(50)

Simulation of a Multicarrier Demultiplexer

by

Michael A. Saginaw

Submitted to the Department of Electrical Engineering
and Computer Science in Partial Fulfillment of the
Requirements for the Degree of Master of Engineering in
Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 19, 1997

Author .....
Department of Electrical Engineering and Computer Science
May 19, 1997

Certified by .....
Gill A. Pratt
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Simulation of a Multicarrier Demultiplexer

by

Michael A. Saginaw

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science on May 23, 1997

Abstract

This thesis was part of the VI-A industry internship program at COMSAT Laboratories. A multicarrier demultiplexer was simulated in the SPW software environment. Patterns for manipulating data and performing fast Fourier transforms were generated and tested. The performance of the demultiplexer was evaluated in an additive white Gaussian noise environment.

Thesis Supervisor: Gill Pratt
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

This work was supported because COMSAT Laboratories participates in the VI-A program and in the VI-A Fellowship, which I gratefully acknowledge. I would also like to express my appreciation for being able to consult with the engineers, scientists, and managers at COMSAT, including Chris Cronin, Gil House, Greg Jividen, Mike Onufry, Soheil Sayegh, John Snyder, Anthony Watkins and especially Jim Thomas.

I would also like to thank all those at MIT who provided on-going support and encouragement: Gill Pratt, my thesis supervisor; George Pratt, my academic advisor; David Staelin, MIT's liaison to COMSAT; Markus Zahn, director of the VI-A program; and Anne Hunter, who is always looking out for students in the EECS department. Finally, I am grateful for the warmth and support of my family.

This work is based in large part on the knowledge gained from 4 years of class work at MIT and two previous summer work periods at COMSAT Laboratories. The classes most directly related to this thesis were 6.003 (Circuits and Systems), 6.341 (Discrete Time Signal Processing), and 18.310 (Discrete Applied Mathematics).

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

COMSAT Laboratories has a contract to design and build a multicarrier demultiplexer. The inputs to the demultiplexer unit are carriers that are multiplexed in frequency. The demultiplexer must separate the carriers, filter them, and pass them on to a demodulator.

1.1 Project Specifications

The inputs to the demultiplexer consist of carriers that are multiplexed in frequency. These carriers are transmissions of binary digital data. In order to transmit the data, the modulation scheme employed is quadrature phase-shift-keying (QPSK). The protocols of the carriers are defined by the INTELSAT Earth Station Standard (IESS) [IESS 308, IESS 309]. Some of the inputs may be digitized voice signals, some may be binary computer data signals, and some may be digitized video signals. The specifications for the unit are as follows:

1. 36 MHz bandwidth: This corresponds to half the bandwidth of a standard INTELSAT transponder.
2. Flexible frequency plan: The demultiplexer must be able to handle a wide variety of arrangements of carriers within the 36 MHz band. However, engineering constraints such as the specifications of the demodulator chip being used prevent the demultiplexer from handling every possible arrangement of carriers. This is acceptable.

3. Various data rates: The demultiplexer must be able to handle all INTELSAT Business Services (IBS) and Intermediate Data Rate (IDR) data rates from 64 kbits/s to 2.048 Mbits/s.

4. On-the-fly switching: The demultiplexer must be able to work with one frequency plan at one time ("at bat") and have another waiting in memory to put into operation ("on deck"). The specification on the performance of the switching is that continued carriers must be maintained uninterrupted. That is, the carriers that are not changed by a switch from one frequency plan to another should continue to be demultiplexed and demodulated without interruption. The kinds of switches allowed are adding carriers and deleting carriers. It is permissible for the demultiplexer and demodulator to take a few seconds to acquire any newly added carriers.

5. Use MCD-1 demodulator ASIC: This refers to COMSAT's MCD-1 demodulator application-specific integrated circuit (ASIC). It has already been designed, emulated, built, tested, and used in other projects. Each chip can demodulate 24 carriers simultaneously. The inputs and outputs are time-division multiplexed (TDM) digital data. The inputs are 8 bit, two's complement binary data corresponding to QPSK symbols. A unique feature of the chip is that the inputs do not need to arrive at an integer number of samples per QPSK symbol. Rather, they need to be between 2 and 4 samples per symbol. The MCD-1 then uses COMSAT's patented Detection Sample Estimation (DTSE) algorithm in the demodulation process. The demodulator chip has already been shown to successfully handle on-the-fly switching successfully, with no interruption of continued carriers.

1.2 Design Approach

After considering other architectures to implement the demultiplexer, COMSAT engineers developed a design using the technique of fast convolution with a shared FFT. An analog front end down-converts the input signal to baseband frequencies and an analog-to-digital (A/D) converter feeds digital input to the demultiplexer.

The demultiplexer itself is entirely digital. First, it performs a fast Fourier transform (FFT) to convert all the carriers of the input to frequency coordinates. Then it uses filters in frequency coordinates to separate and shape the carriers. Finally, it performs several inverse fast Fourier transforms (IFFTs) to convert the carriers back to time coordinates. To make sure these IFFTs are of sequences whose lengths are powers of 2, the filtered points are zero-padded. The amount of zero-padding is chosen in order to accommodate the specifications of the demodulator, which requires that its input be between 2 and 4 samples per QPSK symbol.

The FFT and IFFT operations are performed by seven chips made by the company Butterfly DSP. These are the BDSP 9124 chips. In order to work with a stream of input data, the demultiplexer takes FFTs of overlapping chunks of the input. At the output, 50% of the samples must be discarded. Each demultiplexer unit is designed to handle a 9 MHz bandwidth. Engineers at COMSAT found that this is an excellent match to the demodulator ASIC, which can handle 24 channels. Four demultiplexer and demodulator units put together are capable of handling a 36 MHz bandwidth.

1.3 Accomplishments of Thesis

Given the hardware architecture as designed by John Snyder, the purpose of this master's thesis was to simulate the demultiplexer hardware. The accomplishments of the thesis are as follows:

1. Simulation of hardware: The hardware architecture was simulated down to the level of each bit using SPW and C source code for BDSP 9124.
2. Generation of patterns for transforms: In order to apply the mixed radix FFT and IFFT algorithms to the specific BDSP 9124 hardware, all the patterns were developed for 50% overlap addressing, digit reversal, twiddle coefficients, reshuffling addresses, and 50% discard addressing.
3. Generation and verification of timing and control signals: All the timing and control signals for the BDSP 9124 chips and the memory chips were simulated and verified.
4. End-to-end simulation: The data was carefully simulated at each step of the communication link, from the generation of baseband data for each carrier, to the modulation and multiplexing processes, to the workings of the demultiplexer. The simulated output of the demultiplexer was fed into the emulator for the demodulator. Finally, the demodulator's output was tested against the original input data and bit error ratio (BER) curves were plotted as a function of signal-to-noise ratio.
5. Verification of bit precision adequacy: Since the BER curves were acceptable, the decisions about how many bits of precision to use at each stage of the project were verified as adequate. This includes the A/D converter precision, internal processing stages, and coefficient precision for the FFT and IFFT. The analysis was done as a function of input white Gaussian noise.

1.4 Structure of Thesis

This thesis describes technical background and discusses simulations that have been done so far. It consists roughly of three parts.

Part 1 is background material. Chapter 2 describes relevant ideas in communications. Chapter 3 discusses alternative architectures for the demultiplexer and describes why the fast convolution approach was taken. Having motivated the fast convolution architecture, chapter 4 describes the relevant ideas in digital signal processing and relates them to the demultiplexer project and the INTELSAT carriers being accommodated.

Part 2 is about software and algorithms. Chapter 5 is about the SPW software used for the simulation, as well as the C source code for the BDSP 9124 chips. chapter 6 is about applying the mixed radix FFT algorithm to BDSP 9124 chips and includes a discussion of addressing patterns for the transforms. While Chapter 6 is about the FFT algorithm and the BDSP 9124 chips, chapter 7 is about how the algorithm and the chips fit together to achieve the demultiplexing operation.

Part 3 is about the simulated data at various stages of the simulation, from the baseband data for individual carriers, to the modulated and multiplexed data that enters the demultiplexer, to the results from the demodulator emulation. Chapter 8 describes the procedure for generating baseband data for each carrier and combining data from many carriers to form the input to the demultiplexer. Chapter 9 is about the demultiplexer simulation, including the 50% overlap buffer, the FFT, the filter, the IFFT, and the 50% discard at the end. Chapter 10 is about the end-to-end simulation and the BER curves. Chapter 11 is about remaining items and things that need to be worked on.

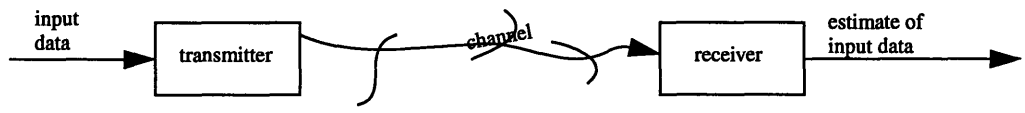
Chapter 2

Communications Background

2.1 System Overview

The task of a communication system is to transmit a message over a noisy channel to a receiver, as illustrated in broad overview in Figure 2.1 and in some more detail in Figure 2.2. For overseas communication, television broadcasts to large areas, and communication in many countries which don't have advanced wired infrastructures, satellite communication is heavily used.

Figure 2.1: System Overview



Data from a source is encoded and modulated by a transmitter. It is sent over a noisy channel. The receiver's job is to estimate what the original signal was.

INTELSAT is the International Telecommunications Satellite Organization, with 135 member nations. INTELSAT provides voice, video, and other services, mostly in digital format, where the input to the communication system is a bit stream at a fixed information rate. Most of the information rates in the INTELSAT system are multiples of 64 kbit/s.

Each member nation that signed the treaty creating INTELSAT is a signatory to INTELSAT. In the United States COMSAT Corp. is the designated representative and acts, under the direction of the U.S. Department of State, as the U.S. Signatory to INTELSAT. COMSAT Laboratories performs research, development, technical support,

and consulting for COMSAT Corp. and for a variety of national and international companies and government organizations.

2.2 Transmitter

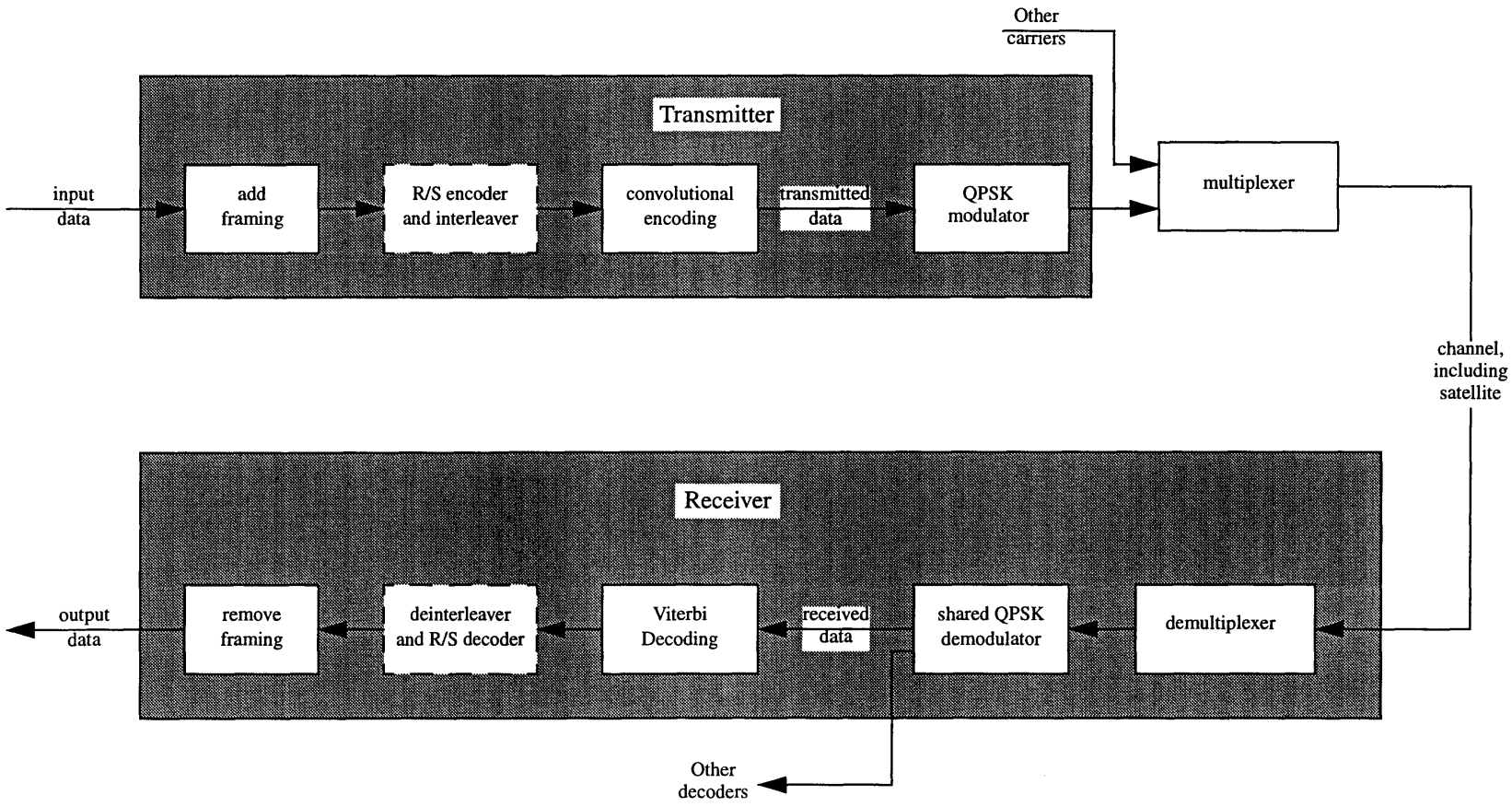
The job of the transmitter is to add codes to the input, modulate the signal, and then send it over a channel, which is noisy. Figure 2.2 shows the major blocks in the transmitter and the corresponding blocks in the receiver.

The most widely used modulation in the INTELSAT system is quadriphase shift-key (QPSK) modulation. It is a method of transmitting digital data over, for instance, a satellite link. In this discussion, the manipulations of a QPSK modulator are described and the baseband spectrum of the QPSK signal briefly discussed. Then the undesirable phenomenon of intersymbol interference (ISI) is discussed, and the filters used to minimize ISI are described. Then the QPSK modulator is described in more detail, including the techniques by which the signals are shifted to radio frequencies. The output of the QPSK modulator is characterized in time and frequency.

2.2.1 Baseband QPSK

For each pair of bits in the input data, a QPSK modulator produces one symbol for transmission. This symbol can have one of four phases, providing a way for the demodulator to determine what pair of bits was sent. In the study of QPSK modulation at baseband frequencies, each bit may be considered independently. However, the actual block diagram of the QPSK modulator, which does not operate only at baseband frequencies, will show how the two bits are combined to form one symbol.

Figure 2.2: System Block Diagram



Data from a source is tagged with framing information. It is augmented with forward error correcting (FEC) codes. Sometimes a Reed/Solomon (R/S) code is used as an outer FEC code. A convolutional FEC code is always used. Then the digital data is sent to a QPSK modulator and transmitted over a noisy channel.

The input data is a series of non-return to zero (NRZ) pulses. A "0" bit in the input data is converted by analog electronics to have a voltage of $+A$ and a "1" bit is converted to have a voltage of $-A$.

To understand the spectrum of this signal, it is helpful to use mathematical representation with box cars and impulses. The input data can be written as the convolution of a box car with an sequence of impulses. An impulse corresponding to a positive amplitude integrates to $+A$, and an impulse corresponding to a negative amplitude integrates to $-A$.

With this representation, the spectrum of the baseband signal can be analyzed. The convolution in time corresponds to a multiplication in frequency. The transform of the box car is a sinc function $\left(\text{sinc}(x) \equiv \frac{\sin(\pi x)}{\pi x}\right)$. Thus the envelope of the transform of the signal is a sinc function, and it extends infinitely across the entire spectrum. If the signal were shifted up to higher frequencies, its tails would still spread across all frequencies.

However, most of the energy of the signal is concentrated over a small range of frequencies. In order to enable many carriers to be sent and to make an environment where frequency-domain multiplexing is possible, the signals are deliberately band-limited. However, the band-limiting must be done very carefully in order to make a signal that the receiver can still decipher.

2.2.2 Intersymbol Interference

In general, passing a data signal through a band-limiting filter causes intersymbol interference (ISI) because the signal is convolved with the impulse response of the filter. The output signal at any one time has contributions from the input signal at many times. Energy from one symbol is smeared into the time-region for other symbols, and there is no way for the receiver to recover the data.

One System That Minimizes ISI

One system that minimizes ISI is a 1/sinc compensating filter followed by a brick wall filter, as illustrated in Figure 2.3.

There are three steps in designing this system. First, the operation of the electronics must be known. Second, the 1/sinc compensating filter must be designed, with its frequency response based on the timing of the electronics. Third, the brick wall filter must be designed, with its impulse response based on the timing of the electronics.

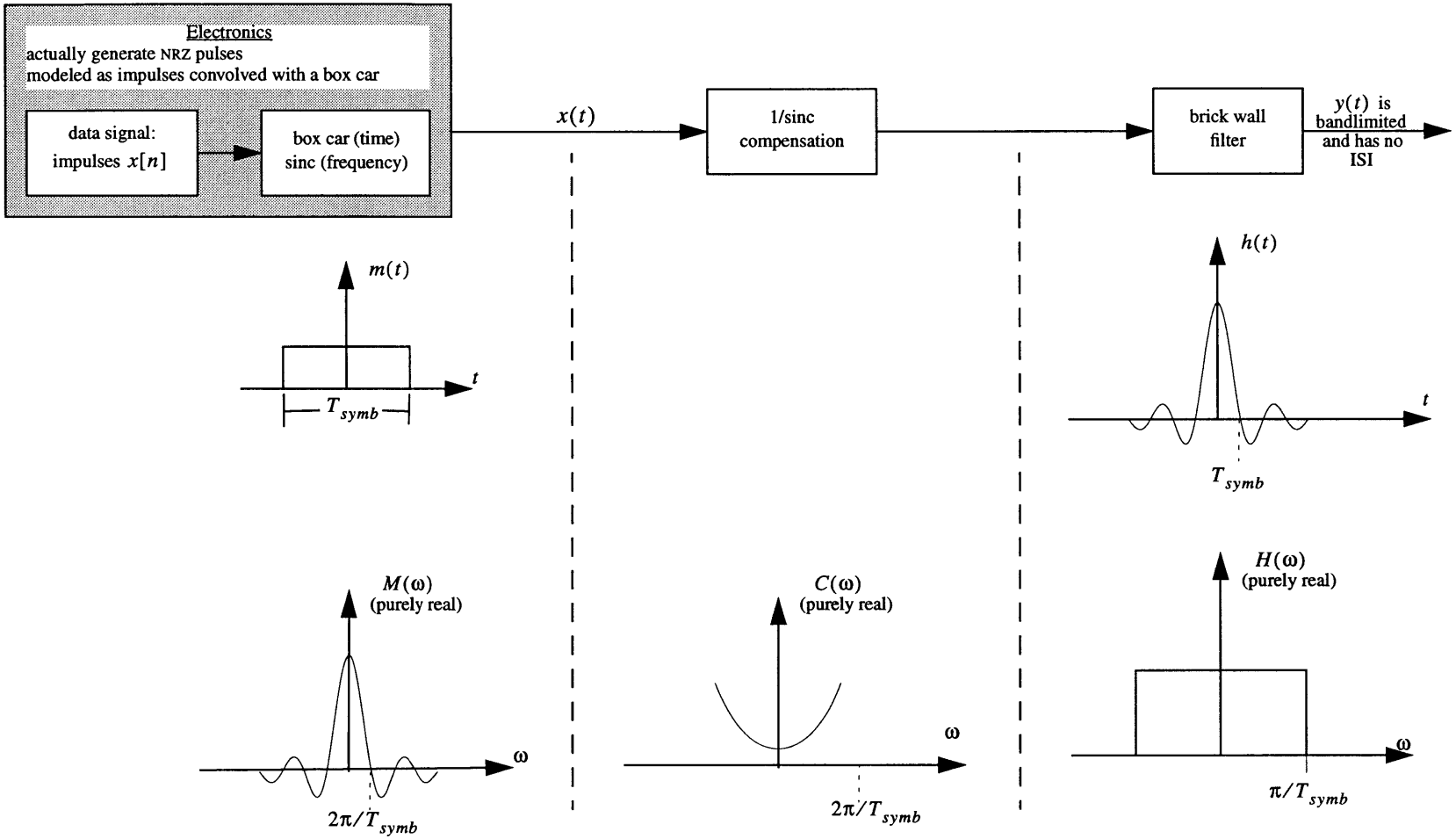
First, the electronics outputs NRZ pulses. These can be modeled as an impulse train (not necessarily periodic) convolved with a box car. The impulses are spaced T_{symb} apart, and the box cars span a period of T_{symb}. In frequency, the output is the transform of the impulse train times a sinc function, which is an envelope. The continuous time Fourier transform of the box car is:

$$M(\omega) = \int_{-T_{symb}}^{T_{symb}} e^{-j\omega t} dt$$

After integrating, substituting in the evaluation points, and combining the complex exponentials, this is seen to be a sinc function:

$$M(\omega) = \frac{2 \sin\left(\frac{\omega T_{symb}}{2}\right)}{\omega}.$$

Figure 2.3: Minimize ISI with 1/sinc Compensation and a Brick Wall Filter



The signal, , consists of NRZ pulses that come from the electronics. The 1/sinc filter makes it as though there were impulses, not NRZ pulses. The 1/sinc filter only needs to operate over a limited bandwidth, so it is achievable. The brick wall filter is designed so that $y(m \cdot T_{symb})$ can be used as the detection point since it only has contributions from $x(m \cdot T_{symb})$ (for all integers m). Therefore the input signal, $x(t)$, can be bandlimited to form an output $y(t)$, in such a way that the input bits can be recovered from $y(t)$.

