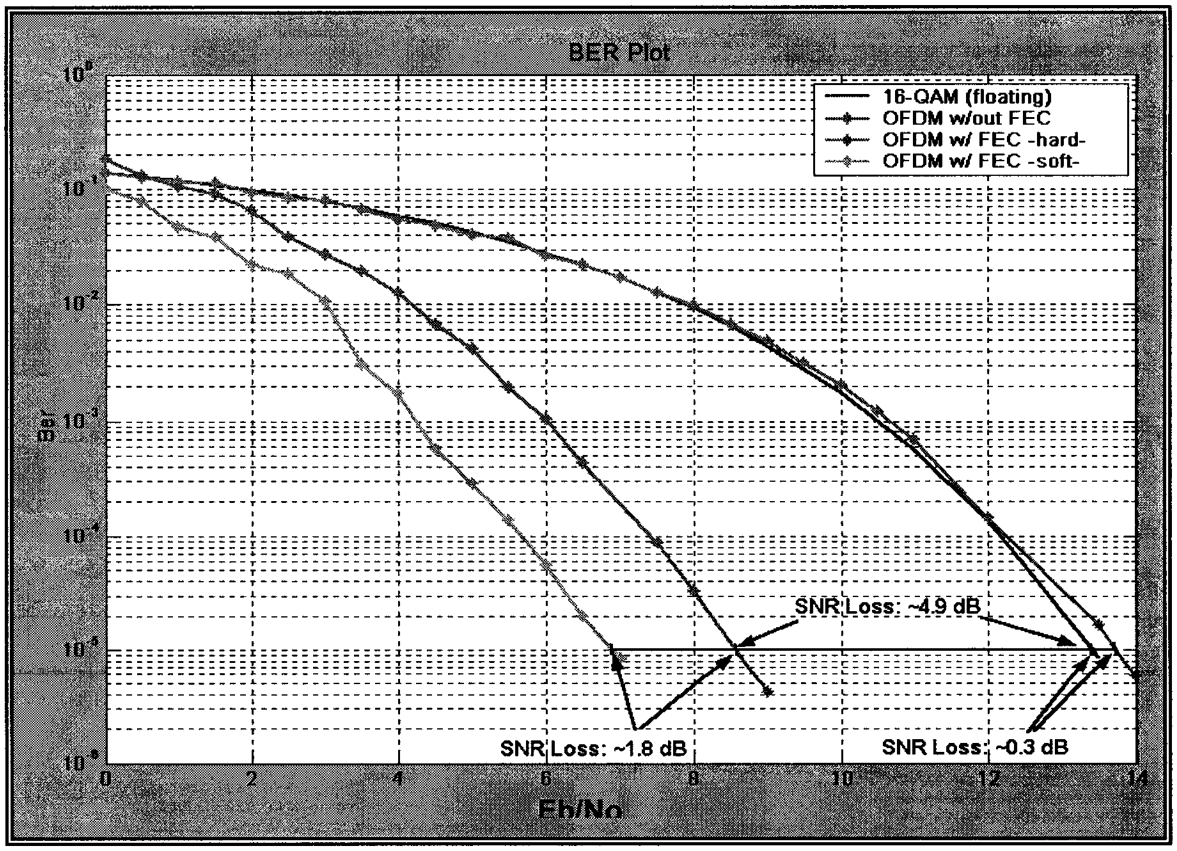


Figure 42: BER Plot of OFDM system

The rightmost curves of the BER plot of Figure 42 shows a comparison between a floating-point 16-QAM system (blue) and the OFDM system tested in this section without FEC (green) [28]. A direct comparison can be made between a 16-QAM system and this OFDM system because the only difference between systems lies in performing the IFFT, which is immediately followed by the FFT. The FFT reverses the IFFT operation therefore the BER will not be affected by these added operations. However, given that the OFDM system was created with fixed-point calculations to perform the FFT and IFFT operations, there will be a loss in signal to noise ratio. Figure 42 shows an SNR loss of about 0.3 dB at a BER of $1\text{x}10^{-5}$. This concludes that the proper use of scaling exhibits an adequate amount of system accuracy.

The system was also tested with the addition of the convolutional encoder (K=3) and both hard and soft-decision Viterbi decoding. The hard-decision plot is shown in magenta and shows a 4.9 dB gain over the floating-point 16-QAM system. The leftmost curve represents the BER of the system working with the soft decision regions. We observe an SNR gain of approximately 1.8 dB over the hard decision plot.