Second, the 1/sinc compensating filter must be designed. The purpose of this filter is to get rid of the effect of the sinc envelope in frequency from the NRZ signals.

$$C(\omega) = \frac{\omega}{2\sin\left(\frac{\omega T_{symb}}{2}\right)}$$

Although this function goes to infinity, the third part of the design will place limits on the extent of the frequency variable for the 1/sinc compensation filter. The 1/sinc filter only needs to have its 1/sinc shape over a limited bandwidth, because after that the low-pass filter has such a small amplitude that it doesn't matter what the 1/sinc filter does at those frequencies.

Third, based on spacing of T_{symb}, the brick wall filter can be designed. Its impulse response is a sinc function whose periodic zeros must have the right spacing. The following Fourier transform pair does the trick:

$$\frac{\sin\left(\frac{\pi t}{T_{symb}}\right)}{\pi t} \leftrightarrow \text{Brick Wall with cutoffs at } \pm\frac{\omega_{symb}}{2}$$

Due to the periodic zeros in time, there is one point when the output depends only on the input from one instant and there is no intersymbol interference. The time when the receiver must look is called the detection point, and there must be clock synchronization between the transmitter and the receiver so the receiver can know when to look. Although this system works in theory, the brick wall filter is extremely sensitive to timing jitter (imperfect synchronization between the clocks of the transmitter and receiver.)

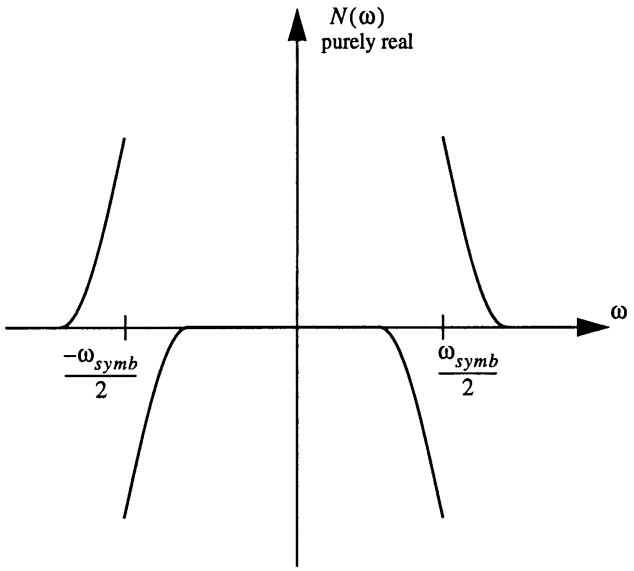
A More Robust System That Minimizes ISI

Fortunately, there are also other 0 ISI filters besides the brick wall filter. Any filter with

the anti-symmetric properties shown in Figure 2.4 also has 0 ISI. This theorem, due to Nyquist, is discussed in [Feher]. The impulse response is computed by taking the inverse continuous time Fourier transform of the frequency response. Using the anti-symmetry properties of the filter, it is possible to show that the impulse response has periodic zeros.

The raised cosine filters are a family of 0 ISI filters. They are formed by adding the frequency response of a brick wall filter to an anti-symmetric frequency response, and the resulting shape is shown in Figure 2.5. The impulse response of the raised cosine filters has 0 ISI because it is equal to the sum of two impulse responses with 0 ISI.

Figure 2.4: Another Type of 0 ISI Filter

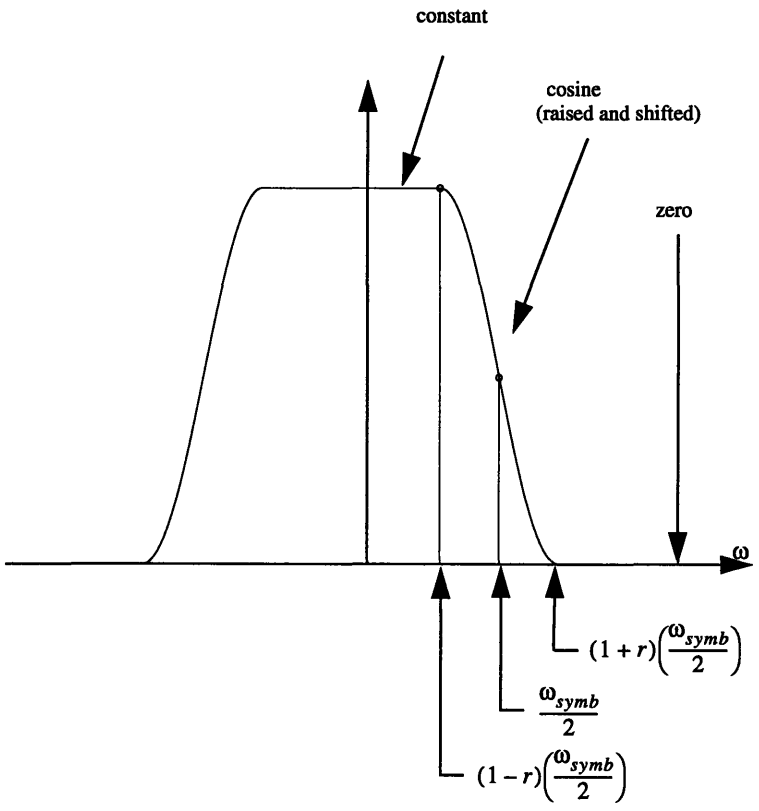


This filter has 0 ISI, as is shown in [Feher] by computing the impulse response. The sum of this frequency response and that of the brick wall filter is the raised cosine filter, which also has 0 ISI.

The raised cosine filters are not as sensitive to timing jitter as the brick wall filter. In fact, there is a trade-off between excess bandwidth and robustness in the face of timing jitter. For the INTELSAT carriers this demultiplexer must accommodate, the rolloff factor, r, is 0.4, meaning that 40% excess bandwidth is used beyond the bandwidth of the

brick wall filter. In practice, the filters are split into a square root raised cosine filter at the transmitter and an identical filter at the receiver. That way, the transmission is over a limited bandwidth and the receiver eliminates out-of-band noise. The demultiplexer must implement this square root raised cosine filter. These filters can be achieved to a high degree of approximation using either analog or digital filters. See [Poklemba], for example.

Figure 2.5: The Raised Cosine Filter

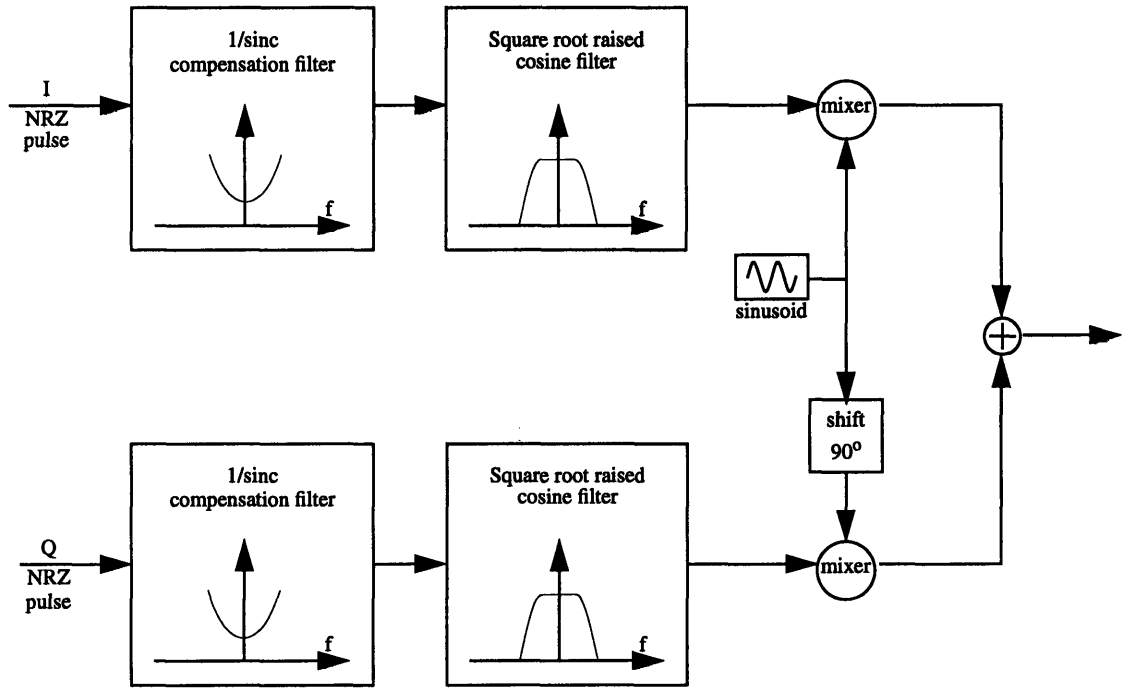


There is a family of raised cosine filters. For the INTELSAT carriers serviced by the demultiplexer, the rolloff parameter, r, is 0.4. They are known as 40% raised cosine filters, because they occupy 40% excess bandwidth beyond the underlying brick wall filter.

2.2.3 QPSK Modulator with Filters to Minimize ISI

A block diagram of a QPSK modulator is shown in Figure 2.6. The modulator combines two bits into one symbol for transmission.

Figure 2.6: Block Diagram of a QPSK Modulator

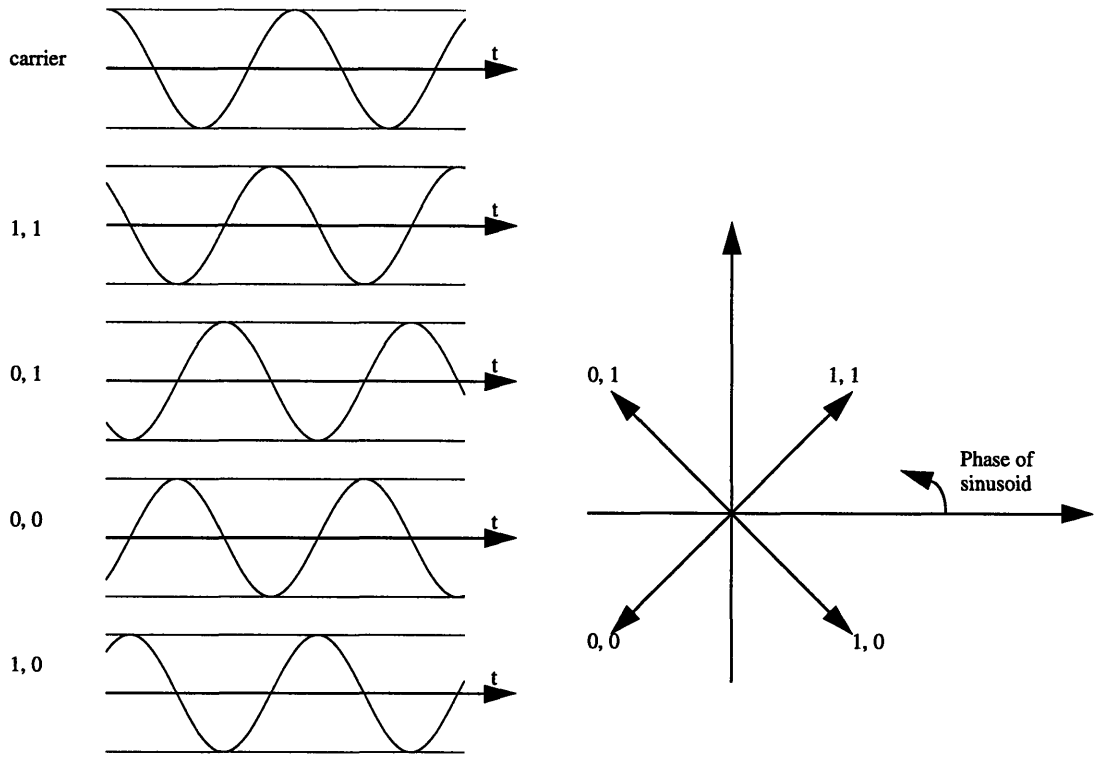


In the QPSK modulator, two bits are combined into one symbol and frequency-shifted up from baseband. When an input bit is 0, it gets mapped to -1 for I and Q. Input bits of 1 get mapped to 1 for I and Q (i.e. the circuitry makes the binary signal bipolar).

Phase Space Interpretation

Two bits are represented by one symbol, so there must be four different symbols. The characteristics of the symbol are best understood by ignoring the filters upon first examination, and are presented that way in Figure 2.7.

Figure 2.7: Time and Phase Plots of the Mapping in QPSK Modulation



Each pair of bits gets encoded as a sinusoid. The information about which pair of bits was sent is in the phase of the sinusoid. On the left is the signal that actually gets sent out. On the right is a representation of the mapping in phase space, where each vector is a sinusoid rotating at a frequency corresponding to the frequency of the modulator's oscillators. The phase of the vector has the information about which pair of bits was sent. The demodulator's task is to recover that phase.

In a time plot, each QPSK signal is a section of a sinusoid. The information about which pair of bits was sent is contained in the phase of that sinusoid, as compared to the phase of the carrier. The demodulator has a carrier recovery loop, and then proceeds to compare the signal's phase to the carrier's phase in order to recover the input bits.

Another helpful way to think of QPSK modulation is in terms of phase space. The idea is that a pair of bits gets mapped to one of four points in phase space. The signal itself is a vector rotating at a rate given by the frequency of modulation. The starting position of the vector is its phase. In the phase plot, the decision region for each symbol is seen to be formed by the two axes.

Spectrum of QPSK Signals

From the block diagram of the QPSK modulator in Figure 2.6, the spectrum of the QPSK signal can be explored. This provides the way to calculate how much bandwidth each QPSK signal occupies, and how the spectrum should be allocated for multiplexing many carriers in frequency.

First, consider the spectrum as though the compensation filters and the data shaping filters were in Figure 2.6 were not present. Each branch of the QPSK modulation scheme would consist of a non-return-to-zero (NRZ) pulse times a sinusoid. The NRZ pulse can be analyzed as an impulse train convolved with a box car. Thus the whole signal in time is as follows:

(impulse train * box car) • sinusoid

In frequency, that is:

(transform of impulse train • sinc)*two impulses

Thus in frequency, there are two replications, one at the positive frequency and one at the negative frequency of the sinusoid. At each, the envelope for the spectrum is a sinc function. When the filters are added, the envelope becomes band-limited, but in a way that will not cause ISI.

2.2.4 Carrier Specifications for the Transmitter

In the transmitter, each block that adds codes affects the bit rate of the transmitted data. Table 2.1 lists the carrier specifications, including overheads, for the carrier types that were simulated.

Table 2.1: Carrier Specifications

Info Rate (kbits/s)	Service Type	Frame Type	Framed Rate (kbits/s)	FEC Specs*	Trans. Bit Rate (kbits/s)	Trans. Symbol Rate (ksymb/s)	Occupied Bandwidth (kHz)	Allocated Bandwidth (kHz)
384	IDR	IBS	409.6	0.75/R	614.4	307.2	368.6	382.5
768	IDR/IBS	IBS	819.2	0.75	1,092.3	546.1	655.4	787.5
1544	IDR	IDR	1,640.0	0.75	2,186.7	1,093.3	1,312.0	1552.5
2048	IDR	IDR	2,144.0	0.50	4,288.0	2,144.0	2,572.8	2992.5

* Reed-Solomon Outer Coding also used; increases transmitted bit rate for the same information rate

Framed Rate

For many carriers in the INTELSAT system, framing signals are added according to he INTELSAT Business Services (IBS) or Intermediate Data Rate (IDR) specifications. The framing adds overhead to the signals. A signal with IBS framing must be transmitted at 16/15 times the information rate. The exception to this figure is for signals with an information rate of 1544 kbits/s, in which case the framed rate is arbitrarily set equal to the framed rate for a carrier with an information rate of 1536 kbits/s. For an INTELSAT IDR carrier, framing adds a fixed overhead of 96 kbits/s. [IESS 308, IESS 309, Snyder].

FEC Specifications

Forward error correcting (FEC) coding is added to improve bit error rates at a given signal-to-noise ratio. As with the framing protocols, the coding adds overhead, and it is necessary to boost the transmitted bit rate accordingly.

One layer of coding is always used: convolutional coding with Viterbi decoding. For a "Rate 1/2" convolutional code, the entire signal must be transmitted at 2 times the information rate while for a "Rate 3/4" convolutional code, that overhead ratio is 4/3 [IESS 308, IESS 309, Snyder].

Sometimes an outer layer of Reed-Solomon (R/S) coding and interleaving are also used for better performance. This is an apt combination because the Viterbi decoder tends to make burst errors, and the deinterleaver and R/S decoder are good at correcting burst errors. There is additional overhead from the R/S coding. The ratio is 126/112 for information rates up to and including 1024 kbit/s, 225/205 for an information rate of 1544 kbit/s, and 219/201 for an information rate of 2048 kbit/s [IESS 308, IESS 309, Snyder].

Transmitted Rates

The transmitted bit rate is computed by multiplying the information rate, the ratio for protocol codes, and the ratio for all the FEC codes. In QPSK modulation, where each symbol encodes two bits, the symbol rate is half the transmitted bit rate.

Bandwidths

The occupied bandwidth of the QPSK signal is determined by the type of raised cosine filters used. For these INTELSAT carriers, the filters are 40% raised cosine filters. INTELSAT documents say that the occupied bandwidths for these carriers is 1.2 times the symbol rate.

The allocated bandwidths for the carriers is the smallest multiple of 22.5 kHz that is at least as large as 1.4 times the symbol rate. The allocated bandwidths for carriers with R/S coding is the same as for otherwise identical carriers without R/S coding. [IESS 308, IESS 309].

2.3 Channel

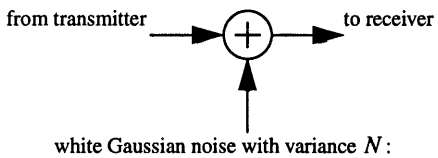
2.3.1 Channel Model

In satellite communications, the channel consists of the atmosphere and space between the ground station and the satellite for the uplink, and between the satellite and the other ground station. The signal from the transmitter is degraded as it travels over the channel. It is attenuated due to spreading and atmospheric absorption, and noise is added to it.

Spreading occurs because an earth station or satellite broadcasts energy out into space, and only some of it is aimed directly at the receiver. The rest spreads out into space and is lost.

The vast majority of thermal noise occurs at the input to the antennae. At this location, there are two sources of thermal noise. The first is the out-of-band noise picked up by the antenna from the sky. The second is due to the random motion of atoms in the antennae and in the electronic circuits [Feher]. By using the central limit theorems from probability theory, it is surprisingly simple to summarize the effect of all this thermal noise with a single parameter. The idea is that a random variable can be used to describe the amount of thermal noise that each atom has on the signal. The sum of all these random variables is another random variable, and a central limit theorem justifies the conclusion that the probability distribution of that sum can be approximated by a Gaussian random variable. The mean value of the noise is 0 and the only parameter is its variance. Since the noise is an independent process, its autocorrelation is an impulse and its power spectral density is a constant (i.e. it is white noise).

Figure 2.8: Channel Model



white Gaussian noise with variance N:

$$f_z(z) = \frac{1}{\sqrt{2\pi N}}\left(e^{\left(\frac{-z^2}{2N}\right)}\right)$$

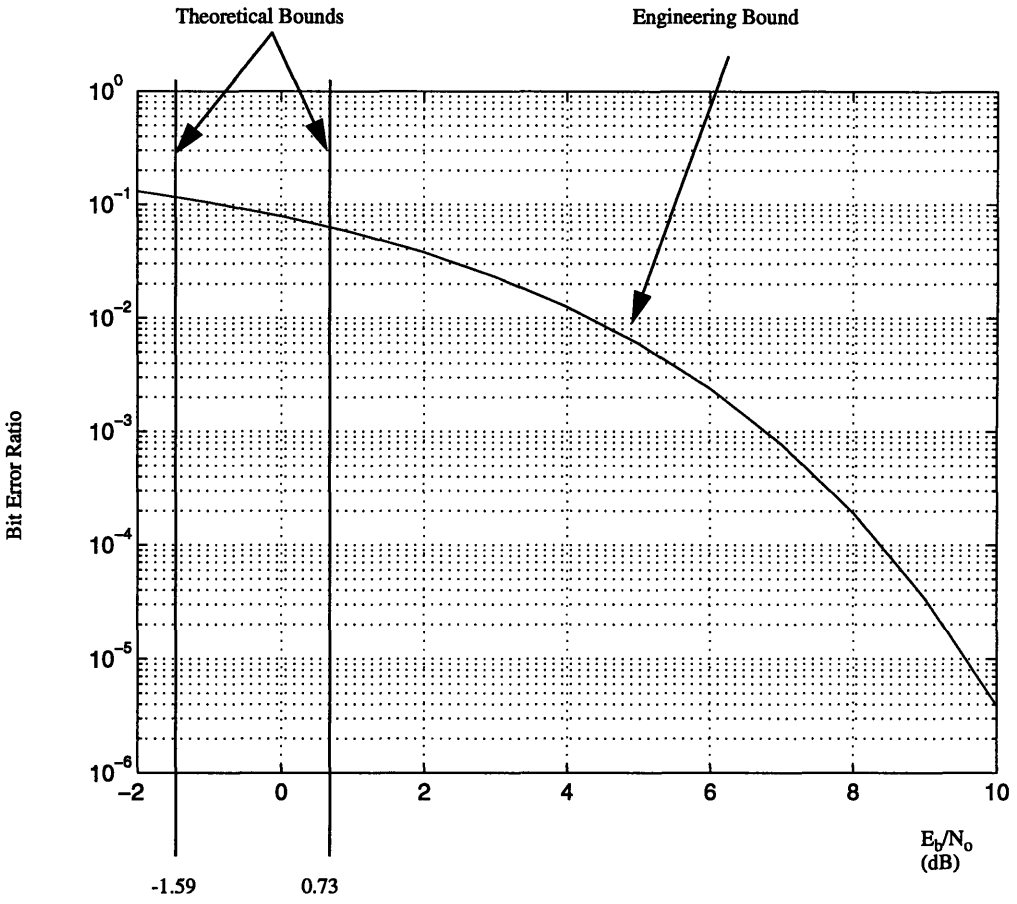
Based on one of the central limit theorems of probability, the fact that the spectrum of the noise is typically much broader than the spectrum of the signal, and the fact that the noise is an independent process, a good model of the channel is that it has additive white Gaussian noise.