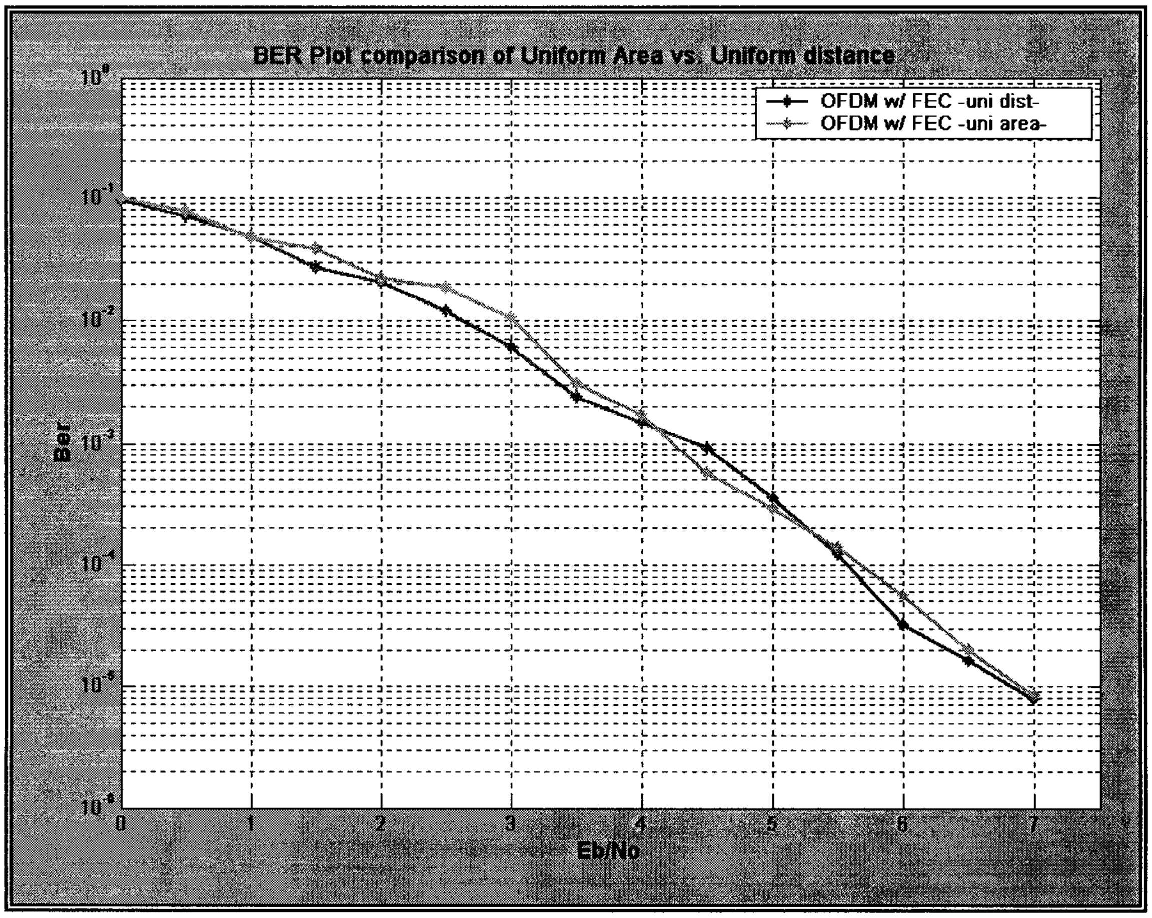


Figure 43: BER comparison of uniform area and uniform distance

Recall from section 4.3.7 that the soft decision regions were divided with uniform distance “D” (red) and then by uniform area (cyan). Figure 43 shows that there was no significant difference between using either of the methods to define the soft decision regions of the QAM constellation.

5.3 Optimization

The following section will describe optimization techniques used throughout the design process in order to help maximize overall throughput in the system.

5.3.1 Data Types

Careful consideration of data types should be given when dealing with the performance of the system. Specifically the "*int*" and "*short*" data types can have an impact on the system.

Whenever possible, the "*short*" data type should be used for multiplications because it makes the most efficient use of the 16-bit multiplier. A "*short * short*" multiplications would use one cycle versus five for "*int * int*". On the other hand the "*int*" should be used for loop counters instead of "*short*" in order to avoid unnecessary sign-extension instructions [54].

Both of the above guidelines were followed in the tested system and were found to improve performance.

5.3.2 Compiler Options

The CCS compiler allows the designer to invoke different optimization combinations that help to reduce code size and/or speed up code execution. Table 4 from section 4.1.1 demonstrates how efficient DSP compiler tools have become for some algorithms. Allowing the compiler build the assembly code for the designer proves advantageous because it reduces the workload and is less prone to errors compared to hand optimization. Hand optimization, if necessary, should only be used a last step in the design cycle.

Eliminating unused code and loop unrolling are some examples of optimization techniques used. A dramatic increase in speed was noted with the use of software pipelining. Software pipelining makes maximum use of the DSPs hardware resources by executing instructions in parallel [54].

5.3.3 Memory Management

Optimization from a memory management point of view comes from making maximum use of the fastest memories first. As described in 4.1.2, the EVM is designed with on-chip memory as well two types of external memory. The on-chip memory is divided into two individual 64 KB sections commonly titled “IPRAM” for program memory and “IDRAM” for data memory. A designer would ideally store all executable code on IPRAM and all variables and data on IDRAM. However, for larger programs there is the possibility of using the larger but slower external 256Kb SBSRAM module that runs at the full clock rate when the CPU is running at 133MHz or half the clock rate when the CPU is running at 160MHz. Finally there are two, separate 4 MB SDRAM modules that run at 100MHz [47].

The software suite allows the designer to manipulate the use of the available memory through the linker command file. The linker command file can be used for various other features to customize an application such as specifying object files or archive libraries. However, by properly defining the “MEMORY” and “SECTIONS” directives, the designer has the capability to “fine tune” the use of memory by assigning different parts of a program to different memory regions [55]. There is a choice of two memory maps, which define the starting address (in hexadecimal values) of each of the different memory types. The memory map is chosen by setting the correct DIP switches on the EVM [47]. These address values must be respected when programming the linker command file. When building the project, a .map file is created containing the memory usage. This file can be reviewed to select and optimum memory configuration. Once the designer is satisfied with the memory layout, the project can be compiled and run.

Specifically, for this project, the code size is approximately 79 KB, which takes up more than the available memory for IPRAM. The second choice can be to assign all of the application's code as well as the supporting library objects to the external SBSRAM. However, given that accessing external memory can use up more clock cycles, this option may prove unsatisfactory from a performance point of view. A more attractive, and efficient implementation would be to fit as much user code as possible on the IPRAM and place any supporting library objects that are used less frequently on the SBSRAM. In this manner all of the executable code is contained on the fast on-chip memory to maximize the instruction pipelining. All data, in the form of variables was subsequently stored on the fast on-chip IDRAM. A graphical representation of the memory layout is shown in Figure 44.

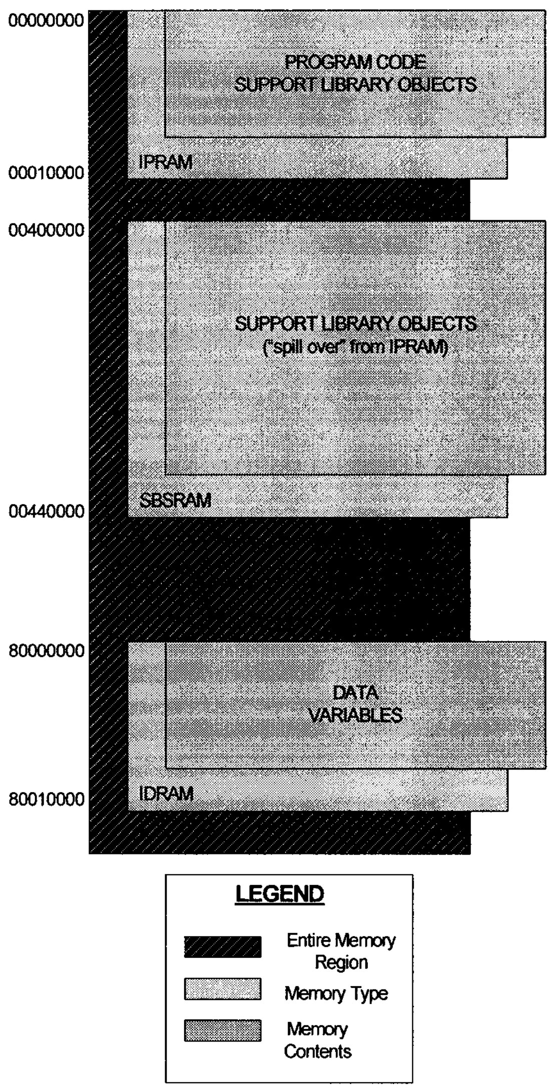


Figure 44: Graphical representation of memory layout

As mentioned previously, memory access speed to the SBSRAM will differ with the clock speed used. The SBSRAM device can be clocked at the CPU clock speed when operating at 133 MHz, or one-half the clock speed when operating at 160 MHz. Both combination were tested for best throughput results.

5.3.4 Code Profiling

Use of the Profile Clock allows the designer to acquire execution statistics of the code. Profiling basically gives an instruction cycle count of the profiled area, which can be as small as a couple of instruction lines or as large as an entire system [44].

A profile analysis can report how many cycles a particular function takes to execute and how often it is called. This functionality helps to pinpoint areas of code that take the most time to execute and can ultimately help eliminate performance bottlenecks.

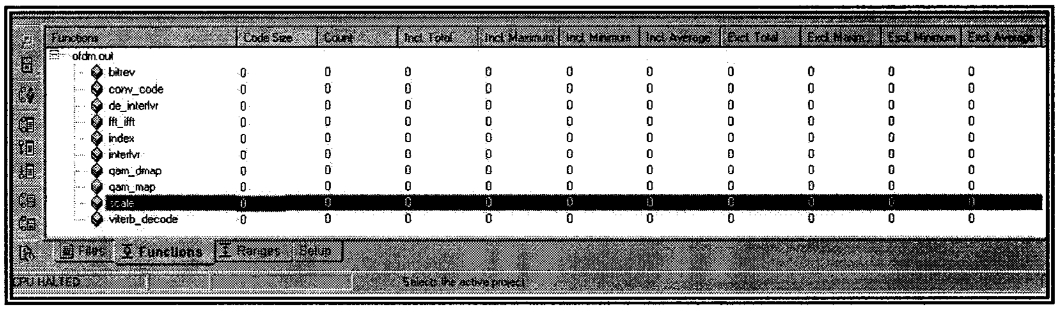
The above strategy was used to remove latencies from the system such as unnecessary calls to C-library functions. Instead, it was found that better performance results could be obtained by performing all known calculations beforehand and storing results in Look-Up Tables or variables.

5.4 System Bit Rate Measurements

Another important criterion for the OFDM system is its real time throughput. The real time bit rate can be measured in bits per second (bits/s) to offer an analogy of transmission rates for a communications system.

5.4.1 Profile Results

In order to measure the amount of bits that can be processed in a given time, the Profile Clock was used on the final system configuration. Figure 45 shows the profiler window taken from Code Composer Studio. The functions *coeff* and *index* were purposely left out, as these were only needed once at compile time. In addition *bitgen*, *addnoise*, *variance* and *err_chk* were also excluded, as these were only needed for assessing the system accuracy through BER plots.



Functions	Code Size	Count	Incl. Total	Incl. Maximum	Incl. Minimum	Incl. Average	Excl. Total	Excl. Maxim...	Excl. Minimum	Excl. Average
ofdm.out										
bitrev	0	0	0	0	0	0	0	0	0	0
conv_code	0	0	0	0	0	0	0	0	0	0
de_interlvr	0	0	0	0	0	0	0	0	0	0
fft_ifft	0	0	0	0	0	0	0	0	0	0
index	0	0	0	0	0	0	0	0	0	0
interlvr	0	0	0	0	0	0	0	0	0	0
qam_dmap	0	0	0	0	0	0	0	0	0	0
qam_map	0	0	0	0	0	0	0	0	0	0
scale	0	0	0	0	0	0	0	0	0	0
viterb_decode	0	0	0	0	0	0	0	0	0	0

Figure 45: Profile window from Code Composer Studio [44]

In order to calculate the maximum bit rate we use the total cycle count and an instruction cycle time of 6.25 ns when running at 160Mhz (or 7.52 ns when running at 133 MHz) to find the total time to run through the entire system. We then divide the number of bits processed by the total time to find the number of bits per second. Table 8 shows the total cycle count and time for the OFDM system run using a frame size of 256 bits. Note that forward error correction was not used therefore all data and code is located on the on-chip memory.