It is important not to take the concept too literally. Pure white noise has an infinite amount of energy, and the noise in real systems does not. However, the power spectral density of the noise in real systems is usually much broader than the signal of interest, and it is approximately flat across the band of interest.

2.3.2 Probability of Error for QPSK Modulated Signals

Claude Shannon derived the fact that for a channel with additive white Gaussian noise, the maximum possible bit rate f_b is $W\log_2\left(1+\frac{P}{N_o W}\right)$, where W is the full bandwidth used for transmission, P is the average power of the transmitted signal, and N_o is the single sided noise power per Hz. This bit rate can in theory be achieved with an arbitrarily small number of bit errors.

Figure 2.9: Theoretical Bounds and Probability of Error for QPSK Modulation



If an infinite bandwidth were used, error-free communication could in theory be achieved so long as the signal-to-noise ratio were greater then -1.59 dB. If the bandwidth is limited to 1.4 times the symbol rate, then in theory, error-free communication could be achieved so long as the signal-to-noise ratio is greater than 0.73 dB. When the specific encoding and decoding schemes of QPSK modulation are used, this imposes an engineering bound, as shown by the curve.

If an infinite bandwidth were used, then the maximum possible f_b becomes $P/N_o \ln 2$. The signal power can be rewritten in terms of E_b, the energy per bit, as follows: $P=E_b f_b$. Then it becomes clear that this maximum possible f_b can only be achieved if $E_b/N_o \geq \ln 2$ (i.e. -1.59 dB).

When 40% raised cosine filters are used, an upper bound on the bandwidth is $1.4 f_{symb}$ or $0.7 f_b$, so the maximum possible f_b cannot be achieved unless $E_b/N_o \geq 1.18$ (i.e. 0.73

dB). In theory, if the signal-to-noise ratio is 0.73 dB or greater, error-free communication can be achieved.

In the actual implementation, there is a trade-off between simplicity and performance. Based on the actual encoding and decoding scheme for QPSK modulation shown in Figure 2.7, there is an engineering bound, and the probability of error can be computed. It is reduced if Gray coding is used for the mapping from bit pairs to symbol phases, because that way an error in one symbol most likely only results in an error in one bit, not two bits. If the energy of each bit is denoted as E_b, then an error occurs if the noise, Z, is greater than $\sqrt{E_b}$, and the probability of error is $Q\left(\sqrt{\frac{E_b}{N}}\right)$, where

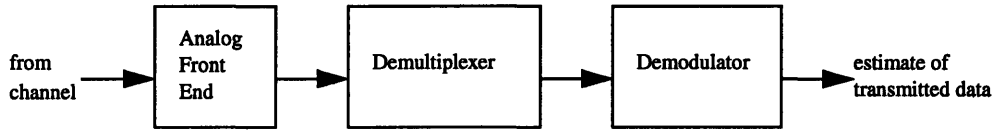
$$Q(x) \equiv \frac{1}{\sqrt{2\pi}} \int_{\tilde{x}=x}^{\infty} e^{\left(-\frac{\tilde{x}}{2}\right)} d\tilde{x}.$$

It is popular to use N_o, the single sided noise power per Hz, instead of N. Using the fact that $2N = N_o$ the probability of error can be rewritten as $Q\left(\sqrt{\frac{2E_b}{N_o}}\right)$. Although this is only a good derivation of the probability of error for a two-phase modulation scheme (i.e. BPSK), [Feher] shows that it is also valid for a QPSK system.

2.4 Receiver

The functions of the receiver are to accept the signal from the channel, down-convert it to baseband frequencies, demultiplex the carriers, and demodulate them.

Figure 2.10: Receiver

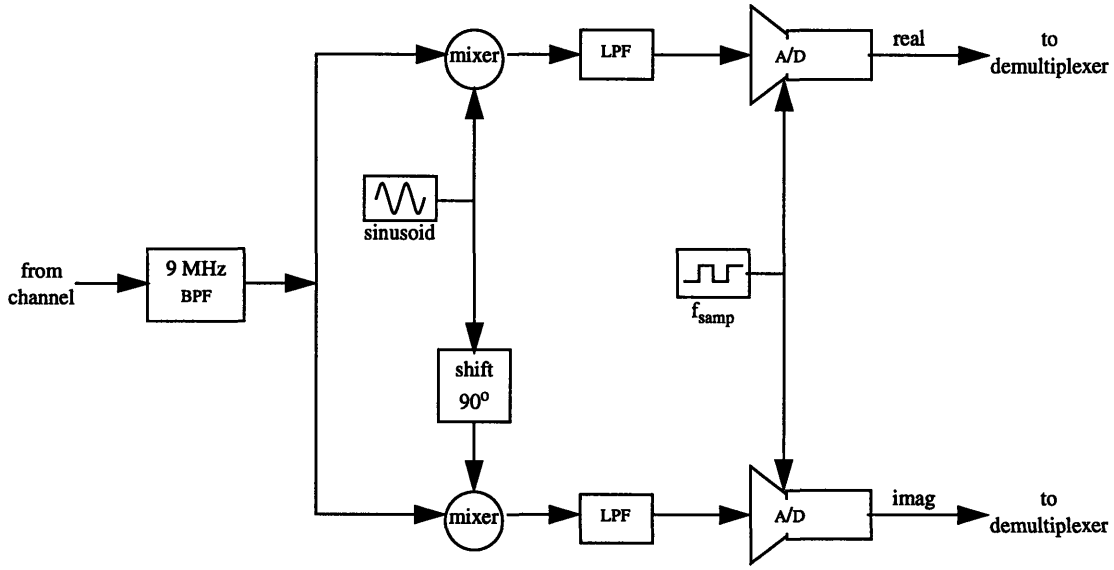


The signal from the channel enters the analog front end, which processes it and converts it to a digital signal. The carriers in the signal are separated out by the demultiplexer and then demodulated.

2.4.1 Analog Front End

The analog front end converts the incoming signal down to baseband frequencies using complex down conversion, also known as quadrature down conversion.

Figure 2.11: Analog Front End



The analog front end has circuitry for a band pass filter (BPF) and a quadrature down-conversion. Each low pass filter (LPF) is both a filter to eliminate unwanted intermodulation products from the mixer and an anti-aliasing filter preceding the A/D converter.

Frequency Manipulations of the Analog Front End

The down converter uses the incoming signal twice. It mixes the signal with a sinusoid. It also uses the signal a second time and mixes it with another sinusoid that is a quarter wave out of phase with the first. It produces two down-converted outputs: one called the real output and one called the imaginary output. Effectively, the input signal is multiplied by a complex exponential:

$$output = input \cdot e^{j\omega t}$$

which can also be written as:

$$output = input \cdot (\cos(wt) + j\sin(wt))$$

Even though the physical mixers only use real sinusoidal waves, the input signal has, in effect, been multiplied by a complex exponential. The ramifications are that the spectrum of the input signal is convolved with only a single impulse in frequency and is simply shifted in frequency. All purely real time signals have a spectrum that is conjugate symmetric. However, the pair of outputs from the complex down converter together form a complex signal whose spectrum is not conjugate symmetric. Complex downconverters can use half the sampling frequency of real downconverters for the same signal. Another way to look at it is that in complex downconversion, there are two outputs, and in real downconversion, there is only one output. Thus the use of a lower sampling frequency makes sense and is justified. The other advantage of quadrature downconversion is that the local oscillators do smallernot have to be synchronized with the carrier.

These two signals enter analog-to-digital (A/D) converters and after that, they are handled by digital hardware, which implements complex arithmetic. A major question to be answered by simulations was whether 8 bits of precision in the A/D were enough.

Two Options for the Anti-Aliasing Low Pass Filters

The low pass filters perform two functions. First, they eliminate undesired intermodulation products that appear at the output of the mixer. Second, they perform anti-alias filtering, so that the signal entering the A/D converters is sufficiently bandlimited (i.e., satisfies the Nyquist sampling criteria.) For this particular demultiplexer, the filter must be designed with a passband edge of 4.5 MHz and a stopband edge of 5.76 MHz. A rule of thumb is that a filter stopband attenuation should be 40 dB or more.

There are two alternative ways to perform the low pass filtering in order to achieve these two function: an all analog approach or a combined analog and digital approach. If the filters are all analog, then elliptic filters are required to achieve the sharp cut-off edges. However, a complication arises because these filters have a group delay that is not nearly flat. This can be compensated for by all-pass filters that serve as group-delay equalizers. The output of the group-delay equalizers is then passed to the A/D converters.

With current technology, the elliptic amplitude filters and group-delay equalizing filters can be designed using software programs such as Superstar, which takes into account the finite Q for inductors and generates ideal values for the resistors, capacitors, and inductors. On an actual board with analog circuitry, these configurations can be made to work, but it can require difficult debugging.

The second way to perform the filtering is to use an analog filter, then the A/D converter at a higher sample rate, another digital filter, and a downsampler. In this case, the requirements for the analog filter are lax. It can be designed so as to have relatively constant group delay across the frequencies of interest, so no equalizer is needed. The sampling frequency of the A/D converters is pushed to $2F_{samp}$ so there is no aliasing even though the first analog filter is not very sharp. Next, the digital signal is processed with a digital filter which has a finite impulse response (FIR). It is designed to be symmetric, so the group delay is perfectly constant. It has enough taps in its impulse response so that it has a sharp frequency response. Simulations with software called the Signal Processing Workstation (SPW) have shown that a filter with 55 taps would be more than sufficient. After the filtering, a down-sampling operation is performed in which half of the digital samples are discarded. The signal thus effectively has the sampling rate F_{samp}.

Samples Per QPSK Symbol

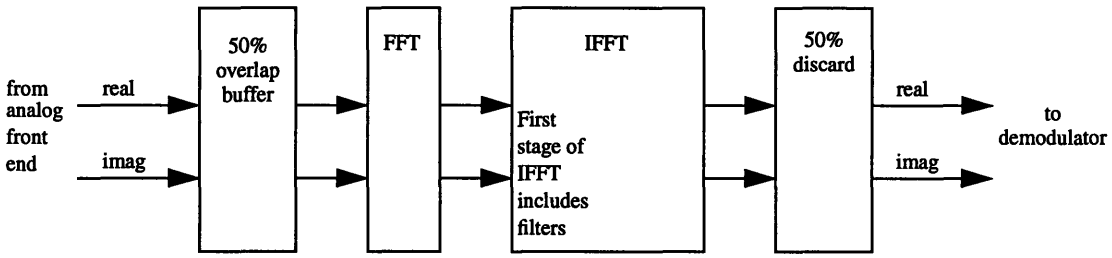
After the A/D converter, each carrier is represented by digital samples. A ratio that comes up frequently is the number of samples per symbol. If the transmitted symbol rate is F_{symbol} and the A/D converter sampling frequency is F_{samp} then the number of samples per symbol at the input to the demultiplexer is F_{samp}/F_{symbol}. The MCD-1 demodulator ASIC can demodulate signals that are between 2 and 4 samples per symbol. The demultiplexer must make sure its output is within that range for each carrier.

2.4.2 Demultiplexer

The demultiplexer separates the carriers that arrive together and are frequency-multiplexed. It applies up to 24 square root raised cosine filters to the signal and produces time-multiplexed signals at its output. It operates not by directly convolving the input signal with the impulse responses of the filters but instead by a much more efficient method called fast convolution. It takes a fast Fourier transform (FFT) of the input signal, applies the filters via multiplication in the frequency domain. Then it performs many inverse FFTs (IFFTs) to convert each signal back to time coordinates.

In this implementation, the filters come at the first stage of the IFFT. This is because the BDSP 9124 chips used for the FFT and IFFT operations have a function for multiplying the incoming data by a set of coefficients.

Figure 2.12: Demultiplexer Overview



There is a FFT, a filter, and an IFFT. The first stage of the IFFT is where the filters are implemented. At each stage, the real and imaginary parts of the data are represented by two's complement binary numbers.

Chapter 3

Survey of Architectures

In this chapter, the FFT architecture is compared to alternative architectures with the same performance to contrast the hardware complexity.

3.1 Our Approach: Fast Convolution with FFT

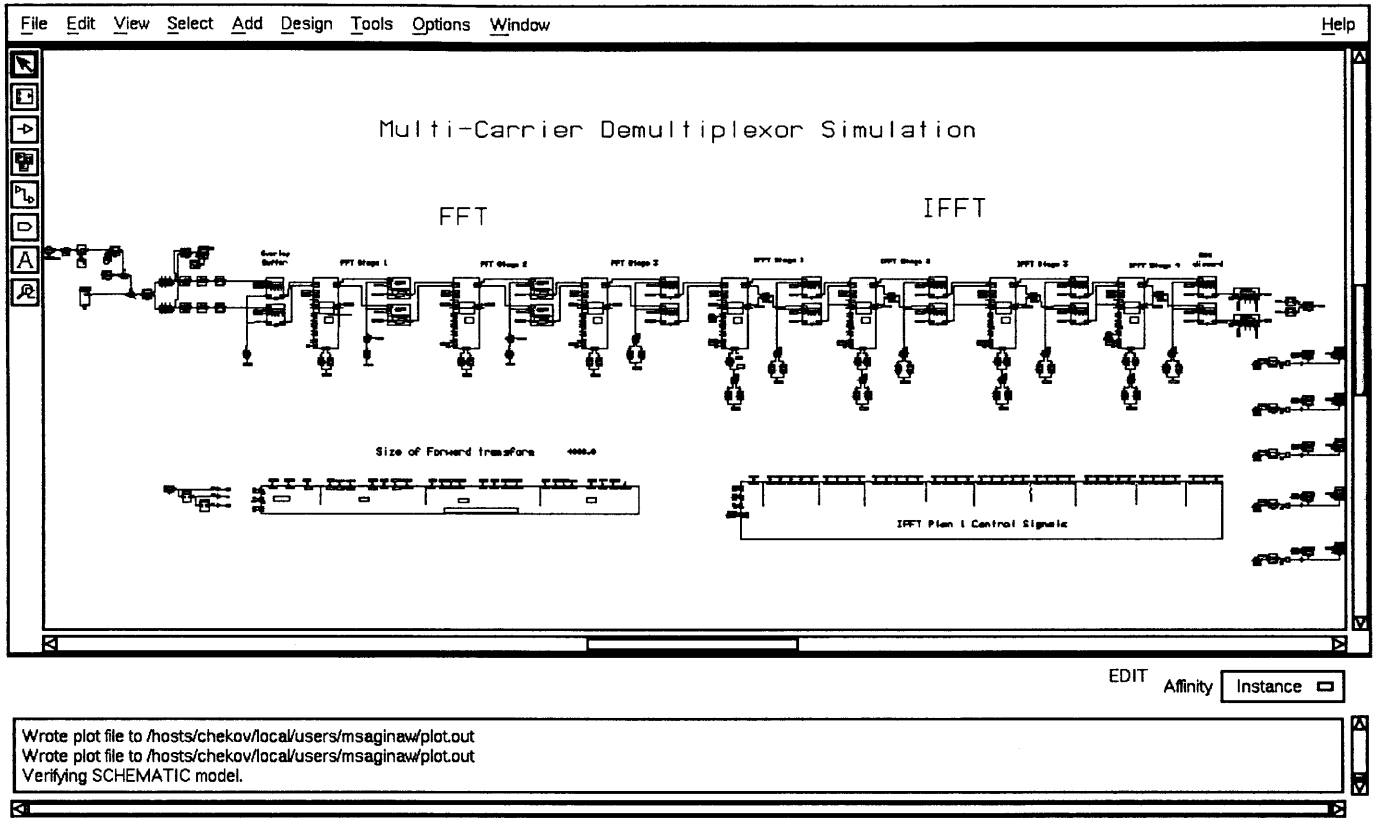
The fast convolution approach with a shared FFT and individual IFFTs permits an architecture with only one analog front end. It provides a very convenient way to shift all the signals to baseband by doing so directly in the frequency domain. Furthermore, it is well suited to handle the mix of unrelated carrier sizes that characterize the INTELSAT IBS and IDR carriers.

Figure 3.1 shows the overall block diagram from the simulation of the demultiplexer. It shows the 50% overlap buffer, the FFT, the IFFT (the filtering is done in the first stage of the IFFT), and the 50% discard buffer.

3.2 Alternate Approaches to Downconversion

If the FFT is not used, some method must be found to bring each carrier to baseband in preparation for the filtering.

Figure 3.1: Demultiplexer Architecture: Simulation Block Diagram



This block diagram from the simulation shows the architecture of the demultiplexer. At the left, digital data enters a 50% overlap buffer. It is processed through the FFT stage, which consists of 3 BDSP 9124 chips and RAMs. The frequency samples are filtered in the first stage of the IFFT and passed through the rest of the IFFT to the 50% discard buffer. At the far right of the simulation, 5 signal sink blocks act like logic analyzers, capturing the data for graphical display. The two blocks at the bottom generate timing and control signals for the hardware in the FFT and in the IFFT, respectively. They are attached to the BDSP 9124 and RAM blocks by connectors. In this high level block diagram, the words for the connectors cannot be seen and are merely shown as rectangles.

3.2.1 One Analog Front End per Carrier

The demultiplexer could be done 24 separate times, with no resources shared. There could be 24 separate analog front end boards, with programmable local oscillators for quadrature downconversion. The A/D converter sampling frequencies could be set for each carrier so that the number of samples per symbol is between 2 and 4, as required by the demodulator. The output of the A/D converters could be passed to FIR or IIR filters. The requirement of 24 analog front end boards immediately makes this an undesirable alternative.

3.2.2 Combination of Analog and Digital Tuners

One analog downconverter could be used, and the carriers could undergo another downconversion by digital methods so that they could each be brought to baseband. After the analog front end, the digital signals would be used 24 times, so fan-out problems would have to be avoided, for instance by using a bank of registers.

To continue the downconversion, each replica from the analog front end output could be multiplied by a digital complex exponential signal at the proper frequency to convert it to baseband. This would require the generation of 24 different complex exponentials and the use of 24 different digital multiplication units. They could operate in parallel and be followed by FIR or IIR filters. To achieve on-the-fly switching, 48 signal generators for complex exponentials and 48 multipliers would be needed. A design with a bank of registers, 48 signal generators, 48 multipliers, and 24 filters (48 parts at minimum, as discussed below), could not be fit on one board.

3.3 Alternate Approaches to Filtering

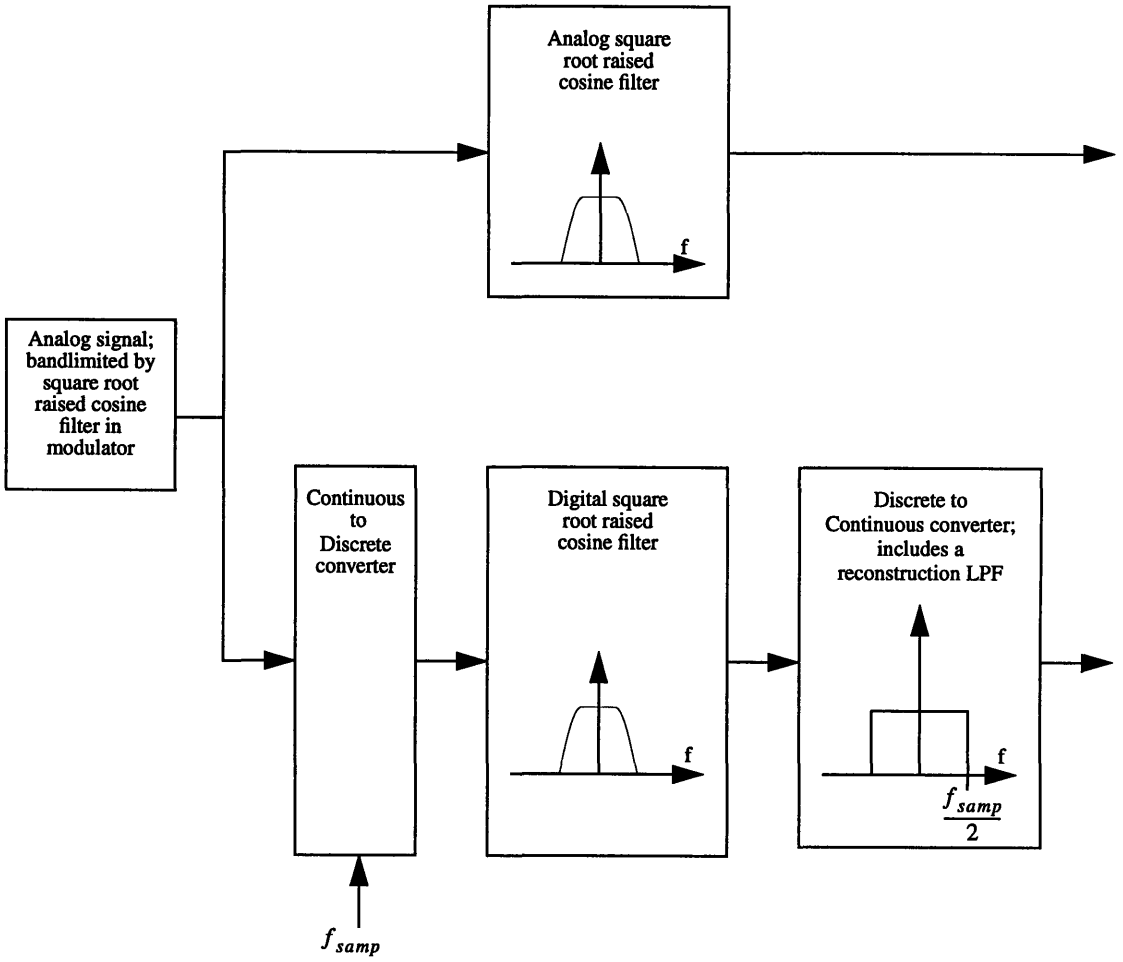
In this section, the assumption is that some method from section 3.2 has been used to get the desired signal to baseband at a number of samples per symbol that is between 2 and 4, to meet the requirement of the demodulator. FIR and IIR filters are explored as alternative ways of implementing the data shaping filter. The comparison yields an interesting gauge on the capability of the demultiplexer's filters in terms of equivalent FIR and IIR filters.