Table 8: Profile results for OFDM system with and without optimization (no FEC)

Function	**Instruction Cycles**	**Instruction Cycles**
	without Optimization	***with Optimization***
interlvr	15597	1328
qam_map	5027	510
fft_ifft (x2)	29074 (x2 = 58148)	4416 (x2 = 8832)
bitrev (x2)	6700 (x2 =13400)	1332 (x2 = 2664)
scale	4257	398
qam_dmap	12358	4065
deintrlvr	15597	1328
	Total cycles = 124384	**Total cycles** = 19125
Total Cycles * 6.25 ns.	**Total Time** = 777µs.	**Total Time** = 119 µs.
256 ÷Total Time	**Bit Rate = 330 Kb/s**	**Bit Rate = 2.15 Mb/s**

Table 8 shows a significant improvement in speed using the optimization techniques described in section 5.3.2. An increase of approximately 550% (from 330 Kb/s to 2.15 Mb/s) is achieved when using software pipelining in the build options. As mentioned, the FFT and IFFT operation are highly similar; therefore the same code can be used to implement each function. The *fft_ifft* and *bitrev* functions were called twice as shown in the table. The *fft_ifft* function, which is the most computationally intense function, was found to take the most cycles to complete and has least efficiency increase of all the functions.

A further increase can be estimated by removing the cycle count for the "*bitrev*" function. As noted in section 2.2.2, bit reversal is needed at the output of a DIF algorithm and at the input of a DIT algorithm. As stated in [24] there is no difference in complexity between the two algorithms only the order in which the input data should be fed, therefore the cycle count would be the same regardless of which algorithm is used to implement the "*ifft_fft*" function. If we used a DIF algorithm to perform the IFFT then output data would be in the proper order for an FFT function based on the DIT algorithm. In this case bit reversal is not needed and can be subtracted from the equation. This could potentially increase our bit rate to 2.49 Mb/s.

Using a code rate of ½ to implement the OFDM system with forward error correction causes the frame size to drop from 256 to 128 bits. The use of a soft-decision Viterbi decoder greatly increases the code size of the "*qam_dmap*" function, which consequently causes the system to outgrow the memory space available in IPRAM. Therefore the use of the SBSRAM device is needed to implement the larger system. Table 9 shows profile results for the OFDM system using the SBSRAM to contain all of the program code. The optimization settings are the same as in Table 8. The different clock rates and SBSRAM access times are compared.

Table 9: Profile results for OFDM system using different clock rates (with FEC)

Function	Instruction Cycles	Instruction Cycles
	using 133 MHz clock and full rate SBSRAM access	***using 160 MHz clock and half rate SBSRAM access***
conv_code	32802	40807
interlvr	13738	18567
qam_map	11191	14514
fft_ifft (x2)	35522 (x2 = 71044)	56452 (x2 = 136115)
bitrev (x2)	**X**	**X**
scale	7853	11872
qam_dmap	50192	65207
deintrlvr	13738	18567
Viterbi_decode	499273	692134
	Total cycles = 699831	**Total cycles** = 997783
Total Cycles * 1/clock ns.	**Total Time** = 5262 ms	**Total Time** = 6236 ms
128 ÷Total Time	**Bit Rate = 24.3 Kb/s**	**Bit Rate = 20.5 Kb/s**

Table 9 shows a considerable drop in throughput by placing all the code in external memory. As a comparison, a single *fft_ifft* operation increases from 4416 cycles in Table 8 to 56452 cycles in Table 9. The system throughput is further taxed by the addition of the *conv_code* and *viterbi_decode* functions. The *viterbi_decode* function takes up by far the most time to execute, proving that it is the most cycle intensive task of the system.

Table 9 shows that although the CPU is running at a lower clock rate, it performs better than a faster clock rate and slower SBSRAM rate. However, we can conclude that placing all of the code on the external SBSRAM slows the throughput to an unacceptable bit rate as shown by either column.

Using the technique described in 5.3.3, we can place certain, less used library objects on the SBSRAM while keeping all of the program code as well as the rest of the library objects on the IPRAM. This will allow us to use the faster clock rate of 160Mhz to perform all critical instructions. Only library objects related to the testing modules such as *bitgen* and *addnoise,* whose profile results are not counted, will be placed on the SBSRAM. Table 10 provides the profile results.