3.3.1 FIR Filters

The infinite impulse response for the raised cosine filter can be truncated to form an IIR filter. As a way of considering the imperfections that are introduced in doing so, two equivalent systems are compared, as illustrated in Figure 3.2.

The impulse response of the FIR filter is typically generated by computing the ideal impulse response and truncating it (i.e., using a rectangular window in time). This leads to nonzero ISI, as can be seen by considering the reconstruction filter in figure Figure 3.2. The portion of the FIR filter that was knocked out by truncation would otherwise have made the filter have 0 ISI. The combination of that missing portion with the sinc function that is the impulse response of the reconstruction filter makes for nonzero ISI. Fortunately, whereas the brick wall filter's impulse response (a sinc function) falls off as $1/n$, which would lead to an infinite amount of ISI, the impulse response of the square root raised cosine filter falls off faster, so the ISI is bounded.

Figure 3.2: Two Equivalent Systems for the Data Shaping Filter



Ideally, the continuous time filter (shown above) and the discrete time filter system (shown below) are equivalent. The digital filter can be approximated by truncating the ideal impulse response and using an FIR filter. However, by truncating the impulse response, the digital system has ISI at the output of the reconstruction filter. This model is used as a way of assessing the performance of the FIR filter.

A rule of thumb is that if the FIR filter has taps that extend over 6 symbols, then the degradation is acceptably small. If the input were at 2 samples per symbol, then 12 taps would be acceptable. If the input were at 4 samples per symbol, then 24 taps would be acceptable. This size impulse response can be implemented by one chip, so the filters could be implemented in 24 different chips.

3.3.2 IIR Filters

Pole-zero approximations for analog implementations of the square root raised cosine filters were studied in depth in [Poklemba]. The ideal shape is unachievable because it is perfectly flat in the passband and totally attenuated in the stopband. Therefore, Poklemba used the following criteria for acceptable deviation: a peak error ripple of 0.5 percent in the passband and 40 dB of attenuation in the stopband. Each corresponds to a degradation of about 0.005 dB in the bit error ratio curve. Poklemba found that the 40% square root raised cosine filter can be acceptably approximated by a system with 4 zeros (2 pairs of complex conjugates that are all purely imaginary) and 8 poles (4 pairs of complex conjugates).

Since IIR filters in general do not have constant group delay, an equalizer is needed, but this also cannot be implemented exactly. Using the same criteria for acceptable error, Poklemba found that the group-delay equalizer could be approximated by an all-pass system with 6 roots, corresponding to an analog filter with 6 zeros and 6 poles.

The impulse invariance technique can be used to create an IIR digital filter with the same magnitude response as the analog filter. The IIR filter would also have 8 poles, which would be mapped directly from the s-plane to the z-plane. However, the impulse invariance technique does not guarantee anything about the mapping of zeros from the s-plane to the z-plane. Therefore, the group delay of the magnitude filter and the equalizer would not be predictable based on the analog design. The new group delay of the magnitude filter in IIR form would have to be evaluated, and an appropriate equalizer designed.

The following discussion of the structure for implementing the magnitude portion of the IIR filter is drawn from [OS], with modifications for the specific filter in this case. The

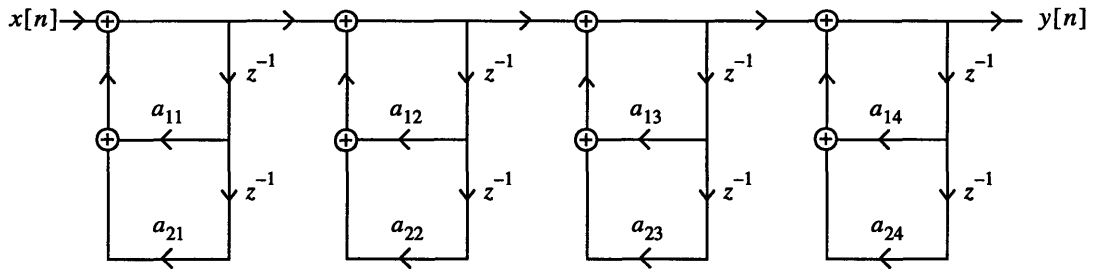
poles are complex conjugate pairs, so a cascade form implementation is appropriate. The cascade form can be designed to have much smaller error due to coefficient quantization than direct form IIR filters of the same order.

Suppose one pole pair consists of the complex conjugates d and d^*. To get a cascade form implementation, first pair the poles as follows: $1/[(1-dz^{-1})(1-d^*z^{-1})]$. This can be rewritten as follows: $1/(1-2Re(d)z^{-1}+|d|^2z^{-2})$. Note that the coefficients are all real. Then new variables can be introduced: $a_1 \equiv 2Re(d)$ and $a_2 \equiv -|d|^2$. Then the pair of poles can be written as follows: $1/(1-a_1z^{-1}-a_2z^{-2})$. Finally, since there are 4 complex conjugate pole pairs total, an extra index can be introduced and the transfer function for the magnitude response of the filter can be written as follows:

$$H(z) = \prod_{i=1}^{4} \frac{1}{1-a_{1i}z^{-1}-a_{2i}z^{-2}}$$

The signal flow diagram to implement the transfer function is shown in Figure 3.3.

Figure 3.3: IIR Filter Implemented as a Cascade of Second Order Sections



The cascade structure can be used to implement the 4 complex conjugate pole pairs for the magnitude portion of the IIR filter.

The number of ways to arrange the 4 stages is 4 factorial. In order to minimize the sensitivity of the filter to noise at the input and to data overflows, a rule of thumb is to arrange the second order sections so that the poles are arranged in either increasing or decreasing magnitude, but not in some other order.

For implementation, 8 multipliers, 8 adders, and 8 delay registers are needed for each of the 24 carriers. These figures need to be doubled to 16 for a system capable of on-the-fly switching. If they are implemented as separate chips, this is 576 parts for the magnitude portion of the filter alone.

3.3.3 Polyphase Filters

Polyphase filters are limited to carriers of the same size. It is possible to cascade polyphase filters in different ways to handle a few related carrier sizes, but this would not suit the diverse carrier sizes in the INTELSAT specifications [Thomas].

Chapter 4

Signal Processing Background

The overlap save technique of fast convolution was used for the demultiplexer. In this chapter, the equations for the family of Fourier transforms are listed and the specifications for the INTELSAT carriers are connected to techniques for using discrete time signal processing. Finally, the location of the good samples in the overlap save operation is discussed.

4.1 Transforms

Two widely used coordinate systems for representing signals are time coordinates and frequency coordinates. There exist mathematical transformations to convert a signal from one set of coordinates into the other, as shown in Figure 4.1.

In the demultiplexer that is the subject of this thesis, the signals are all discrete. The signals come out of the A/D converter and are discrete in time. The electronic hardware that transforms the signals into frequency coordinates is limited to taking a finite number of samples in frequency; the frequency samples in the demultiplexer are discrete in frequency also. The type of transformation that applies to these kinds of signals is the Discrete Fourier Series, and the related Discrete Fourier Transform (DFT) for signals that have finite support, and are zero outside of a specified range of index values.

4.2 INTELSAT Carrier Specifications and DSP

Table 4.1 shows parameters in the spreadsheet for each carrier that are relevant to the DSP details discussed in this chapter.

Figure 4.1: Fourier Transforms for Continuous and Discrete Signals

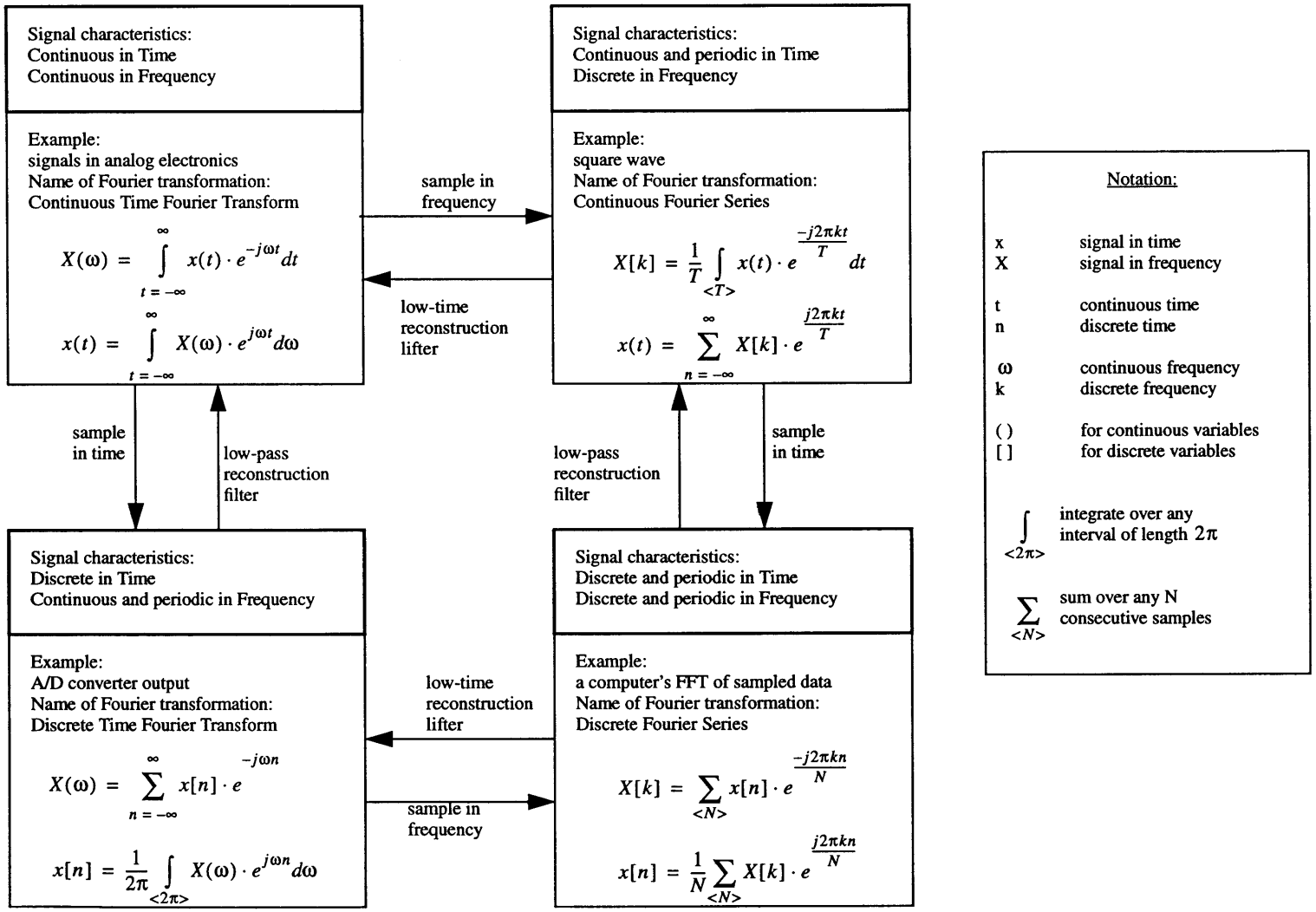


Table 4.1: INTELSAT Carrier Specifications and Manipulations for DSP

Info Rate (kbits/s)	Trans. Symbol Rate (ksymb/s)	Samples per Symbol after A-to-D	Symbols per block of 4096 samples	IFFT Size	Samples per Symbol after FFT	Number of carriers of this type
384	307.2	37.50	109.2	256	2.34	23
768	546.1	21.09	194.2	512	2.64	11
1544	1,093.3	10.54	388.7	1024	2.63	5
2048	2,144.0	5.37	762.3	2048	2.69	3

Transmitted Symbol Rate

This column is copied over from Table 2.1.

Samples per Symbol after A-to-D

This column is the A/D converter sampling rate (11.52 MHz) divided by the transmitted symbol rate.

Symbols per Block of 4096 Samples

This column is the number of FFT samples (4096) divided by the number of samples per symbol after the A/D.

IFFT Size

This column is a power of 2, and it is adjusted so that the next column is between 2 and 4.

Samples per Symbol After FFT

This column is the number of IFFT samples divided by the number of symbols per block of 4096 samples. The entries in the column must be between 2 and 4, as required by the demodulator.

Number of Carriers of This Type

In general, the demultiplexer will work with different types of carriers. However, if the carriers were all the same type then this column lists the number of carriers that could be accommodated. There are three constraints at work:

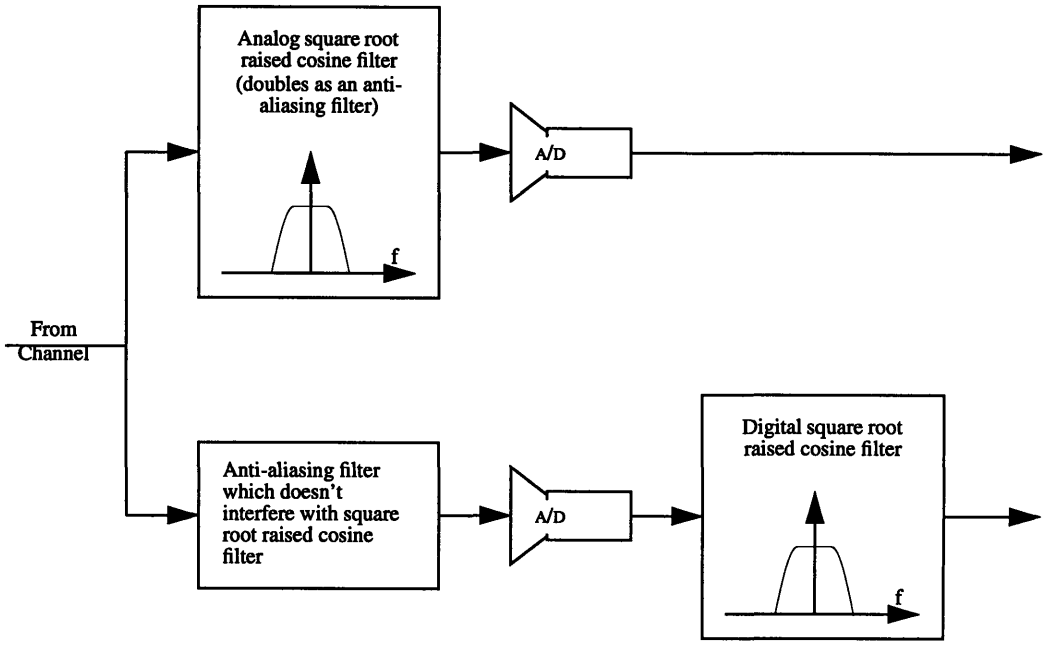
1. Demodulator handles 24 carriers: The demultiplexer output is the demodulator input. COMSAT's MCD-1 is a shared demodulator ASIC with 24 demodulators. The maximum number of carriers that can be handled is therefore 24.
2. 9 MHz bandwidth: The demultiplexer is designed to accommodate a 9 MHz bandwidth. The total allocated bandwidths of the carriers must be no greater than 9 MHz.
3. 8192 IFFT points: The IFFT operation has 8192 clock cycles. Within those cycles, there must be room for the samples in the IFFT.

4.3 Filter Design Technique

4.3.1 Filter Shape

The filter design is based on the equivalent systems shown in Figure 4.2. The digital data shaping filter is not actually identical to the analog one. Using the fast convolution and FFT approach for the demultiplexer, the filter is implemented by multiplying FFT samples (frequency coordinates) by filter coefficients (also frequency coordinates). In other words, the filter is sampled in frequency. This results in aliasing in the time domain since the ideal impulse response is not time-limited. It is exactly analogous to sampling a signal in time and producing frequency aliases if the signal was not band-limited. That time-aliasing means that the filter cannot have strictly 0 ISI, but the contribution from all the aliasing is acceptably small.

Figure 4.2: Equivalent Systems for Filtering in the Receiver



The filter design is based on the equivalence of these two systems. On top, the filtering is done by analog hardware. Underneath, it is done by digital hardware, and the filter ideally has the same shape. However, in the fast convolution and FFT approach, the filter in the digital system consists of samples of the desired frequency response. This corresponds to aliasing in time, which corresponds to nonzero ISI.

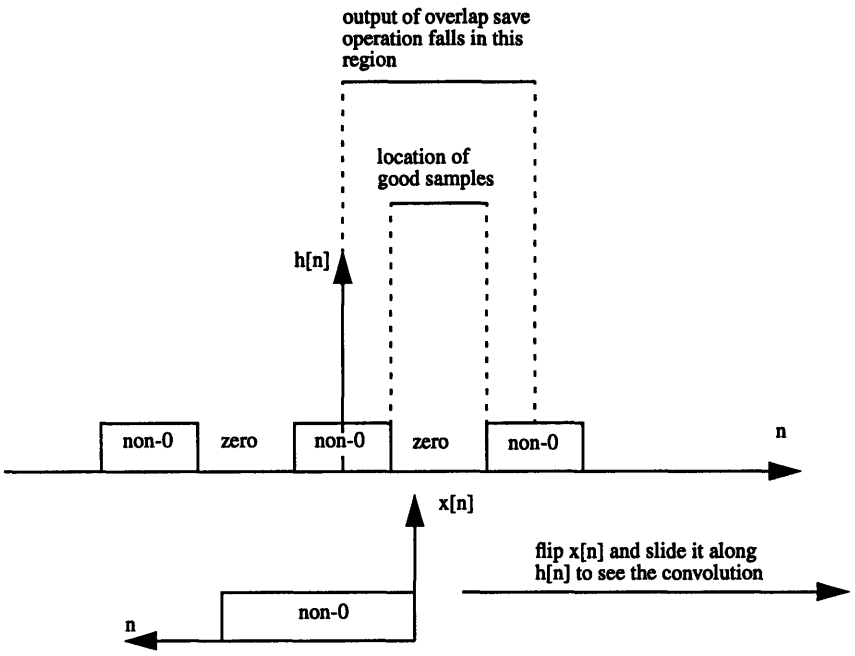
4.3.2 Location of Good Samples

In the overlap save method, some of the samples must be discarded. The location of the good samples can be seen by considering the impulse response of the filter. Even though that impulse response is not explicitly calculated or worked with, it comes into play in considering the location of the good samples.

Since the frequency filter is purely real, the impulse response is conjugate symmetric. For the overlap save operation to work with a 50% overlap, the assumption is that the impulse response is all 0 or sufficiently small for 50% of its samples. Since the filter is

sampled in frequency, the impulse response is periodic in time. The convolution and the resulting location of good samples is shown in Figure 4.3.

Figure 4.3: Location of Good Samples



Non-aliased samples are produced when the flipped signal, x[-n], only overlaps one segment of non-zero h[n] samples. The impulse response, h[n], is known to be conjugate symmetric, so it must have non-zero values clustered near the n-axis and zeros outside of that region. The impulse response is periodic because the frequency response is sampled in frequency.

After the circular convolution operation, the non-aliased samples are left in the middle half of the output. If it were desired to change this location, the impulse response could be shifted (convolved with an impulse). Since the filter is being implemented via the frequency coefficients, they would have to be multiplied by a complex exponential in order to change the location of the good samples.

Chapter 5

SPW and Simulation Software

5.1 SPW

The demultiplexer simulation was written and run using the software program called Signal Processing Worksystem (SPW). In SPW, one draws a block diagram of a system and then runs the simulation. Blocks are connected to each other through wires, connectors, or ports. Wires are direct connections. Ports are connections that appear at one spot and then again at another spot, and are identified by their name. Ports are for connecting the external inputs and outputs of a block to the internal workings of a block.

Some SPW blocks are built in and fundamental. Others are based on the fundamental SPW blocks. The user can define new SPW blocks. These can be based on other SPW blocks or on external C code. In this simulation, most of the blocks were actually calling external C code, so that the exact operations the blocks performed could be specified. The memories were based on simple external code. The BDSP 9124 DSP chips were simulated by external code as well. In that case, the code came from the BDSP company, whose software engineers wrote the code specifically to simulate the BDSP 9124 hardware chip. By using this exact simulation chip, every bit of the input and output of the chip could be simulated. Issues such as quantization errors are thus taken into account by the simulation. Also, the simulation can serve as a generator for test vectors and can be used to help debug the hardware. This reduces the risk of the project, which was one of the goals of the customer.

After the user enters the block diagram and gives SPW the command to run, SPW examines the connections in the block diagram and generates a program to generate data at the beginning of the simulation, process it at each stage of the simulation, and store output where the user has placed a device such as a signal sink, which is analogous to an oscilloscope. If a block diagram has already been examined, and a program already written and compiled, and only certain changes have been made to the block diagram, then the block diagram does not need to be re-examined and re-compiled. The pre-existing one can be used instead -- and this savings of time can be very convenient for complex simulations.

The clock timing in the simulation is somewhat different from the clock timing in hardware. In hardware, the timing is based on the rising edge of signals. In the simulation, this level of detail is not simulated. Instead, at each iteration of the simulation, trigger signals are examined, usually to see if they are 0 or 1. If they are 0, this is considered not a clock. If they are 1, this is considered a clock active signal. The main clock in this simulation is a signal that is 1 at every iteration. The slower clocks are signals that are 1 every second cycle, and every fourth cycle, respectively.

In the hardware, there is propagation and contamination delay. In the simulation, these are not present. A block functions by looking at its inputs, running code, and producing outputs. These all happen in one iteration. Any dependencies of the input of one block on the output of another are arranged by the SPW simulation scheduler so that the outputs of the previous block are computed and updated before the next block is considered.

5.1.1 Professional Software Package

The SPW software program by Cadence has thorough documentation. The manuals contain examples and they have indexes that are cross-referenced. In addition, the

technical support from Cadence is very helpful. For instance, through technical support it was learned that in order to make simulations that call external C code that has more than just standard C functions, the standard SPW simulator could not be used. Instead, the other kind of simulator, the Code Generation System (CGS) had to be used. In CGS, SPW takes longer to compile the simulation because it generates actual C code from the block diagram, and then compiles that code. However, it has advantages over the standard simulator because that compiled simulation then runs faster than the standard simulator. In another instance, the technical support from SPW was helpful when they made it clear that SPW's built-in bidirectional ports cannot be used in custom-coded blocks.

5.1.2 Powerful Signal Display Tool

SPW has a SigCalc display tool which can display complex signals by showing their real and imaginary parts or by showing their magnitudes and phases. It allows users to zoom in or out in scale. It allows the user to easily copy and paste signals, and to combine two signals by operations such as addition, multiplication, and convolution. It has buttons for performing filter operations on signals, for taking transforms of signals, and for making eye diagrams.

5.1.3 Simulation Scheduling Performed by SPW

Another extremely useful feature of SPW is that it performs simulation shceduling. The user only needs to draw blocks, connect them, and sometimes, specify a fundamental node in the block diagram. However, the user does *not* need to write functions that call other functions with arguments, worry much about the type of variable being used, or worry about what functions are calling what other functions. All this is performed in a systematic way by SPW.

5.1.4 Useful Blocks Built In

Many of SPW's built-in blocks were very helpful, and they are described here.

SPW has many blocks to be the source of data. For instance, there are blocks that generate random patterns of 1's and 0's. These were used to generate random bit patterns. Another SPW block that is based on these random bit blocks is the random QPSK source block. It provides an output which is a complex number that is a multiple of (1,1), (1,-1), (-1,-1), or (-1,1). One parameter this block takes is the signal power. This can be used to ensure that the signals of various bit rates have the same energy per bit.

The raised cosine filter block was very helpful, and it was used extensively to make simulated baseband data. It can optionally include a 1/sinc response, which is useful for simulating the output of a modulator. It has the option of making its filter a full or square root raised cosine shape. For simulating only the modulator, the square root shape was used. It accepts as a parameter the sampling frequency of the input and also the percent roll-off of the filter. Thus it was easy to specify that the filters should be 40% raised cosine filters.

The A/D converter block was used to quantize the floating point data as a way of simulating the effect of A/D converters. The number of bits of quantization can be set as a parameter. The A/D converter block handles overflow properly by clipping the data if it overshoots the maximum allowable value. The block also allows the user to set the maximum amplitude and it sets its thresholds accordingly.

The complex spectral shift block was used many times, in simulating the input data. It takes an input signal and multiplies it by a complex exponential digital signal, which is the equivalent of shifting the signal in frequency. It takes as parameters the sampling frequency of the incoming signal and the desired shift frequency. It was used in order to

perform the frequency multiplexing, to move the signals to different frequencies before they were added together.

Finally, there were two kinds of file input/output blocks that were used extensively. One was the various signal source blocks and the other was the signal sink blocks.

The signal source blocks are used to read data from a file and provide them as input data for an SPW simulation. There are signal source blocks for files with real or complex data. There are also signal source blocks for reading a file with a real or complex constant. Unfortunately, these only read the first line of a file and held it for one iteration of the simulations. Thus when it was desired to have a constant read from a file for the entire simulation, it was necessary to put that signal - read from a constant - into a register and then to hold that value for the entire simulation. This issue arose because if a file with a constant is changed, then the simulation does not need to be recompiled, but if a built in block that does hold a constant for the duration of the simulation was used, and it was necessary to change the value of that constant, then the simulation had to be recompiled. For the entire simulation, this took 14 minutes.

These source files can contain data in binary or ASCII form. The ASCII form takes about eight times as much disk space, but the file contents can be read with a conventional text editor.

Similar to the signal source blocks were the signal sink blocks. These blocks are analogous to oscilloscopes. They store incoming data into files, which can be viewed with the SigCalc signal viewer. They can store real or complex signals in ASCII or binary format. In addition to storing signals for display, the signal sink blocks were useful in another role as well. Many of the signals used as sources were generated in an SPW simulation and written to files with the signal sink. Then they were read with the signal source blocks for a different simulation.

5.1.5 Blocks Can be Built Based on Other SPW Blocks

One feature of SPW is the ability to put together some SPW blocks and make another, higher level SPW block. The higher level block can then be used in another block diagram. If the underlying block diagram is changed, that is automatically reflected in the higher block. This can proceed for many levels, so that a complex system can be built up from simpler underlying layers.

In this thesis, one block built up from other SPW blocks is the divide-by-n clock. This block takes in a clock signal and a parameter for the division. The output is another clock signal. It is arranged so that it meets two constraints. First, it is a slower clock signal than its input, by a factor of n, the parameter. Second, it is synchronized with its input. To relate this to the way these blocks are used in the simulation, the fastest clock is one all the time; the next fastest block is one every other cycle; the slowest clock is one every fourth cycle. The two slowest clocks are synchronized, meaning that when the slowest clock is 1, the middle clock is one also. This simulates the clocks that are used in the hardware, where the slower clocks are phase locked to the fastest clock.

5.1.6 Using External C Code

The capabilities offered by SPW's built in blocks, and the other blocks which can be constructed from them, was not enough for this project. In fact, it was necessary to write custom coded blocks for most of the blocks used in the simulation. These include the RAMs, PROMs, and the blocks that simulated the BDSP 9124 DSP chip.

SPW provides two ways of using blocks whose behavior is specified by external code. In the first way, one can continue to use the standard SPW simulator. However, this way is limited to using C code in the standard C libraries.

In order to use blocks which are specified by external code which is not part of the standard C library, one can no longer use the standard SPW simulator. Instead, it is necessary to use another SPW simulator called the Code Generation System (CGS). When SPW uses CGS, it examines the block diagram and actually writes C code for the simulator. It then compiles that code and runs it. The simulations take longer to prepare, but they run faster.

5.2 BDSP 9124 C Source Code

5.2.1 Software Code

Unfortunately the simulator for the BDSP 9124 did not work on the workstations running the Unix operating system. It could not multiply two numbers, unless one of them was 1 or -1. However, this correct operation was traced to lines in the source code that handle those special cases. Other multiplications were giving completely unsatisfactory results. The test code, which tested the simulator against the actual hardware, was examined and found to contain a very limited number of function codes. In particular, it did not contain the function codes that involve multiplications.

The software was tried on a personal computer running MS-DOS, and it worked. Ultimately, the problem was chased down to an endian error, meaning, a difference in the order in which bytes are stored on the different computer systems. For long integers with 4 bytes of storage, the MS-DOS and the Unix operating system place the bytes in opposite order. The C source code routine for multiplication was studied and a type of variable called a union was found. That union held 4 bytes. Sometimes, they were treated as a long integer; sometimes they were treated as two short integers. The order in which the two short integers was accessed was different in MS-DOS (in which the code was written)

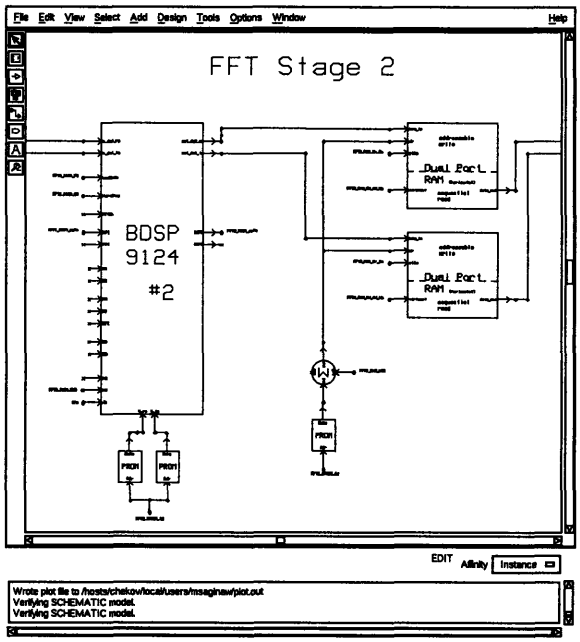
and the Unix operating system, which is run by the workstations -- and those workstations have and run the SPW software.

The original code was backed up, and eight lines of code were changed to alter the workings of the union variable. The software was run on the PC and the workstation with the same inputs and the function codes being used for the demultiplexer simulation (as well as one other function code: the CMUL (complex multiplication) function code). The outputs were identical to the last bit, and the conclusion was that the software had been successfully modified for the workstation.

5.2.2 Using the C source Code with SPW

The BDSP 9124 C source code was successfully integrated with the SPW simulation. A figure was made to symbolically represent the inputs to and outputs from the BDSP 9124. That figure is associated with a parameter file which indicates how many of the BDSP 9124's 24 bits should be passed on to the output. The other file the figure is linked to is a file with the extension .expr, and the code in that file is the first layer in the code for simulating the BDSP 9124 chip.

Figure 5.1: The BDSP 9124 Block Symbol



This figure shows the symbol made to represent the BDSP 9124 DSP chip, as it was used to simulate the second stage of the FFT. Data inputs come into the chip at the left and coefficients come in at the bottom. Other signals enter at the left, such as the enable signals, the function code, and the StartStop signal. Other signals are also at the output on the right, such as the data scale factor output and the block scale factor output.

Real and imaginary data signals enter the chip at the left and coefficients enter at the bottom. Other signals also enter at the left, including the enable signals, the function code, and the StartStop signal. Other signals at the output include the data scale factor output and the block floating point output. The two issues of timing for serial and parallel data streams and structures in the C language came up in interfacing the SPW simulation to the BDSP 9124 C simulator. To make the interface work, the simulation works with three layers of code.

Timing Issues

There are three layers of code to write about: the layer that interfaces with SPW, the layer that calls, parallelizes, and serializes data, and the layer that calls the bdsp 9124 C source code functions. The code that calls the BDSP 9124 source code from SPW is in the form

of an interface, for two reasons. There are two issues here. One is that the SPW simulation works with serial data whereas the BDSP 9124 C source code works with structures, and works with data sets in chunks of 4 or 16 pieces of data at a time. The other is that the interface which works with structures itself needs an interface, since SPW blocks apparently cannot explicitly work with structures.

The first issue is the issue of serial verses parallel data. The C source code that simulates the BDSP 9124 chip works with the data in datasets of 4 or 16 points at a time, depending on the function code. In order to call it from SPW, the C source code for the BDSP 9124 block must funnel its input data into the structures, and call the BDSP 9124 C source code functions with inclusions of pointers to the input and a pointer to the output. This interface code must then funnel the data from the output structure into the serial format of the SPW simulator.

SPW and Structures in C

The other issue is an issue about structures. It is not a general issue, but rather, it is peculiar to SPW. Even though SPW allows users to use the CGS system and call their own custom coded C blocks, it has a restriction. Unfortunately, there is no ability to define structures and use them in the code that interfaces directly with SPW. All efforts to do so failed.

It was necessary to get around that by defining a three tier system of code. First, there is the code that interfaces directly with SPW. It takes in input and gives out output. That first layer sends values to the second layer. The second layer has a clock and keeps track of incoming and outgoing data. It puts incoming data into a structure and selects data from the output structure and serializes it. In other words, this second layer parallelizes the input and serializes the output. It uses structures, but it is shielded from SPW by the

first layer interface. This second layer calls the third layer, which is the BDSP 9124 C source code. It takes in the structure with input data, and the control signals such as the function code. It computes the exact output of the BDSP 9124 chip and puts those quantized numbers in the output structure. That output structure is received by the second layer and serialized.

First Layer of Code

The first layer of code calls the second layer of code every iteration of the simulation when the clock to the BDSP 9124 chip is active. For the BDSP chips in the IFFT, this occurs every iteration of the simulation but for the BDSP chips in the FFT, this occurs only every other iteration.

Another shortcoming in the BDSP 9124 C source code simulator is that it does not have any provision for accepting and acting on the enable A or the enable B signals. It simply reads all of its control signals every time it is called. Therefore, the functionality of the enable A and the enable B signals was added to this first layer of code. The functionality was simulated based on the description of the functionality in [9124 UG].

Second Layer of Code

The second layer of code calls the third layer of code every 16 or 4 times it is called, depending on whether the function code is BFLY16 or something else. The BFLY16 function code is the only one for which the BDSP 9124 chip simulator calls works with data in datasets of 16 at a time. For all other function codes, the chip simulator works with datasets of 4 at a time.

Third Layer of Code

The third layer of code is the BDSP 9124 code itself. This has only one function that the user interfaces with: the ProcessData function. The second layer of code prepares the structures and, based on its clocking, decides whether to call the ProcessData function to run the simulator.

This three layered system would not have been necessary if the code that interfaces with SPW could handle structures. In that case, a two layered system would have sufficed: one layer to interface with SPW, to the parallelization to the input and the serialization to the output, and handle structures in order to call the BDSP 9124 C source code functions.

5.3 Other Simulation Efforts

Before the decision to carry out the simulation in SPW with an interface to the BDSP 9124 C source code was reached, other ideas were considered.

5.3.1 Ptolemy

This well developed software is a work in progress. It is a project by communications professors and graduate students at the University of California at Berkeley. It is similar to SPW in many ways. It provides a way of making a graphical simulation with symbolic block diagrams. Data is exchanged between blocks by connecting them with wires. There are many built-in blocks and the facility to write custom coded blocks as well. In fact, the libraries of Ptolemy are more extensive than those of SPW. In addition, Ptolemy has more than one domain for handling signals, which may have made the simulation of the parallel inputs to the BDSP chip simulator easier. Finally, Ptolemy is freely available, and there are no licensing restrictions on how many copies can be running at a time. It can be used

by any number of engineers at once. In SPW, licenses must be purchased for each subsystem (e.g. the Block Diagram Editor, the Signal Calculator), but one can download all of Ptolemy from the Internet.

That is also the reason why it was not chosen for the demultiplexer simulation. It is not officially supported in the same way that SPW is supported by Cadence. In addition, it was felt that it is not an industry standard in the same way that SPW is.

5.3.2 SHARP's Real Time Simulator

Before the decision to purchase the BDSP 9124 C source code simulator, SHARP's Real Time Simulator (RTS) was considered. This software for simulating the BDSP 9124 was distributed along with the BDSP 9124 user's guide by SHARP. Similarly, a program for simulating the BDSP 9320 address generator was distributed with the user's guide for that chip. (The 9320 address generator was not used in the demultiplexer project because it could not support a multicarrier situation in the demultiplexer.) Unfortunately, these software products were unusable for several reasons.

First, there were pages missing from the RTS simulator manual. The RTS simulator for the BDSP 9124 came with the ability to do a custom simulation and the ability to perform prepackaged examples. Of special interest for the demultiplexer project, the RTS software had an example about doing a 4096 point FFT and another example about using the chip for fast convolution. However, the pages describing how to view and use these examples were not in the manual for the RTS software. Because of this, another manual for the software was ordered. However, in that second manual, the same pages were missing. SHARP made the DSP chip and the software to simulate it. They passed the product on to BDSP for development, sales, and customer support in the early 1990s.

After that, SHARP did not support its RTS software. In fact, SHARP directed inquiries to BDSP.

Second, assuming the software could have been used without aid from the manual, the examples included with the software were limited and did not include all the kinds of mixed radix transforms needed for the demultiplexer. In an effort to use the software for custom configurations, address files were made using the 9320 chip simulator from sharp, in the format described by the manual. However, these were not accepted by the 9124 RTS simulator software. The address files that were output from the 9320 software were in binary and it was not clear what was wrong or how to correct the files so that they could be accepted by the 9124 RTS software. The format for the binary files was not documented by either manual (for the 9124 or the 9320 simulator).

Third, and far more critical, it would not have been feasible to use the RTS software with the SPW simulation. For one thing, the RTS software runs only on PCs. So to use it with SPW, data would have had to be sent over the network. The network was not designed for such intensive data transfer and was already overloaded. But even more critical was the fact that the command to run the RTS software could not be executed remotely from the workstation running the SPW simulation. Instead, the RTS software could only be executed via a .com command that invoked a menu. From this menu, it was possible to load data and coefficient files from disk and run them through the 9124 simulator. The output could be stored on disk. Worse still, this applied to only one transform, whereas each simulation of the demultiplexer involved seven chips, each performing several hundred transforms. This means that for each transform, the simulation user would have had to walk from the workstation to the PC with the RTS software, load up the new data, invoke the command to simulate, store the output, and put

that into the workstation simulation. This kind of sneaker net was not deemed a wise way to carry out the simulation.

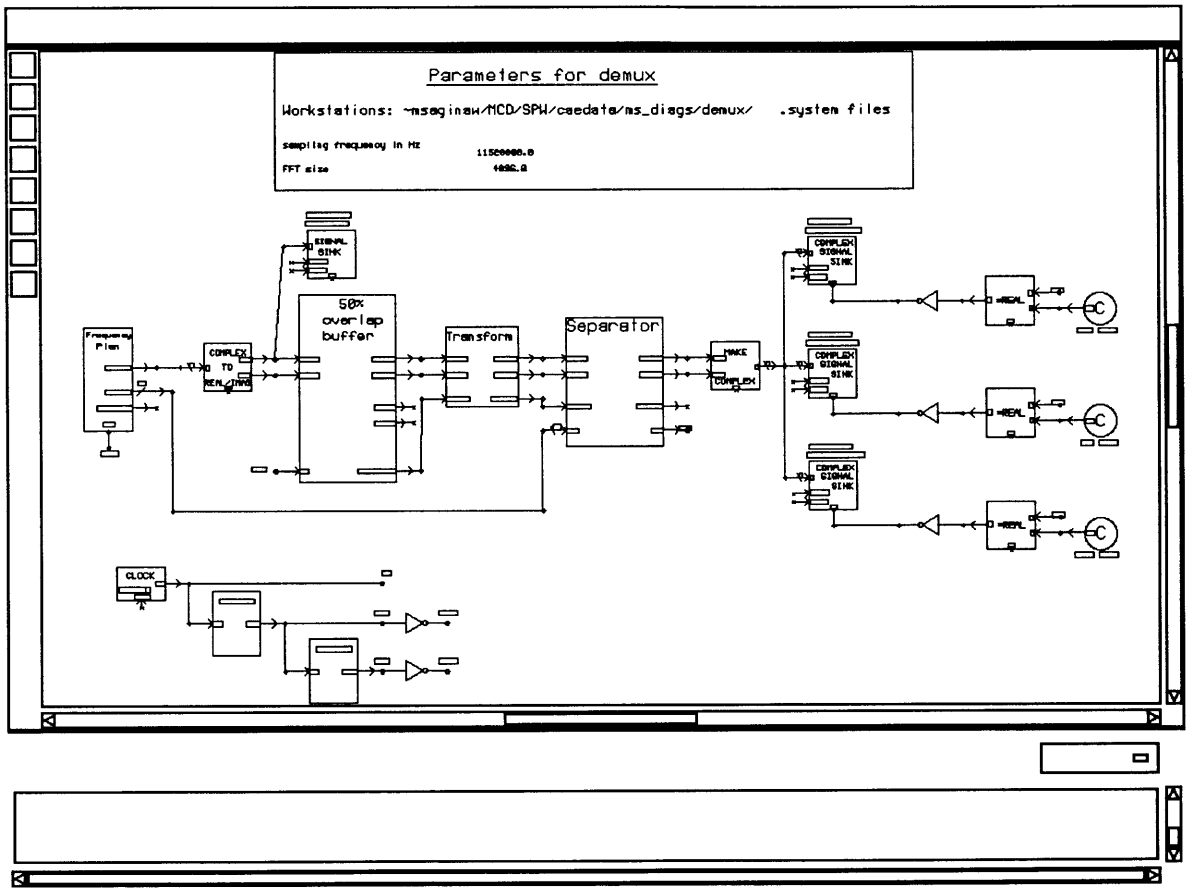
5.3.3 High Level Simulation

Before the low level simulation of the demultiplexer (complete with all the twiddle coefficient addresses and data reshuffling address, and calls to the BDSP 9124 C source code simulator) was carried out, a high level simulation was constructed. It encompassed an attempt to simulate the demultiplexer with many built-in blocks, as well as an effort to learn about SPW thoroughly enough to construct the low level simulation.

The high level simulation had the same three clocks as the fully detailed simulation and the hardware. It had an input with frequency multiplexing and a flexible plan. The input went through a 50% overlap buffer and then a Fourier transform was taken. The transform was based on built in SPW blocks. It could handle a 4096 point transform all at once and did not need external twiddle coefficients (since those complex numbers were generated by the software), digit-reversed inputs, or patterns for data reshuffling. The output was sent to a separator block. This was an exercise in writing a block that called custom C code, where that code used functions outside of the standard C libraries. Thus it was necessary to use the CGS compiler. The separator applied the square root raised cosine filters to transformed data. The output was captured by signal sinks.