Table 10: Profile results of OFDM system using memory management

Function	**Instruction Cycles**	**Instruction Cycles**
	using 160 MHz clock and half rate SBSRAM access (no noise)	***using 160 MHz clock and half rate SBSRAM access (SNR of 0 dB)***
conv_code	2182	2182
interlvr	1328	1328
qam_map	510	510
fft_ifft (x2)	4416 (x2 8832)	4416 (x2 8832)
bitrev (x2)	**X**	**X**
scale	398	398
qam_dmap	9720	13027
deintrlvr	1328	1328
viterbi_decode	64387	64387
	Total cycles = 88685	**Total cycles** = 91992
Total Cycles * 1/clock ns.	**Total Time** = 554 μs	**Total Time** = 575 μs
128 ÷Total Time	**Bit Rate = 231 Kb/s**	**Bit Rate = 223 Kb/s**

A significant increase in the bit rate is noted by keeping the code on the fast on-chip memory and moving less important library objects to the external memory. In fact, all of functions, with the exception of *qam_dmap* return to the original values found in Table 8. The *qam_dmap* function has a higher cycle count in the full OFDM system when compared to the system without FEC because of the division into soft-decision regions, which extends the search time for the correct soft decision region. In addition, given the increased cycle count of the added FEC modules, the bit rate drops to 231 Kb/s as shown in the first column of Table 10. This profile result was taken without the added Gaussian channel to give a best-case scenario. The second column uses a very low signal to noise ratio for reasons in section 4.3.7. The extra processing time is demonstrated in the extra cycles used in the *qam_dmap* function. A signal to noise ratio representative of real world application would give a profile result somewhere between these two numbers.

5.5 Estimates

The following sections will detail estimates on modifications to system parameters and resources.

5.5.1 Modifications to System Parameters

Many practical systems using convolutional encoders as a means of forward error correction operate with a constraint length higher than $K=3$ used in the system tested here. The reason is that a higher constraint length can offer improved bit error rates than lower ones. An SNR gain of about 2 dB can be gained by increasing the constraint length from $K=3$ to $K=8$ [28]. However the cost for a better BER is increased memory consumption as well as increased cycle count for data processing. Both increases can be adequately estimated for an OFDM system performing with a constraint length of K=7.

The memory requirements are largely determined by the two-dimensional arrays used in the system, which are directly dependant on the number of states. As the number of states grow exponentially from $2^{K-1} = 4$ to $2^{K-1} = 64$ for a constraint length of $K = 8$ so do all the arrays that are dependant on this number. Table 11 gives a summary of the memory consumption in bytes assuming that all arrays are declared as "*short*" so that each array element occupies 16 bits. We also assume that the code rate remains 1/2.

Table 11: Memory requirements for different constraint lengths

		K = 3	K=7
	Number of States	***4***	***64***
Encoder	g [][]	12 bytes	28 bytes
	shift_reg[]	6 bytes	14 bytes
Decoder	input [][]	32 bytes	8192 bytes
	output [][]	16 bytes	256 bytes
	next_state [][]	16 bytes	256 bytes
	accum_err_metric [][]	16 bytes	256 bytes
	state_history [][]	128 bytes	4608 bytes
	state_sequence []	32 bytes	72 bytes
	TOTAL	**258 bytes**	**13 682 bytes**

From Table 11 we can deduce that the memory requirements increase by a factor of 53 to implement the system with constraint length K=7.

Cycle time can be estimated by calculating the amount of time spent in the function calls *conv_code()* and *viterb_decode ()*. However, since the time to encode is negligible with respect to the time to decode, only the increase in cycle time for *viterb_decode()* will be considered.

The increase in cycle time can once again be accounted to the additional states created for a constraint length *K=7*. The time spent in the Viterbi decoder algorithm can be broken down into three major loops titled: 1) Metric Calculation (MC) 2) Minimum Accumulated Error Search (MAES) and 3) TraceBack (TB). Loops MC and MAES, grow exponentially as a function of the constraint length *K*, whereas TB grows by a linear factor of *K*. Table 12 summarizes the increase in loop number from *K=3* to *K=7*.

Table 12 Comparison of loop values for constraint lengths, K=3 and K=7

Loop Name	Equation to find number of loops	Number of loops for K=3	Number of loops for K=7
MC	$2*2^{K-1}$	8	128
MAES	$½*2^{K-1}$	2	32
TB	5*K	15	35

The factor of *2* added to the equation for the MC loop calculation comes from the fact that loop is entered twice for each state, once for each binary input (i.e. "0" and "1"). A factor of ½ is added to the equation for the MAES loop calculation because the minimum accumulated error is found by comparing two state values at a time in each loop.

From Table 12 we see an increase by a factor of 8x for the MC and MAES loops and a factor of about 2.3x for the TB loop. Applying this estimation to the present OFDM system further reduces the bit rate. A solution is offered in the next section.

5.5.2 Faster Processor

The C6201 chip used in this project belongs to the C6000 performance line of TI's DSP chips. However, it is not fastest of the group. The C64xx line of chips offers much more performance in many respects. The fastest chip operates at 1.1 GHz allowing for 8 800 MIPS, which is a 5.5 factor increase over the C6201 chip operating at 200Mhz. Other benefits of the C64 architecture over the C62 are wider data paths, larger register file, and new instructions that support packed data processing. Because of these improvements, more work can be done in a clock cycle and fewer instructions are needed to perform the same operations thus increasing performance and reducing code size [56]. Ratios for cycle count improvements between the two architectures are shown in Table 13 for some commonly used algorithms.

Table 13: Cycle count ratio between C64 and C62 architectures [56]

Algorithm	Cycle Performance Improvement Ratio C64x:C62x
FFT- Radix 4 Complex	2.1x
Reed Solomon Decode (Forney)	3.2x
Viterbi Decode	2.7x

Using the 1.1 GHz C64 processor to execute all of the functions of the OFDM system as shown in the first column of Table 10, we can safely estimate bit rate of 1.59 Mb/s. However given the architectural improvements as described in the preceding paragraphs the actual bit rate should be greater.

In addition to the above, the C64 chip can use a Viterbi Coprocessor (VCP) to perform Viterbi decoding without using any of the CPU's resources therefore the decoding function can be performed in parallel with the rest of the system. Since the decoding algorithm would be handled by the VCP it will run in parallel with all other functions. Therefore we can separate the calculation of clock cycles. Once we have estimated the number of instructions cycles for both the Viterbi function and all other functions we simply take the larger of the two to estimate the systems overall cycle count.

An estimation for performing the Viterbi decoder function using the VCP alone can be achieved through the benchmarks given by TI's application notes and Eq. (5.4) [57].

$$for\ K=7:\ \left(\frac{72+2}{6}\right)\times\left(F+K-1\right) \tag{5.4}$$

Eq. (5.4) calculates the number of cycles needed for the Viterbi coprocessor to perform the decoding for a constraint length K, and frame size F. Choosing a constraint length of *K=7* and a frame size of *F= 256* encoded bits we get a cycle count of 3231. A VCP cycle is equivalent to four CPU cycles; therefore the final number must be multiplied by four in order to compare to operations processed by the CPU [56]. Our CPU cycle count thus becomes 12924 cycles.

Table 14 summarizes the increase in performance by separating the functions handled by the CPU and the Viterbi decoder, which is handled by the Viterbi coprocessor. Note that the *conv_code* function was also profiled for a constraint length of K = 7.

Table 14: Estimated profile results for system using C64 processor (with FEC)

Function	**Simulated C64x cycle count CPU Core**	**Estimated C64 cycle count VCP**
conv_code	6327	**X**
interlvr	1328	**X**
qam_map	510	**X**
fft_ifft (x2)	4416 (x2 8832)	**X**
bitrev (x2)	**X**	**X**
scale	398	**X**
qam_dmap	9720	**X**
deintrlvr	1328	**X**
viterbi_decoder	**X**	12924
	Total cycles: 28443	**Total cycles: 12924**
Total Cycles * 1/clock ns.	**Total Time: 25.9 μs**	
128 ÷Total Time	**Bit Rate: 4.94 Mb/s**	

More cycles are consumed by the CPU compared to the VCP therefore, the system timing would depend on the length of the greater latency. Using a 1.1 GHz processor, the bit rate would increase substantially to 4.94 Mb/s.

5.6 DSP Performance Analysis

The bit rate in the OFDM system without FEC was found to be 2.59 Mb/s. This was a dramatic increase over the same system profiled without optimization proving that the compiler can make very efficient use of software pipelining without having to resort to low level assembly language programming. However, when FEC is added the system bit rate drops considerably to 231 Kb/s in the best case scenario. This result is comparable to the results obtained from a similar OFDM system used for telemetry applications in [18]. However the throughput demonstrated by the DSP baseband modem in this thesis would prove to be too slow for mobile communications using systems based on 3G. Such systems in operation is Japan deliver typical data rates of approximately 384kb/s to the user and are expected to deliver data rates as high as 2Mb/s [58].

A performance estimate of 1.59 Mb/s can be achieved using TI's most powerful processor under the C64 architecture. However given that the C64 architecture is designed to be more efficient than that of the C62 in both processing speed and code size, the actual bit rate would be higher. Another feature of the C64 line of processors is the addition of a Viterbi coprocessor which is designed with enough flexibility to provide decoding capabilities using hard or soft decisions and in a number of constraint lengths and code rates. Allowing the VCP to take up the task of decoding, the bit rate can theoretically reach 4.94 Mb/s. This proves that an OFDM modem based on a DSP platform can meet the flexibility and performance needs of a Software Defined Radio used for mobile communications.

However, certain other issues should be taken into consideration. The C62 and C64 chips are the most performing chips and therefore the most power consuming. These chips would be ideally suited to perform tasks in base stations where power consumption is not as important as in mobile devices. The processors would prove to be too power consuming for mobile handhelds, which require more power efficiency.

The system tested above did not include a synchronizer. A real world application would require synchronization with the transmitter to ensure proper demodulation. This would require the addition of software modules to include pilot symbols on the transmitter portion and a synchronizer on the receiver portion. Synchronizers are known to be very cycle consuming. For example, certain algorithms implementing synchronizes are estimated to need approximately 2000 MIPS. However, this is more than is available in a dedicated C62 processor operating at 200 MHz [59].

6 Conclusion

The growth in the mobile communications industry demonstrates society's growing dependence on accessing information and communicating wirelessly. In order for this growth to continue, technology must be able to deliver cost effective networks that provide high performance transmission at high data rates. In addition, compatibility amongst different air interface standards is needed for seamless roaming amongst various types of networks. This thesis has presented a real-time implementation of an OFDM modem suitable for Software Defined Radios, which is thought to be an answer to the problem.

It was demonstrated that various research has been taking place in Software Defined Radios because it is believed that it can fulfill the need for building powerful and highly reconfigurable communication systems. A DSP platform was chosen to implement the OFDM modem. It was shown that a DSP platform provides an advantage over other competing technologies because of the ease through which systems based on such a platform can be designed and reconfigured. This coupled with numerous programming resources such as reusable, high quality code and very efficient compilers helps make the DSP platform a very interesting choice.

An industry leading company like TI provides a DSP platform with development tools such as a software suite and accompanying hardware, which greatly increase development time by coding in C or C++. In the implementation portion we were able to present an OFDM modem coded entirely in C and leave the compiler to convert the code into a format ready for downloading to the DSP target. We were able to discretize all software modules into blocks of code with specific tasks. These modules are highly modifiable given the nature of the C programming environment. All optimization techniques were implemented at the C code level through proper use of data types, as well as the implementation of cyclic shift registers and look up tables. It was demonstrated that a software design flow for DSP development removes the need for tedious and error prone assembly language coding.