The SigCalc tool was used to examine the filtered data. This was not efficient and not useful for long simulations, but it was a way to check the overall algorithm by making eye patterns and seeing that there was indeed no intersymbol interference.

Figure 5.2: High Level Simulation of the Demultiplexer in SPW



The high level simulation of the demultiplexer included multiplexed carriers at the input, a 50% overlap buffer, an FFT which was constructed from built in SPW blocks, and a custom coded block that separated and filtered the carriers.

Many steps were involved. For each carrier, the filtered data was reordered into transform order. Zeros were inserted in between for zero padding. Then the inverse transform was taken. Since the inverse transform was taken on only a select group of the filtered samples, the carrier was in that way converted to baseband. Only the middle 50% of the samples (the location of the good samples) from each transform were retained in the output. That output was compared to the corresponding input from the same carrier. By viewing both signals in magnitude-phase space, the constant phase shift between the two signals could be calculated. This phase was subtracted off from the output. Finally, eye patterns of the adjusted output were made, and they had negligible intersymbol interference. Thus it was concluded that the manipulations in the algorithm were understood.

In conclusion, the experiences with Ptolemy and SHARP's RTS were instructive but had little to do with the final, low level simulation. The high level simulation effort was a means of learning thoroughly about SPW and verifying the overall algorithm.

Chapter 6

FFT Algorithm and Addressing for the BDSP 9124

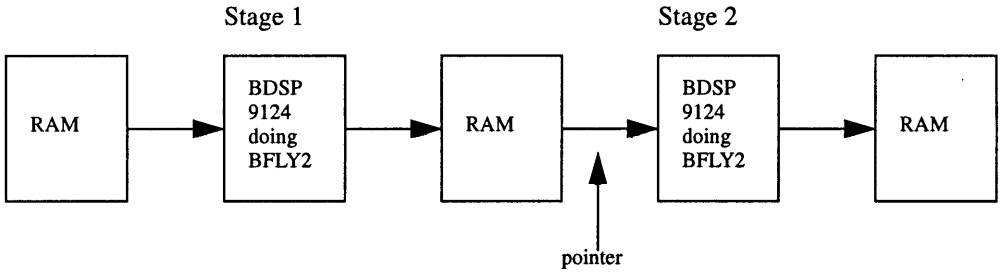
The discussion in this chapter begins with a description of the function codes of the BDSP 9124 chip that are used in this project. Then there is a more general derivation of equations needed for more complicated FFTs. The ideas are developed in an order that is based on the algebra of the transform, and not from, say, the first stage of the transform to the last. This algebra is used to derive the patterns for twiddle coefficients, data reshuffling, and digit-reversed inputs. Sample blocks of code are presented to illustrate how to create those patterns in the context of the BDSP 9124 chip being used to perform the FFT calculations. Since nearly all of the patterns are implemented via write addresses instead of read addresses, the issue of remapping is discussed. Furthermore, since the BDSP 9124 has a special structure for performing the radix-16 operations, the way to derive the special twiddle coefficients for that block is presented. Finally, all these ideas are put together in an example, and all the patterns for the 32 point transform are shown. Throughout the chapter, the notation W_N is used to represent the quantity $e^{\frac{-j\omega}{N}}$ for convenience. The built in operations in the BDSP 9124 chip that are used in this project are BFLY2, BFLY4, and BFLY16 for transforms of sizes 2, 4, and 16 respectively, and BWND2 and BWND4 for transforms combined with multiplication of the input data by coefficients. The equations describing the output-input relationships for these function codes can be found on pages 3-4 to 3-14 of [9124 UG].

6.1 Toy Example: A 4 Point FFT

The purpose of presenting this toy example is to show the general line of thought involved

in creating all the patterns for larger transforms. Furthermore, the presentation of items for larger transforms is streamlined, but the context is set in the description of this toy example. The goal of the example is to show how to compute a 4 point DFT of a sequence with the following hardware setup:

Figure 6.1: Hardware for Toy Example



Here is the equation for each of the 4 terms of the transform:

$$X[k] = \sum_{n=0}^{3} x[n] W_4^{nk} \tag{6.1}$$

However this can be rewritten as follows:

$$X[k] = \sum_{n \text{ even}} x[n] W_4^{nk} + \sum_{n \text{ odd}} x[n] W_4^{nk} \tag{6.2}$$

By introducing a new variable, this can be rewritten as follows:

$$X[k] = \sum_{m=0}^{1} x[2m] W_4^{2mk} + \sum_{m=0}^{1} x[2m+1] W_4^{(2m+1)k}$$

This can be rewritten as two 2 point DFTs:

$$X[k] = \sum_{m=0}^{1} x[2m] W_2^{mk} + W_4^{k} \sum_{m=0}^{1} x[2m+1] W_2^{mk} \tag{6.3}$$

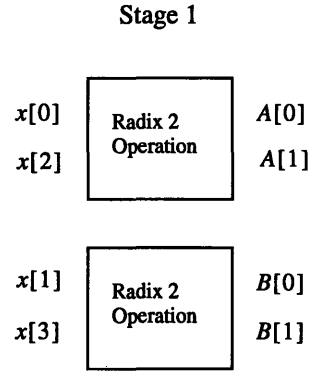
Each summation in equation (6.3) is a 2-point DFT, and the term W_4^k in front of the second summation is a twiddle factor which must be provided by the hardware to the BDSP 9124 chip after the pointer in Figure 6.1.

Intermediate variables can be defined as the output of the two 2 point DFTs in equation (6.3). The equations with intermediate variables are as follows:

$$X[k] = A[k] + W_4^k B[k]$$

A streamlined picture of the hardware to the left of the pointer now looks like this:

Figure 6.2: Calculations in Toy Example to the Left of the Pointer



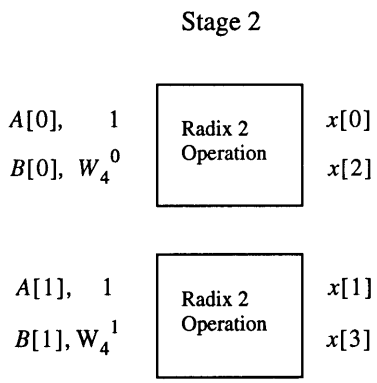
The functions $A[k]$ and $B[k]$ repeat when k gets to 2. Also, the twiddle coefficient W_4^k changes sign when k gets to 2. Using these properties, the equation about the way to combine values of $A[k]$ and $B[k]$ can be simplified. In fact, they are actually 2 point DFTs themselves, and can be computed using the same hardware as the first stage. They can be written as follows:

$$\begin{cases} X[0] = A[0] + W_4^{\,0} B[0] \\ X[2] = A[0] - W_4^{\,0} B[0] \end{cases}$$

$$\begin{cases} X[1] = A[1] + W_4^{\,1} B[1] \\ X[3] = A[1] - W_4^{\,1} B[1] \end{cases}$$

Recognizing that these equations are equivalent to two 2 point DFTs, the hardware diagram to the right of the pointer from Figure 6.1 can be presented, in streamlined form, as follows:

Figure 6.3: Calculations in Toy Example to the Right of the Pointer



Scanning the figures from left to right, there are several things to point out. First, there are no twiddle coefficients needed for the first transforms. These are pure DFTs and there is no twiddle coefficients for them called for by the algebra. Second, the output of the first stage must be reshuffled in order that it is put into the second stage properly. Third, twiddles are needed for the second stage, and the are shown alongside the data in Figure 6.3. Fourth, the output of the second stage must also be reshuffled.

In the actual hardware, the radix sizes used to perform the transforms are as shown in the following tables.

Table 6.1: Radix Combination for the FFT

size	stage 1	stage 2	stage 3
4096	16	16	16

Table 6.2: Radix Combinations for each IFFT Transform Size

size	stage 1	stage 2	stage 3	stage 4
32	2	16	1	1
64	4	16	1	1
128	2	16	1	4
256	4	16	1	4
512	2	16	4	4
1024	4	16	4	4
2048	2	16	16	4

6.2 Double Sum

The idea of breaking the large transform into smaller ones is now developed more formally and more generally. Suppose we wish to calculate an N point DFT, where

$$N = N_1 N_2 .$$

The steps are to define new indices, rewrite the DFT in terms of them, and then analyze that expression. The ideas here are from [OS], page 611. The indices are defined as follows:

$$n = N_2 n_1 + n_2 \qquad \begin{cases} 0 \le n_1 \le N_1 - 1 \\ 0 \le n_2 \le N_2 - 1 \end{cases}$$

$$k = k_1 + N_1 k_2 \qquad \begin{cases} 0 \le k_1 \le N_1 - 1 \\ 0 \le k_2 \le N_2 - 1 \end{cases}$$

Using these indices and the properties of powers of W_N, the DFT can be written as follows:

$$X[k] = X[k_1 + N_1 k_2] = \sum_{n_2=0}^{N_2-1} \left[\underbrace{\left(\sum_{n_1=0}^{N_1-1} x[N_2 n_1 + n_2] W_{N_1}^{k_1 n_1} \right)}_{G[n_2, k_1]} W_N^{k_1 n_2} \right] W_{N_2}^{k_2 n_2} \tag{6.4}$$

The inner summation represents the N_1 point DFTs that are performed in previous stages. The term $W_N^{k_1 n_2}$ is for the twiddle coefficients, and the outer summation represents the combination of the results of the smaller transforms. As with the toy example, the outer sum is actually a collection of N_2 point DFTs. The way this equation is used in this thesis is to start by considering the entire transform to be performed. The transform is broken into two pieces, where the second piece is size 2, 4, or 16. Twiddle coefficients and reshuffling patterns are generated. Then the same equation is used to work on the first of the two transforms, breaking it into one transform and a second one of size 2, 4, or 16. The process continues until the first transform is size 2, 4, or 16, and the entire transform can be performed by the BDSP 9124 chips.

6.3 Twiddle Coefficients

The twiddle coefficients come from the double sum equation. Since the outer sum is over n_2, with k_1 held constant for each of the transforms at the final stage. The exponent for the desired twiddle coefficient can be generated from the following pseudo code:

```
for k1 = 0 to (N1 - 1)
for n2 = 0 to (N2 - 1)
exponent = k1*n2
next n2
next k1
```

The twiddle coefficients stored in memory correspond to the coefficients for the largest transform size being performed. For other transforms, the exponent can be scaled by the ratio of the size of the largest transform being performed to the size of the smaller transform, for which the twiddle coefficients are desired.

6.4 Twiddles for Radix 16 Operations

The BDSP chips can perform a radix 16 operation. The reshuffling address patterns for this function are as discussed above. However, the twiddle coefficients are different. The chip performs a radix 16 operation by accepting 16 inputs, performing four radix 4 operations, and putting the results into four more radix 4 operations to obtain the final output. Although it seems as though there are 32 twiddle coefficients needed, only 16 are needed. The reason why and the way to generate the twiddle coefficients needed are explored in this section.

In the rest of the chapter, the discussion moves from the final output stage backward towards the inputs. Here, it is the opposite. Given that there is a radix 16 operation as part of the structure of a transform, this discussion moves forward through that radix 16 operation, showing how it can be recomposed as eight radix 4 operations.

First of all, if there is a radix 16 transform as part of the transform, the twiddles entering it are all powers of a single base twiddle, which will be called W_T here. This can be seen by looking at the code for generating twiddles, in which all the exponents are

multiples of a single base exponent. As we break the radix 16 transform into eight radix 4 transforms, it is crucial to note this simplification in the twiddles in the radix 16 transform.

The equation for the radix 16 operation, including the twiddles, is as follows:

$$X[k] = \sum_{n=0}^{15} x[n] {W_{16}}^{nk} {W_T}^{n}.$$

After separating this into four sums (each including every fourth point), introducing a new index variable m, pulling out twiddle factors, and reducing the base twiddle factor, we obtain:

$$X[k] = \begin{cases} \displaystyle\sum_{m=0}^{3} x[4m] {W_4}^{mk} {W_T}^{4m} \\ + \quad (W_T {W_{16}}^{k}) \displaystyle\sum_{m=0}^{3} x[4m+1] {W_4}^{mk} {W_T}^{4m} \\ + \, ({W_T}^{2} {W_{16}}^{2k}) \displaystyle\sum_{m=0}^{3} x[4m+2] {W_4}^{mk} {W_T}^{4m} \\ + \, ({W_T}^{3} {W_{16}}^{3k}) \displaystyle\sum_{m=0}^{3} x[4m+3] {W_4}^{mk} {W_T}^{4m} \end{cases}$$

The four summations are the first four radix 4 operations. They all have the same twiddles: ${W_{\frac{T}{4}}}^{m}$. Thus it makes sense that the BDSP 9124 chip only requires one copy of those coefficients. It requires 4 numbers instead of 16.

The twiddles for the second set of four radix 4 operations can be seen by doing manipulations similar to those used in the 4 point toy example. First, some intermediate functions are defined and $X[k]$ is rewritten as follows:

$$X[k] = A[k] + (W_T {W_{16}}^{k}) B[k] + ({W_T}^{2} {W_{16}}^{2k}) C[k] + ({W_T}^{3} {W_{16}}^{3k}) D[k]$$

Then the twiddles can be seen by writing the first four values of $X[k]$:

$$
\begin{aligned}
X[0] &= A[0] + (W_T W_{16}^{\ 0})B[0] + (W_T^{\ 2} W_{16}^{\ 0})C[0] + (W_T^{\ 3} W_{16}^{\ 0})D[0] \\
X[1] &= A[1] + (W_T W_{16}^{\ 1})B[1] + (W_T^{\ 2} W_{16}^{\ 2})C[1] + (W_T^{\ 3} W_{16}^{\ 3})D[1] \\
X[2] &= A[2] + (W_T W_{16}^{\ 2})B[2] + (W_T^{\ 2} W_{16}^{\ 4})C[2] + (W_T^{\ 3} W_{16}^{\ 6})D[2] \\
X[3] &= A[3] + (W_T W_{16}^{\ 3})B[3] + (W_T^{\ 2} W_{16}^{\ 6})C[3] + (W_T^{\ 3} W_{16}^{\ 9})D[3]
\end{aligned}
$$

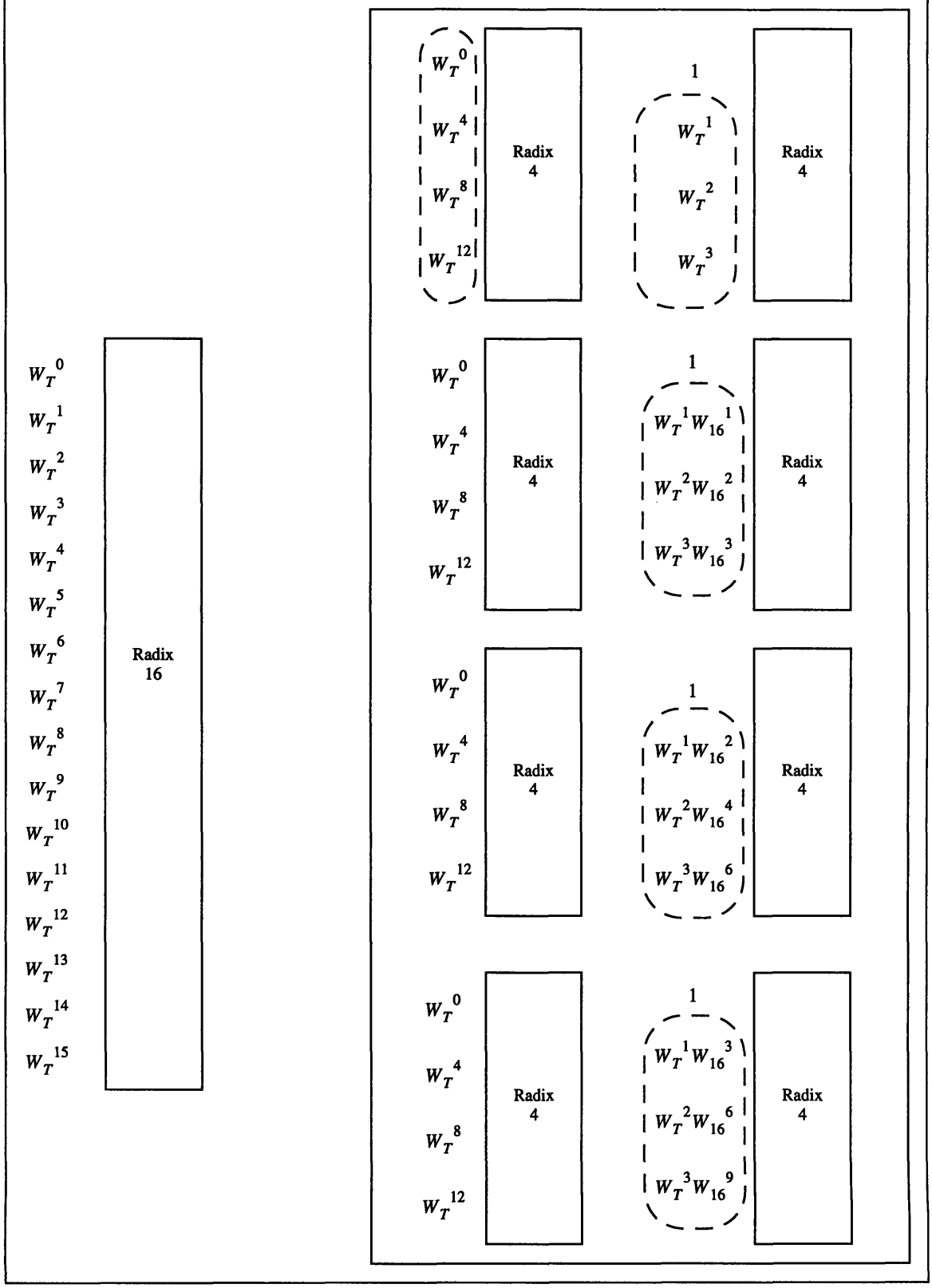
The other 12 terms fill out the four 4 point DFTs, due to the periodicity of the intermediate functions when k is a multiple of 4, and due to the properties of W_{16} raised to powers of 4. From this list, the twiddle coefficients for the second stage are can be read off. The twiddle coefficients for the first of the four radix 4 operations is in the row with $X[0]$, the twiddles for the second of the four operations is in the next row, and so forth. The conversion from twiddles for a radix 16 operation to twiddles for 8 radix 4 operations is shown in Figure 6.4. In particular, the 16 coefficients that are required by the chip are circled.

6.5 Reshuffling

The double sum also shows that there is a need for reshuffling the data between stages and at the output of the transform. Reshuffling is necessary because the stage with N_1 point DFTs produces outputs $G[n_2, k_1]$, with n_2 constant in each radix operation and k_1 incrementing. However, for the next stage, each block of radix N_2 block requires inputs with a fixed k_1 and an index of n_2 that increments. The code required for input reshuffling is therefore:

```
for k1 = 0 to (N1 - 1)
for n2 = 0 to (N2 - 1)
read_adr = N1*n2 + k1
next n2
next k1
```

Figure 6.4: Equivalence of a Radix 16 Operation and Eight Radix 4 Operations



6.6 Digit-Reversed Input

In the toy example of a 4 point FFT, the inputs to the first stage were not in sequential order. The proper order for them can be seen from the way equation (6.1) was rewritten as equation (6.2) or equivalently, as equation (6.3). In general, if an N point DFT is being broken down into N_1 point DFTs, followed by N_2 point DFTs, then the read order for the inputs is given by the following code:

```
for n2 = 0 to (N2 - 1)
for n1 = 0 to (N1 - 1)
read_adr = N2*n1 + n2
next n1
next n2
```

When the N_1 point DFTs are broken down into smaller radix operations, it is necessary to perform another, similar manipulation on the order of the inputs. The previous stages are dealing with transforms that have fewer points. The patterns are generated by the code and then they repeat, with an offset address tacked on. Note that it is necessary to compute the digit-reversed addresses for each of the two radix 4 operations when the radix 16 function code is used.

6.7 Multiple Mappings

The digit-reversed patterns require more than one mapping. The code to implement one mapping after another is straightforward. Suppose one routine has been run to generate "pattern1," and another routine has generated "pattern2". Then the following pseudocode will perform the desired mapping:

```
for i = 0 to (N-1)
k = pattern2(i)
read_adr = pattern1(k)
next i
```

6.8 Remapping

The patterns for manipulating data are generated as read addresses. However, in the hardware, most of the patterns are implemented as write addresses. For a few special patterns, the read addresses are the same as the write addresses but in general they are different. Therefore, it is necessary to do a remapping.

Suppose that the patterns are in an array called "read," which has N. Then the following pseudocode will create the proper write addresses:

```
for i = 0 to (N-1)
index = read(i)
write(index) = i
next i
```

6.9 Example: Patterns for the 32 Point Transform

Taking the 32 point transform as an example, all the algorithms developed above are used here to generate the proper patterns for digit-reversed inputs, twiddle coefficients, and data reshuffling. Table 6.2 shows that the 32 point transform is performed by a radix 2 operation (performed 16 times), followed by a radix 16 operation (performed 2 times). This fits into the pseudocode by assigning N_1 to be 2 and N_2 to be 16. The results of creating all the patterns are shown in Table 6.3.

Table 6.3: Derived Patterns for the 32 Point Transform

Index	"Stage 0"		Stage 1			Stage 2		
	RAM		BDSP 9124	RAM		BDSP 9124	RAM	
	Digit-Reversed Input		Twiddle Addresses	Reshuffling Addresses		Twiddle Addresses	Reshuffling Addresses	
		remapped			remapped			remapped
	read adr.	write adr.		read adr.	write adr.		read adr.	write adr.
(col. 1)	(col. 2)	(col. 3)	(col. 4)	(col. 5)	(col. 6)	(col. 7)	(col. 8)	(col. 9)
0	0	0	0	0	0	0	0	0
1	16	8	0	2	16	0	16	2
2	4	16	0	4	1	0	1	4
3	20	24	0	6	17	0	17	6
4	8	2	0	8	2	0	2	8
5	24	10	0	10	18	0	18	10
6	12	18	0	12	3	0	3	12
7	28	26	0	14	19	128	19	14
8	1	4	0	16	4	256	4	16
9	17	12	0	18	20	384	20	18
10	5	20	0	20	5	256	5	20
11	21	28	0	22	21	512	21	22
12	9	6	0	24	6	768	6	24
13	25	14	0	26	22	384	22	26
14	13	22	0	28	7	768	7	28
15	29	30	0	30	23	1152	23	30
16	2	1	0	1	8	0	8	1
17	18	9	0	3	24	256	24	3
18	6	17	0	5	9	512	9	5
19	22	25	0	7	25	768	25	7
20	10	3	0	9	10	64	10	9
21	26	11	0	11	26	128	26	11
22	14	19	0	13	11	192	11	13
23	30	27	0	15	27	192	27	15
24	3	5	0	17	12	384	12	17
25	19	13	0	19	28	576	28	19
26	7	21	0	21	13	320	13	21
27	23	29	0	23	29	640	29	23
28	11	7	0	25	14	960	14	25
29	27	15	0	27	30	448	30	27
30	15	23	0	29	15	896	15	29
31	31	31	0	31	31	1344	31	31

Digit-Reversed Input

The sum for the entire 32 point DFT can be rewritten as a combination of 4 different 8 point DFTs, in the same way that equation (6.1) was rewritten as equations (6.2) and (6.3). The pseudocode for digit-reversed input addresses can be used to generate a first set of digit-reverse values.

Next, each 8 point DFT can be rewritten as a combination of 4 different 2 point DFTs. The pseudocode for digit-reversed input addresses can be used again to generate a second set of digit-reversed values. The two patterns can be combined using the pseudocode for multiple mappings, and the result is shown in column 2 of Table 6.3. The values in column 2 are the final read addresses for digit-reversal, and they can be remapped to the write addresses in column 3.

Twiddles for Stage 1

The twiddle coefficients for first stage are all 1, or W_{32}^{0}. Thus the address to get the right exponent is 0, as displayed in column 4 of Table 6.3. In fact, the twiddle coefficients for the first stage are always all 1. This fact is shown by example in equations (6.2) and (6.3). In those equations, the summations for the first stage have no twiddles that are not already part of the BDSP 9124 chip radix operations. The twiddle coefficients in equation (6.3) are for the second stage of computation.

This rule that the twiddle coefficients to the first stage are always 1 is very handy. It means that when the first stage is a BWND2 or a BWND4 operation, the combination of non-twiddle and twiddle coefficients is trivial, which makes it easier to work with filter coefficients. The exception to this rule is when the first stage is radix 16, and it is in a sense two stages. Then the twiddle coefficients are not all 1.

Reshuffling Between Stages 1 and 2

After the operations in stage 1, it is necessary to reshuffle the data in preparation for stage 2, and the pseudocode can be used to generate the addresses. The resulting pattern is shown in column 6. The multiple mapping of column 5 followed by column 6 is shown in column 7, and is simply the same as column 5. Once these reshuffling steps have been performed, the read addresses can be remapped to write addresses using the pseudocode. The result is shown in column 6.

Twiddles for Stage 2

The twiddles for stage 2 can be generated according to the pseudocode, and then must be altered for the radix 16 operation. The alterations necessary are illustrated in Figure 6.4. The values shown in Table 6.3 in column 7 are scaled exponents. They are scaled because the twiddles for all the IFFTs (all the transform sizes except 4096) are stored in a common memory. All the twiddles for the 2048 point transform are in that memory. In order to get only the subset needed for the 32 point transform, it was necessary to scale all the exponents for the twiddles by 2048/32, or 64.

Reshuffling After Stage 2

Finally, reshuffling must be performed after stage 2, according to the pseudocode. The resulting read addresses are shown in column 8. They are remapped to write addresses, as shown in column 9.

Chapter 7

Software for Generating Address Patterns

The patterns needed for the demultiplexer are patterns for twiddle coefficients and for data reshuffling. In the FFT, most (but not all) of these patterns are independent of the frequency plan. In the IFFT, the patterns do depend on the frequency plan. Software on a host PC needs to be given the frequency plan and generate patterns accordingly. All this software has been written and tested. It takes about 5 seconds to run. File names were made eight characters or fewer, so that they can be transferred from the unix workstations where they were made to a PC. The programming was done in ANSI C. The information given about each carrier is the transmitted bit rate, the allocated bandwidth, and the center frequency.

Two issues which seemed to be unrelated before the software was written were actually intimately tied up with one another. These were the different latencies of the different BDSP 9124 function codes being used, and the addresses for data reshuffling. They are related in the sense that addresses for data reshuffling take into account the different latencies for the different function codes.

Many of the address manipulation ideas that are discussed here were described in general in Chapter 6. Here, they are extended so that they apply to the specific configuration of the demultiplexer hardware. The tables below show the types of addressing which are performed by the software at each stage. Then these manipulations are described in subsequent sections.

7.1 Input to FFT: 50% Overlap

The samples from the A/D converter are pumped sequentially into data RAMs at the clock rate, F_S. The data is read out at $2F_S$, and the RAMs are used as circular buffers. This is the only place where the address pattern is used as a read address. In subsequent stages of the project, patterns that are generated must be suitable for use as write addresses.

The patterns for reading data out are independent of the frequency plan. They involve two manipulations: digit reversal and overlapping. The overlapping is achieved by copying the digit-reversed patterns and adding an offset.

Table 7.1: Addressing for Circular Buffer

RAM write	RAM read
sequential	overlapping digit-reversed

Digit Reversal

The digit reversal depends on the radix structure of the transform operation. For the FFT, the radix structure is fixed; all three stages perform a radix 16 operation, and the digit-reversed addresses must be computed according to the pseudocode in section 6.6.

Overlapping

In order to implement the overlapping, the PROMs which contain the patterns for this stage have 16,384 address locations. They repeat a pattern of 4096 addresses, slightly modified, four times. They have the patterns for reading "AB," "BC," "CD," and "DA." First, the software computes the digit-reversed pattern for one set of 4096 points, using addresses 0 through 4095. The next 4096 addresses are equal to the first 4096, with 2048

added to each address. This is for reading the "BC" parts of the buffer. The next two sets of 4096 points are similarly copied from the first and shifted.

7.2 FFT Addressing

The addressing for the first two stages of the FFT does not depend on the frequency plan. Thus the address patterns are stored in PROMs

Table 7.2: Addressing for FFT Stage 1

BDSP 9124 coefficients	RAM write	RAM read
twiddle coeffs.	BFLY latency: fixed	sequential
	reshuffled	
	remapped	

Table 7.3: Addressing for FFT Stage 2

BDSP 9124 coefficients	RAM write	RAM read
twiddle coeffs.	BFLY latency: fixed	sequential
	reshuffled	
	remapped	

7.2.1 Twiddles in the FFT

Twiddle Factors for the BDSP 9124 Chip

The twiddle factors in the FFT are lookup tables for the functions $\cos\left(\frac{-2\pi k}{4096}\right)$ and $\sin\left(\frac{-2\pi k}{4096}\right)$, where k is an integer from 0 to 4095. Note that the argument of the functions can be thought of as an angle that *decreases* from 0 down to almost -2π. Alternatively,

the lookup tables are x and y coordinates on a unit circle as a vector from the origin to the point (1,0) sweeps around in the *clockwise* direction. The cosine function is the same no matter whether the vector moves clockwise or counterclockwise, but the sign of the sine function is reversed, a fact which is included in the software. The literature from BDSP describes each function code and for the BFLY2, BFLY4, and BFLY16 codes. The BDSP literature says that there is a negative sign table assumed for the imaginary data of a forward FFT. However, this is misleading. The negative sign table is provided in the simulation, and if the BDSP 9124 simulation software is an accurate simulation of the hardware, then the6.4 negative sine table must indeed be provided, and is not assumed.

FFT Twiddle Patterns Combined Together Into Final Pattern

In the IFFT, the twiddle coefficients are stored in order and they are called in the order they are needed according to a pattern. In the FFT, however, the pattern in which the twiddle coefficients are needed does not depend on the frequency plan, so a few steps can be rolled into one memory device. One memory device, which is read sequentially, can hold the twiddle coefficients, and it can also hold them in the order they are needed for the transform. The twiddle coefficients are generated according to the pseudocode in section 6.3 and 6.4.

7.2.2 Reshuffling in the FFT

The data reshuffling patterns follow the pseudocode in section 6.5 and the remapping is done according the pseudocode in section 6.8.

7.3 From FFT Stage 3 to IFFT Stage 1

At this crucial stage of the demultiplexer, the addressing manipulations which need to be performed are numerous. First, the output of the FFT must be handled. Then, there are many manipulations that must be performed on the samples for each carrier. Finally, the data is put into memory, and the address to that memory increments in a staccato manner, instead of incrementing steadily. The patterns for all these come from the software and are downloaded into the RAMs for each frequency plan. As in nearly all other stages of the hardware, the addressing is controlled via the write addresses. Since there are so many manipulations to be done, and since they are typically discussed in the literature in terms of read addresses, all the addressing is done with read addresses, which can be cascaded using the pseudocode for multiple mappings in section 6.7. The last step is to remap these to write addresses.

The number of samples from each FFT is 4096. In contrast, the number of cycles in the IFFT is 8192. At the end of the FFT, 4096 samples are pumped into memory. The IFFT needs to read in 8192 samples. This is feasible because the IFFT uses a clock rate of $4F_S$, whereas the FFT uses a clock rate of $2F_S$. However, the question arises of where the extra samples come from.

These extra samples consist of the zero padding. While these are read into the IFFT, the read address to the memory with FFT output samples remains constant. When it is necessary to read a new sample from the FFT output, the read address to the memory is incremented. In this context, the manipulations on samples of the FFT are described.

Table 7.4: Addressing for FFT Stage 3

BDSP 9124 coefficients	RAM write	RAM read
twiddle coeffs.	BFLY latency: fixed	staccato
	reshuffled	
	natural freq. order	
	for each carrier:	
	transform order	
	zero-padding	
	digit-reversed	
	remapped	
	crunched	

7.3.1 Manipulations for the Entire FFT Output

Reshuffling

The data emerge from the FFT butterflies and are not in sequential order. They are in staggered order since the actual hardware is serial, not parallel. It is necessary to reshuffle them.

Natural Frequency Order

The data at the output of the FFT is in transform order, meaning that the first sample is the sample at frequency 0. The next samples are for positive frequencies, and the last half of the samples are for negative frequencies. These must be rearranged into natural frequency order so that carriers can be separated. For instance, it is quite possible that a carrier straddles the 0 frequency. Either the data must be explicitly reordered or the subsequent data manipulation must take into account the fact that the data is in transform order. In this implementation, explicit reordering is performed to simplify subsequent stages.

7.3.2 Manipulations for Each Carrier

The software knows the frequency plan. Once the data for the entire FFT are handled, the software generates patterns to work with samples from each individual carrier. By handling the samples for each carrier this way, the carriers are each converted to baseband. The beat frequencies are eliminated, assuming that the local oscillators on the transmitter and receiver are operating at the proper frequency.

The individual carriers have zero padding for the IFFT, but there is room for all that because there are 8192 cycles in the IFFT and only 4096 cycles in the FFT.

The software inserts the FFT samples corresponding to the allocated bandwidth from one carrier. The decision was made arbitrarily to include the smallest frequency of the allocated bandwidth and all the samples up to the highest frequency, but not including the sample for the highest frequency, for that is the sample at the lowest frequency for the next carrier. This has a slight effect because some of the carriers exceed their allocated bandwidths and their data shaping filter coefficients are greater than 0.0 at the edges of the allocated bandwidths.

Thus a major shift occurs here. Previously, all manipulations worked with the entire FFT. Here, the software is using its knowledge of the frequency plan to select samples from the allocated bandwidths. The software has to know the location and bandwidth of the band for the carriers to do this.

Now that each carrier has been separated, there are many more manipulations to be done. The software works on each carrier individually and retains the proper offset addresses based on the number of frequency samples in each carrier.

Transform Order

At this point, each individual carrier is in natural frequency order. However, to prepare the sample for the IFFT, they must be placed in transform order.

Zero Padding

At this point, the software needs to do zero padding. This is because the FFT samples are selected according to the allocated bandwidths; the number of FFT samples for each carrier is in general not a power of two. However, the size of the IFFT must be a power of two in the BDSP 9124 chips. Thus the sequence of FFT samples is extended by adding zeros to it. The number of zeros added to the FFT sequence is designed to meet the constraints of the demodulator, whose input must be between 2 and 4 samples per symbol.

The zeros are added by using a coefficient of 0.0 in the first stage of the IFFT. When it is time to add a zero, the address to the data memory in the FFT is held constant. At the same instant, the RAM under the coefficients calls the coefficient from the coefficient memory at address 0.

Another option for implementing the zero-padding was considered but rejected because of uncertainty in the BDSP 9124 chip specifications. That was to implement the zero padding by using the BDSP 9124 chip's data zero in (DZI) pin. When the signal to this pin is active, the data entering the BDSP chip is nullified. However, the source code for the BDSP 9124 simulator was examined and a concern about this approach was raised. The simulator takes inputs that are in groups of 16 (for the BFLY16 function code) or in groups of 4 (for all other function codes). In the simulator, when a DZI signal is received, the entire group of 16 or 4 data points are nullified. That is not the behavior desired in hardware. If the software is an accurate simulation of the hardware in this detail, then the DZI pin is not suitable. Perhaps the software does not match the hardware in this detail,

and the hardware DZI pin would have been suitable. However, either method requires one control bit, and where one strategy of multiplication by a coefficient of 0.0 is well understood, the other is uncertain and was avoided.

Digit Reversal

Since the data for each carrier is about to enter a transform, it must be digit-reversed since this is required for the fast transform algorithm. Digit reversal depends on the structure of the radices in the transform. Therefore, for each carrier, the radix structure must be looked up and the digit-reversal computed accordingly.

Remapping

Since the patterns for handling the data are implemented via write addresses, and all the previous work has been done with read addresses, the patterns must be remapped for use as write addresses.

Crunching

Since the number of cycles in the IFFT is 8192 but the number points at the output of the FFT is only 4096, the software needs to do crunching. It needs to take the patterns it generated for the 8192 points in the IFFT and extract only the addresses having to do with the FFT samples. It needs to exclude those address having to do with zero padding. It is left with only the addresses for FFT points that fall within the allocated bandwidth of the carriers used in the frequency plan. That is far fewer than 4096 points. In fact, it is fewer than 3200 points, since the 4096 points of the FFT correspond to a 11.52 MHz band, but the input only has signals on a 9 MHz band.

Staccato Read-Address-Increment Signal

While the software crunches down the addresses for the data, it builds up a sequence of 1's and 0's for the read-address-increment signal. This read-address-increment signal takes into account all the manipulations that were performed on the addresses, so it is not a steady block of ones followed by zeros. When the samples for a carrier are read out, the read-address-increment bit changes from 1 to 0 in an irregular pattern. Since it is not one steadily but only in bursts, it is called the staccato bit.

7.4 Filter Coefficients in the First Stage of the IFFT

The coefficients in the first stage of the IFFT are not twiddle coefficients. They are coefficients for the data shaping filters. The filter coefficients have to match the input to the IFFT: they must be in transform order, they must include the zeros for zero-padding, and they must be digit-reversed.

Table 7.5: Addressing for IFFT Stage 1

BDSP 9124 coefficients	RAM write	RAM read
filter coeffs.	BFLY latency: variable	sequential
transform order	reshuffled	
zero-padding	remapped	
digit-reversed		

Transform Order

The coefficients must come out of the memories and up to the BDSP 9124 chips in transform order. Only half the coefficients are stored, since they are symmetric. First,

they are read in one direction. After the zero-padding is inserted, the coefficients are read in the opposite direction.

Zero-Padding

For each carrier, the IFFT is made of some samples from the FFT and some samples that are zero pads. When a zero-pad occurs, the read-address for the memory with FFT output samples does not increase, and the 0^{th} address for the coefficient memory is called, in order to multiply the input to the BDSP 9124 chip by 0.0.

Digit-Reversal

The data at the start of the IFFT are digit-reversed, and the coefficients must also be digit-reversed. The digit-reversal pattern depends on the radix structure of the transform being performed. The filter coefficients are all stored in one memory in hardware, so the address patterns must call them in the proper order. Also, each filter corresponds to a known transmitted bit rate, and a known IFFT size. Therefore, in retrospect, the software could have been written to access a file with the address patterns pre-computed, including the transform order arrangement, the zero padding, and the digit-reversal. However, as the software evolved, it was written to recompute the filter coefficients in floating point, make decisions about which addresses to include for filter values of 1.0, intermediate filter coefficient values, zero padding, intermediate coefficient values, and values of 1.0, and then to digit-reverse those addresses. However, this does not take very much time so the difference between the implemented software and the possible software is worth pointing out but not worth changing.

Twiddle Coefficients All 1

Finally, there is the fortuitous fact that the twiddle coefficients to the first stage of the IFFT are all 1, so the operation to combine the twiddle coefficients with the filter coefficients is trivial. This is in contrast to the twiddle coefficients for the first stage of the FFT, which were not all 1. That was because the BFLY16 function code was used in the FFT. In that function code, there are actually two sets of four radix-4 transforms being performed. That second set of transforms was why the twiddles were not all 1.

In the IFFT, where the window multiplication function codes are used at the first stage, the only window multiplication function codes available are BWND2 and BWND4. There is no BWND16 because of the unique arrangement of coefficient inputs for that function code.

7.5 IFFT Addressing

7.5.1 Twiddles for IFFT Stages 2, 3, and 4

In the IFFT in stages 2, 3, and 4, the coefficients are twiddle coefficients. They are computed according to the formulas from the DSP background chapter and the twiddles for radix-16 structures are computed according to the formulas from this chapter, where the specific implementation of the BFLY16 function is discussed.

The twiddle coefficients for the transform being performed, at the particular stage where the memories are, are culled up from a file in memory and put into the pattern. If there is a second carrier, then there follows the addresses for the proper twiddle coefficients. Finally, when there are no more carriers, the pattern for twiddle addresses

simply calls the 0 address until the end of the 8192 point cycle. These don't do anything since the StartStop signal on the chip is not active.

The twiddle coefficients stored in memory for IFFT stages 2, 3, and 4 are for a 2048 point transform. For smaller transforms, the appropriate set of twiddle coefficients is a subset of the coefficients for a 2048 point transform. The 2048 coefficients correspond to trigonometric lookup tables for an angle that moves clockwise around the unit circle in increments of $2\pi/2048$. For a smaller transform, the increments of the angle are proportionately larger. Therefore, the same twiddle coefficients can be, and are, reused for all 2048 point and smaller transforms.

7.5.2 Data Reshuffling in IFFT Stages 1, 2, and 3

Table 7.6: Addressing for IFFT Stage 2

BDSP 9124 coefficients	RAM write	RAM read
twiddle coeffs.	BFLY latency: variable	sequential
	reshuffled	
	remapped	

Table 7.7: Addressing for IFFT Stage 3

BDSP 9124 coefficients	RAM write	RAM read
twiddle coeffs.	BFLY latency: variable	sequential
	reshuffled	
	remapped	

Incorporate 9124 Chip Latency

The first thing the data reshuffling addresses must do is to handle the BDSP 9124 chip latency. This depends on which function code is used. Thus the software has a lookup table. The software knows the frequency plan, so it knows the sizes and radix structures and function codes for the IFFTs. Therefore it can refer to a table and look up the chip latency for the function code being used.

For increasing addresses in the reshuffling pattern, at first there is no pattern. Once the latency has been waited out, the addresses of the reshuffling pattern have contents that are the write addresses for the BDSP 9124 chip outputs.

Reshuffling

The contents for each transform are read from a file which has the pattern for the transform to be run individually. The way the software puts them all together is by using offset addresses.

Remapping

The reshuffling addresses must be remapped since the hardware uses write addresses instead of read addresses to implement data reshuffling. The files that the software refers to for each individual transform already have the patterns in remapped order.

Data Scale Factor Output in IFFT Stages 1, 2, 3, and 4

From the BDSP 9124 chips, one clock cycle when the data scale factor output (DSFO) is valid is after the last output data point has been pumped out. This value tells by what power of 2 the output of the chip needs to be scaled down in order to prevent overflow in the next BDSP chip. It is meant for architectures with cascaded BDSP 9124 chips. The

DSFO value can serve as the data scale factor input (DSFI) input to the next BDSP 9124 chip. That chip examines the DSFI value and the function code. Different function codes lead to different combinations of the input data and different amounts of overflow that are possible. Therefore the next BDSP 9124 chip considers both the DSFI value and the function code and decides by what power of two it needs to scale down the input before carrying out the function code.

The original plan for the project was to read the DSFO from one chip into the data RAMs and then read it into the next BDSP 9124 chip. The simulation software and SPW block diagram were developed with that plan in mind. The DSFO came out *after* the last data value. At the corresponding address in the data reshuffling coefficients, the contents caused the DSFO to be written into the data RAM *before* the addresses where the data was placed.

There was to be a switch in the hardware to accommodate the DSFO. In the simulation, the switch's two inputs were the BDSP 9124's imaginary output and the BDSP 9124's DSFO value. The output of the switch fed into the ram that held the imaginary data. Most of the time, the switched control caused the BDSP 9124 imaginary output to be connected through to the data RAM. However, after the last data value of a transform the control bit caused the switch to change and connect the BDSP 9124's DSFO to the memory RAM, allowing the DSFO value to be fed through. At that moment, the address in the reshuffling pattern causes the DSFO value to be written to a location before the other transform outputs.