Traditionally, DSPs have lagged in performance compared with competing technologies. However a staggering increase in throughput was found with proper coding techniques as well as a very efficient compiler when compared to program builds not using build optimizations. In one comparison, the compiler was able to increase throughput by 550% with the use of software pipelining which allows for much more instruction execution in parallel. This, along with the exponential increase in DSP performance over the last few years shows that DSPs are capable of more and more computationally intensive tasks.

In addition, a new trend in DSP design is to include the use of high performance coprocessors. This leads DSP designs to include dedicated hardware to perform tasks commonly used in telecommunications with great efficiency while leaving the CPU core to perform other tasks. It also increases performance of a DSP platform by further increasing the parallelism of specific processes in a system. The coprocessors are designed with enough programmability that they can still carry the advantage of a highly flexible platform. We were able to estimate a bit rate of 1.59 Mb/s using a faster processor and a further increase to 4.94 Mb/s when including a Viterbi coprocessor to handle the lengthy task of decoding. The above together with the predicted continuing trend of DSP speed results in a platform well suited for use in Software Defined Radios.

However, as mentioned, the DSP processors considered in this thesis are designed primarily for performance at the expense of power efficiency. Therefore their use would be limited to wireless infrastructure applications, such as in base stations, where power consumption is not generally an issue. The DSP processors considered in this thesis would prove to be too power consuming for mobile applications and products where power efficiency is a major factor.

As a direction for future works, development using newer processors with significant speed enhancements and better architectures should be attempted to see the actual throughput increases. In addition software modules performing cyclic extension, pilot insertion, and synchronization to complete the OFDM system should be added.

7 References

[1] "Key Global Telecom Indicators for the World Telecommunication Service Sector," *International Telecommunication Union (ITU),*www.itu.int/ITU-D/ict/statistics, 2001.

[2] J. Sundgot, "2 billion wireless subscribers by 2007", *Infosync World*, Aug. 7 2003.

[3] R Prasad, *Towards a Global 3G System: Advanced Mobile Communications in Europe Volume 1,* Boston: Artech House, 2001.

[4] R. Van Nee, R Prasad, *OFDM for Wireless Multimedia Communications*, Boston: Artech House, 2000.

[5] W. Webb, *Understanding Cellular Radio,* Boston: Artech House Publishers, 1998.

[6] M. Cummings, S. Haruyama, "FPGA in the Software Radio," *IEEE Communications Magazine*, February 1999, pp 108-112.

[7] A. Doufexi, S. Armour, A. Nix, M. Beach, "Design considerations and initial physical layer performance results for a space time coded OFDM 4G cellular network," *The 13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*, Sept. 2002, pp. 192 -196 vol.1.

[8] H. Shiba, T. Shono, Y. Shirato, I. Toyoda, K. Uehara, M. Umehira, "Software Defined Radio Prototype for PHS and IEEE 802.11 Wireless LAN," *IECE Trans. Commun.*, Vol E85-B, No. 12, Dec. 2002, pp. 2694-2702.

[9] S. J. Vaughan-Nichols, "OFDM: Back to the Wireless Future," *Computer, IEEE Computer Society*, Dec. 2002, pp. 19-21.

[10] J. Mitola, *Software Radio Architecture: Object-Oriented Approaches to Wireless Systems Engineering,* New York: John Wiley & Sons, 2000.

[11] M. Uhm, "The Maturing Movement Towards Heterogeneous Processing," *Compact PCI Systems*, December 2002.

[12] J. Eyre, J. Bier, "The Evolution of DSP Processors: From Early Architectures to the Latest Developments," *IEEE Signal Processing Magazine*, March 2000, pp. 43-51.

[13] J. Nikolic-Popovic, "Decoding Convolutional and Turbo Codes in 3G Wireless," *Texas Instruments Application Report SPRA878*, December 2002.

[14] S. Haruyama, *Wireless Communication Technologies: New Multimedia Systems*, Netherlands: Kluwer Academic Publishers, July 2000.

[15] K. Moessner, S. Hope, P. Cook, W. Tuttlebee, R. Tafazolli, "The RMA – A Framework for Reconfiguration of SDR Equipment," *IECE Trans. Commun.*, Vol E85-B, No. 12, Dec. 2002, pp. 2573-2580.

[16] Y. Suzuki, "Interoperability and Regulatory Issues around Software Defined Radio (SDR) Implementation," *IECE Trans. Commun.*, Vol E85-B, No. 12, Dec. 2002, pp. 2564-2572.

[17] M. Fahim Tariq, Y. Baltaci, T. Horseman, M. Butler, A. Nix, "Development of an OFDM based High Speed Wireless LAN Platform using the TI C6x DSP," *Communications, 2002. ICC 2002. IEEE International Conference on Communications*, May 2002, pp. 522-526

[18] F. Frescura, S. Andreoli, S. Cacopardi, E. Sereni, "An OFDM Radio Transmitter Based on TMS320C6000 DSP For Telemetry Applications," *International Conference Signal Processing Applications and Technology*, Oct 16, 2000.

[19] A. Cinquino, Y.R. Shayan "Real-time Software Implementation of an OFDM Modem Suitable for Software Defined Radios," to be presented in *IEEE Canadian Conference on Electrical and Computer Engineering,* May 2004.

[20] M. Beaulieu, *Wireless Internet: Applications and Architecture*, Boston: Addison Wesley, 2002.

[21] Y. Sun, "Bandwidth-Efficient Wireless OFDM, " *IEEE Journal On Selected Areas in Communications*, vol. 19, no. 11, November 2001, pp 2267-2278.

[22] I. Koffman, V. Roman, "Broadband wireless access solutions based on OFDM access in IEEE 802.16," *Communications Magazine IEEE*, Vol. 40, Issue 4, April 2002, pp. 96-103.

[23] D. J. DeFetta, J. G. Lucas, W. S. Hodgkiss, *Digital Signal Processing: A System Design Approach*, New York: John Wiley & Sons, 1988.

[24] A.V. Oppenheim, R.W. Schafer, *Discrete-Time Signal Processing*, New Jersey: Prentice-Hall, 1989.

[25] R.A. Haddad, T. W. Parsons, *Digital Signal Processing: Theory, Applications, and Hardware*, New York: Computer Science Press, 1991.

[26] R. Stewart, D. Garcia-Alis, "Concise DSP Tutorial", *Companion CD to SystemView Software*. 2001.

[27] C. Courtney, "Bit-Reverse and Digit-Reverse: Linear-Time Small Lookup Table Implementation for the TMS320C6000," *TI Application Report SPRA440*, May 1998.

[28] B. Sklar, *Digital Communications: Fundamentals and Applications Second Edition*, New Jersey: Prentice-Hall, 2001.

[29] A.B. Carlson, *Communication Systems: An Introduction to Signals and Noise in Electrical Communication*, New York: McGraw-Hill Book Company, 1968.

[30] S. Lin, *An Introduction to Error-Correcting Codes*, New Jersey: Prentice-Hall, 1970.

[31] B.E. Keiser, *Broadband Coding, Modulation, and Transmission Engineering*, New Jersey: Prentice Hall, 1989.

[32] J.G. Proakis, *Digital Communications Fourth Edition*, New York: McGraw-Hill, 2001.

[33] "IEEE Std 802.11a-1999(Supplement to IEEE Std 802.11-1999), Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications," *IEEE Standards*, September 1999.

[34] R.G. Winch, *Telecommunication Transmission Systems: Microwave, Fiber Optic, Mobile Cellular, Radio, Data, and Digital Multiplexing*, New York: McGraw-Hill, 1993.

[35] M.C. Jeruchim, P. Balaban, K. Sam Shanmugan, *Simulation of Communication Systems: Modeling, Methodology and Techniques; Second Edition*, New York: Kluwer Academic/Plenum Publishers, 2000.

[36] M.J.F. Garcia, J.M. Paez-Borallo, "Tracking of time misalignments for OFDM systems in multipath fading channels," *Consumer Electronics, IEEE Transactions on*, Vol. 48 Issue 4, Nov 2002 pp 982-989.

[37] H. Steendam M. Moeneclaey, "Analysis and Optimization of the Performance of OFDM on Frequency-Selective Time-Selective Fading Channels," *IEEE Transactions on Communications*, Vol. 47, No. 12, December 1999.

[38] S. Trautmann, N. J. Fliege, "A New Equalizer for Multitone Systems Without Guard Time," *IEEE Communications Letters*, Vol. 6, No. 1, January 2002.

[39] "SDR Primer", http://www.sdrforum.org/sdr_primer.html.

[40] P. Burns, *Software Radio Defined Radio for 3G*, Boston: Artech House, 2002.

[41] L. Pucker, "Paving Paths to Software Radio Design," *CommsDesign Magazine*, June 2001.

[42] A. Pandey, S. Agrawalla, S. Manivannan, "VLSI Implementation of OFDM Modem, *WIPRO Technologies White Paper*.

[43] Texas Instruments, *DSP Selection Guide 4Q 2003 (Rev. M)*, September 2003.

[44] Texas Instruments, *Code Composer Studio User's Guide SPRU296*, February 1999.

[45] Texas Instruments, *Code Composer Studio Tutorial SPRU301C*, February 2000.

[46] Texas Instruments, *TMS320C6000 DSP/BIOS User's Guide SPRU303B*, May 2000.

[47] Texas Instruments, *TMS320C6201/6701 Evaluation Module Technical Reference SPRU305*, December 1998.

[48] B. Gottfried, *Schaum's Outlines: Programming with C; Second Edition*, New York: McGraw-Hill, 1996.

[49] B. Hori, J. Bier, "Effective Fixed Point DSP Design for Low Cost Consumer Multimedia Appliances*", Iappliance,* June 2003,

[50] Texas Instruments, *TMS320C62X DSP Library Programmer's Reference SPRU402A*, April 2002.

[51] H. Schmidt, K. Kammeyer, "Quantization and its Effects on OFDM Concepts for Wireless Indoor Applications,*" 1st International OFDM Workshop*, Hamburg Germany, Sept. 1999.

[52] C. Fleming, "A Tutorial on Convolutional Coding with Viterbi Decoding", *Spectrum Applications.* http://pw1.netcom.com/~chip.f/viterbi/tutorial.html.

[53] Comtech AHA Corporation, "AHA Application Note; Soft Decision Thresholds and Effects on Viterbi Performance," *ANRS07_0203*, February 2003.

[54] Texas Instruments, *TMS320C6000 Optimizing Compiler User's Guide SPRU187K,* October 2002.

[55] Texas Instruments, *TMS320C6000 Programmer's Guide SPRU198G*, August 2002.

[56] Texas Instruments, TMS320C64x Technical *Overview SPRU395B*, January 2001.

[57] Texas Instruments, *TMS320C64x DSP Viterbi-Decoder Coprocessor (VCP) Reference Guide SPRU533C*, November 2003.

[58] "3G Wireless Communication FAQ," *Eurotechnology*, http://www.eurotechnology.com/3G/.

[59] J. J. Van de Beek, M. Sandell, and P.O. Börjesson, "ML Estimation of Time and Frequency Offset in OFDM Systems", *IEEE Transactions on Signal Processing*, vol. 45, no. 7, pp. 1800-1805, July 1997.