However, the extra switch in the circuit caused an unacceptably large delay and has been removed from the hardware design. Instead, the DSFO is pumped into a first-in-first-out buffer (FIFO) whose value is then fed into the BDSP 9124 in the subsequent stage.

The software for the old system can remain the same for the new hardware, since it will now just put a meaningless value before the data in the data ram.

7.6 Addressing for 50% Discard

The data reshuffling addresses for IFFT stage 4 are very similar to the reshuffling for stages 1, 2, and 3.

Table 7.8: Addressing for IFFT Stage 4

BDSP 9124 coefficients	RAM write	RAM read
twiddle coeffs.	BFLY latency: variable	sequential
	remapped	
	50% discard	
	remapped	

The write addresses must include a reshuffled pattern since this is the final stage of a transform. The write addresses help implement the 50% discard. They write the bad samples to addresses that are so high that they will not be looked at by the demodulator. Finally, the patterns must be remapped for use as write addresses.

7.7 Tests of Patterns

As the software was being written, many tests were performed in order to ensure that what was being done was right. The entire software project almost certainly would never have come together if these tests had not been performed and the results examined to root out bugs and ensure proper operation at each stage along the way.

7.7.1 Checking Each Transform Individually

Each size of transform was tested individually to verify the patterns for digit-reversal at the input, twiddle coefficients, data reshuffling coefficients, and staggering at the output.

The C source code simulator for the chip was so long in coming that these tests were actually run with an ad hoc floating point simulator of the simulator, the major function of which was called "simsim." It was written based on the description of the function codes BFLY2 and BFLY4, and the equations for them in the BDSP 9124 User's Guide. The BFLY16 function code was also included, and its algorithm consisted of eight calls to the BFLY4 function. The MOVD function was also simulated, with outputs that were equal to inputs.

These four function codes themselves were tested. The MOVD function was tested by making sure the outputs were equal to the inputs. The BFLY2, BFLY4, and BFLY16 functions were tested by taking two 2, one 4, and one 16 point transforms, respectively, and testing the output against the equations and the output of SPW's built-in mechanisms for computing FFTs. Each chip received input data, coefficients, and a function code, and each produced output.

Once the operation of simsim was verified, it was used for transforms from size 32 through 4096. A cascaded architecture of four chips was used for the simulation of the first seven transforms. An architecture of three chips was used for the simulation of the 4096 point transform.

The same input that was fed into these systems was also put through a built in FFT operation by SPW. The outputs were compared and when they were not identical, the patterns for digit reversal, twiddle coefficients, data reshuffling, and staggering were

examined and corrected. The small transforms were tested first, and insights from fixing those patterns were applied to fixing the patterns for the large transforms.

These tests did not include the BWND2 or BWND4 functions, any timing parallelizer or serializer, the transfer of data from the FFT to the IFFT, or the 50% discard at the output. The test of the 4096 point transform included a test with blocks to produce a 50% overlap at the input. Also, none of the tests for sizes 32 through 2048 were explicitly testing IFFTs. The assumption was that if the FFT test worked, then the IFFT would work once the conjugation of the input and output were in place. Fortunately, this was valid and there were no snags about this point.

When the C source code simulator for the BDSP 9124 did arrive, it was flawed but examined and fixed and tested. Once the serializer code was in place, the exact C source code simulator was used in the same way that simsim was used; namely, to verify the digit-reversal, twiddle coefficient, data reshuffling, and staggering patterns. Since those patterns were known to be working, this was really a test of whether the C source code for the BDSP 9124 was being successfully integrated into the SPW environment and whether the parallelizer and serializer in the BDSP 9124 SPW block were working properly.

7.7.2 End to End Simulation Tests

Finally, after including a 50% overlap buffer, arranging to transfer the data from the FFT to the IFFT, taking steps to allow the IFFT to work with many carriers, and including the 50% discard, the demultiplexer output was fed into the demodulator. After a lot of effort, the bugs associated with these last stages were rooted out one by one until the demodulator emulation software could recover the input data for any chosen carrier. Then noise was added and the signal-to-noise ratio was calibrated carefully.

This end-to-end simulation, from the simulated input through the demultiplexer and through the demodulator, provided a test on all the demultiplexer work because the resulting bit error ratio (BER) curves look as expected. The degradation due to a single carrier passing through the demodulator alone was known. The demultiplexer was using so much bit precision that it was not expected to add much degradation.

Chapter 8

Input to Demultiplexer

The input to the demultiplexer was painstakingly simulated. First, data for many baseband carriers was generated. These carriers correspond to a few of the rows on the spreadsheet of INTELSAT IBS and IDR carrier specifications. In the simulation itself, these baseband carriers were modulated to frequencies within the 9 MHz band that the demultipexer accepts. They were added together, and then passed through a partial simulation of part of the analog front end. Signals at radio frequencies were not simulated.

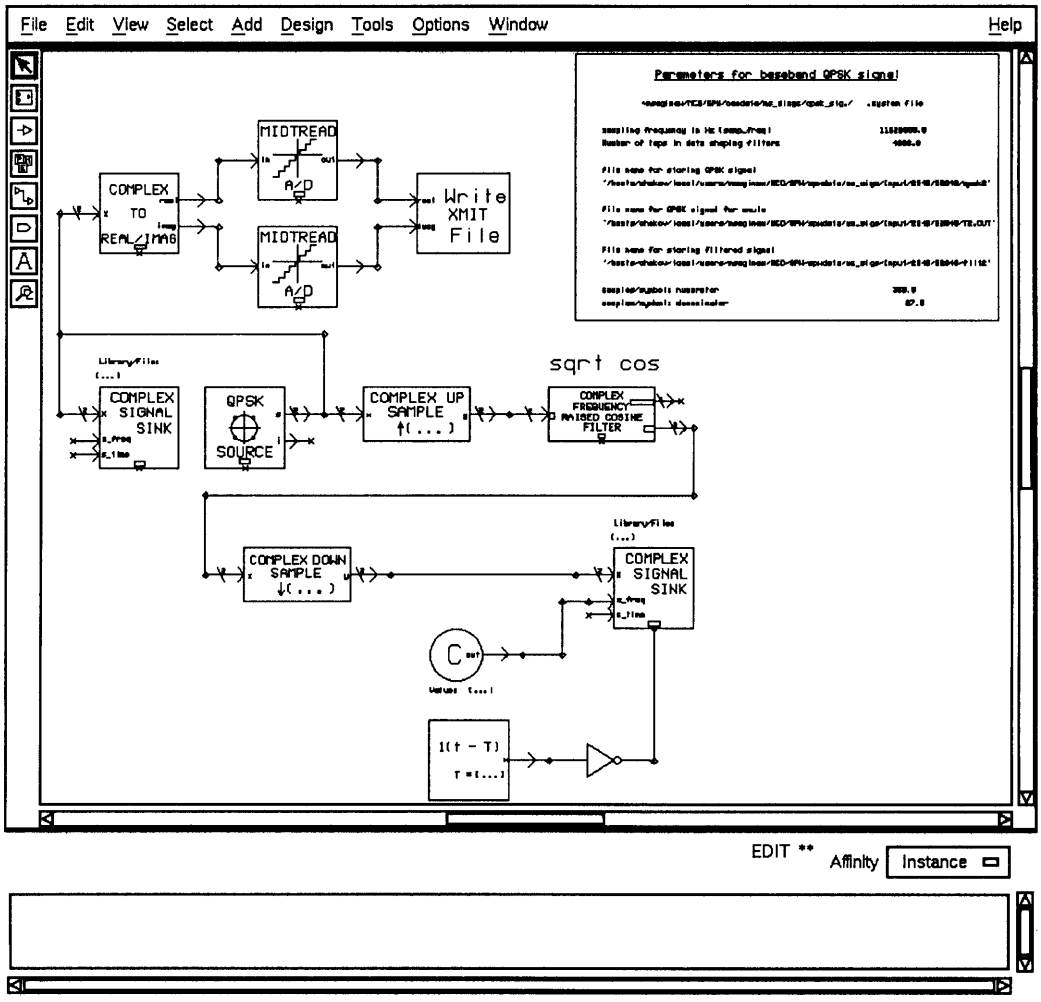
In the partial simulation of the analog front end, the signals were sent through a lowpass anti-aliasing filter. They were then processed by an A/D converter, which converted the signal amplitudes from highly precise double floating point numbers to amplitudes that were quantized to 8 bits (256 levels).

Signal sink blocks in SPW were used as oscilloscopes to record the input signals so that they could be examined. Signal-to-noise ratios were calculated and the total levels were monitored to make sure there was no danger of saturation or cutoff in the A/D converters.

8.1 Generating Baseband Data

The generation of baseband data entailed a partial simulation of a modulator. First, a bit stream was created. It is passed through a data shaping filter and stored in a file for later use.

Figure 8.1: Generating Baseband Data for a Carrier



Random data patterns are generated and stored. They are also filtered with square root raised cosine filters. Due to the upsampling, the inputs to the filter are already impulses, and there is no need for a 1/sinc compensation filter. This filtered signal simulates the processing of the modulator. It is stored in a file and used to form the input to the demultiplexer.

Figure 8.1 shows the SPW block diagram that was used to generate baseband data for the carriers. The block entitled "QPSK source" generates a random symbol pattern. The output of the block is a multiple of one of the following four complex numbers: (1,1), (1,-1), (-1,-1), or (-1,1). In the QPSK source block, the user can set a parameter for the signal power, and that determines the amplitude of the output signal. The block can be replaced by other blocks that generate more well-defined patterns, such as square waves or

constants. These were used to test various stages of the demultiplexer simulation while it was being constructed.

This four-valued signal is used for two purposes: it is stored for use with the demodulator emulation software and it is filtered to simulate a modulator's output.

Storing the Signal for Use with the Demodulator Emulation Software

One branch of the output of the QPSK source block is quantized and then written to a binary file. This is by no means the only way to perform this task. For instance, in retrospect it is all to clear that the quantizers (called midtread blocks) are not necessary. The whole point is to perform a mapping:

I	Q	mapping to binary
1	1	0x 0 1 0 1
1	-1	0x 0 1 f f
-1	-1	0x f f f f
-1	1	0x f f 0 1

This is the binary format used by the software that emulates the demodulator ASIC. The overall purpose is to create binary data for a carrier and store the original signal. The signal is then modulated, multiplexed, sent over a noisy channel, demultiplexed, and demodulated. The demodulator output for this carrier should match the original signal. They are compared and the number of errors is counted and watched closely.

Filtering to Simulate a Modulator's Output

The other function of the SPW block diagram shown in Figure 8.1 is to simulate the

modulator. The translation to radio frequencies is not simulated: only the modulating filter.

A complication arises because after the A/D converter in the receiver, there is not an integer number of samples per QPSK symbol. Care must be taken to make sure that the simulation of the output of the A/D converters is realistic.

To that end, the signal was oversampled, filtered, and then downsampled. The data-shaping filter also served as an interpolating filter. The final downsampled signal was at the right number of samples per symbol. In order to figure out how much to upsample and downsample, the number of samples per symbol in the INTELSAT carrier specifications was computed as a fraction, as shown in Table 8.1. To perform the filtering, an SPW block was used. It performs filtering by transforming its input to the frequency domain, multiplying by the frequency response for a raised cosine shape, and transforming the data back to time domain. One of the parameters it requires is a number of taps for the filter. This must be a power of two, because the size of the transform is a power of two. In order to make the filter serve as a suitable interpolating filter for the upsampled input, the number of taps in the filter was specified to be the power of two that was the ceiling of $(10 \cdot \text{upsampling term})$. Thus 10 symbol lengths were included in the filter, which is long enough to be considered ideal.

Since the symbols were upsampled, they were impulses and not NRZ square pulses. Therefore it was not necessary to use a 1/sinc filter. A square root filter was used since that is what is performed by the modulator.

This method of upsampling, filtering, and downsampling is highly inefficient and can be made much more efficient by writing code to simulate the filter. The code could use two tricks. First, most of the samples in the upsampled signal are 0 and the filter does not need to consider those in a multiply-and-add convolution operation. Second, most of the

outputs are thrown away and do not need to be computed in the first place. These ideas were not considered when the system to generate baseband data was constructed. The system was made to work, and runs were performed overnight. Once the data was generated, it was saved and used for many different simulations of the demultiplexer. If it is desired to generate more simulations of baseband data, for instance for all the carriers on the spreadsheet, then it would be wise to replace the system of upsampling, filtering, and downsampling with efficient code.

8.2 INTELSAT Carrier Specifications

Table 8.1 shows the INTELSAT carrier specifications, including columns for generating input data.

Transmitted Symbol Rate

This column is copied over from Table 2.1.

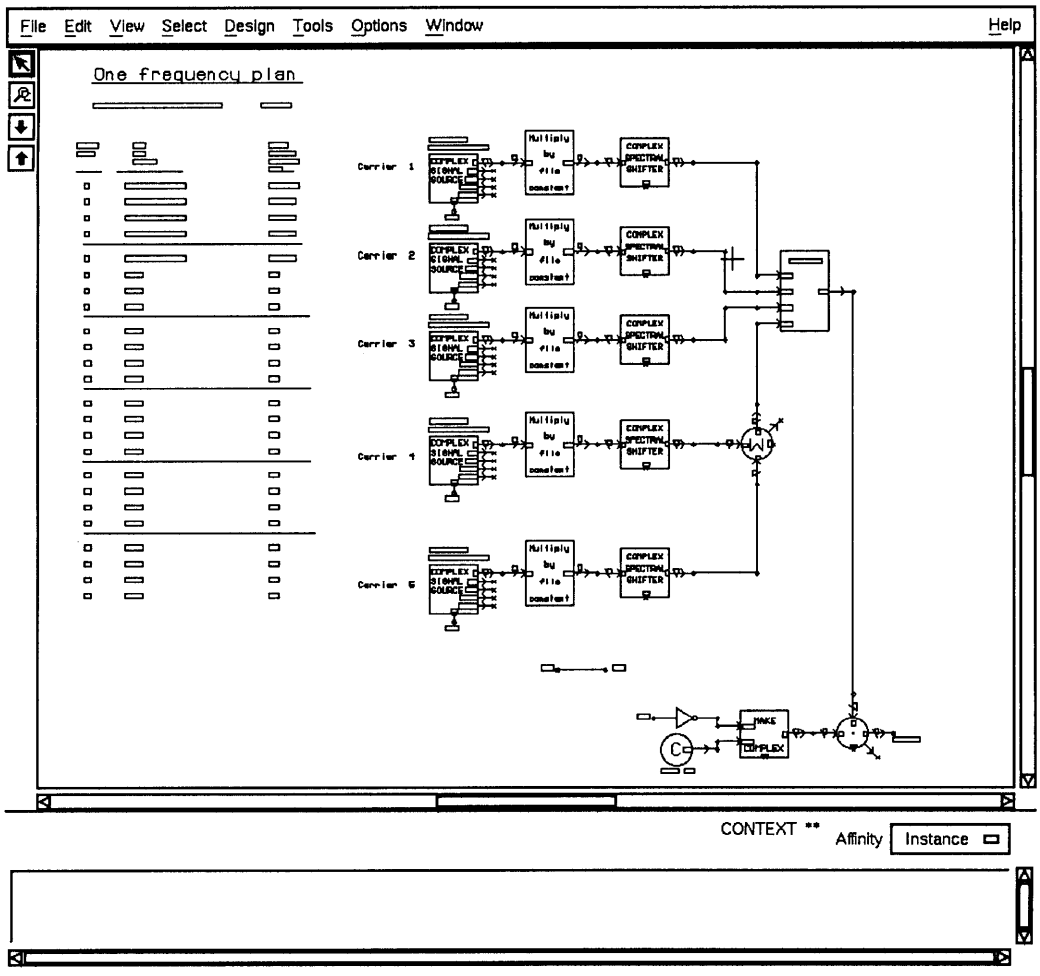
Samples per Symbol After the A/D Converter

These columns take into account the 11.52 MHz sampling frequency of the A/D in the analog front end. They present the data in fractional form, which corresponds to the upsampling and downsampling ratios to be used in generating baseband data for each carrier.

Table 8.1: Carrier Specifications in a Form for Simulating Input

Info Rate (kbits/s)	Trans. Symbol Rate (ksymb/s)	Samples per Symbol after A-to-D in Fractional Form		
		Numerator	/	Denominator
384	307.2	75	/	2
768	546.1	675	/	32
1544	1,093.3	432	/	41
2048	2,144.0	360	/	67

Figure 8.2: Modulating and Multiplexing the Inputs



Baseband data from many different carriers can be modulated and multiplexed. The chart at the left contains the information about the frequency plan. The actual numbers cannot be read at this viewing scale and are represented by rectangles. On the right, in the block diagram, there is a bank of source blocks which read the baseband data from files. Then there are blocks to multiply the data by a file constant, in order to alter its power. Next, there are complex spectral shift blocks to modulate the data. Finally, the modulated data is added together to form the input to the next stage, which is the channel model where noise is added.

8.3 Modulating and Multiplexing

8.3.1 Blocks Involved in Modulating and Multiplexing

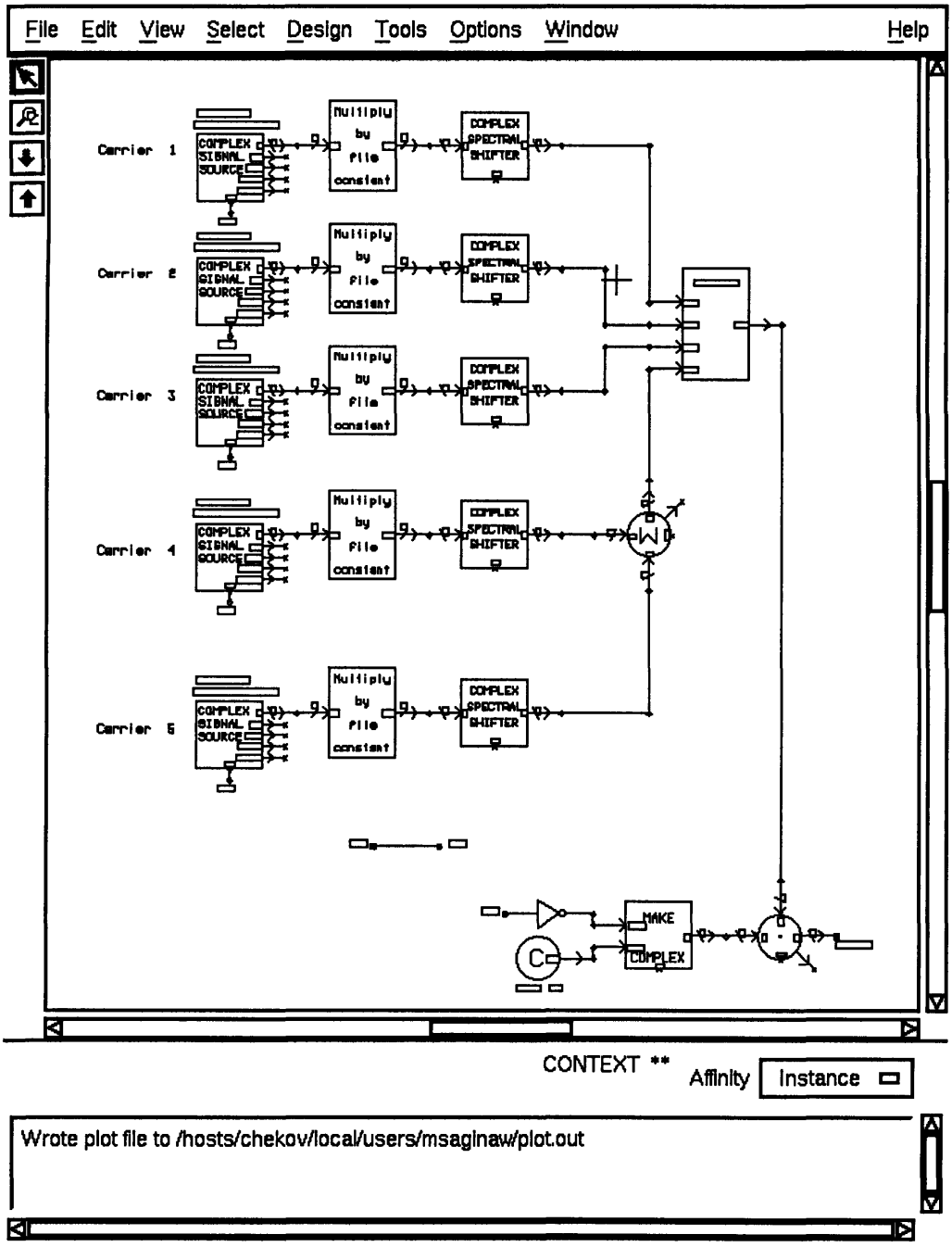
The simulated carriers are generated at baseband and stored in files. In another SPW block diagram, Figure 8.2, which is part of the simulation of the demultiplexer itself, the baseband carriers are modulated to various frequencies and added together (multiplexed).

The chart at the left of Figure 8.2 has 24 rows. This is where the user can enter parameters for up to 24 carriers. For instance, in the tests run for this thesis, up to five carriers were multiplexed. Thus the first five lines of the chart are used to specify information about carriers, and there are five sets of three blocks at the right.

The two parameters for each row are the name of the file containing baseband data and the center frequency to which that data is to be modulated.

Figure 8.3 shows the three blocks for each carrier. On the left is a signal source block. The parameter chart dictates the name of the file from which this source reads. When the hold signal of this block is active, the block does not read new data from the file. Instead, it holds the last value.

Figure 8.3: Larger Diagram of Modulating and Multiplexing Blocks



In this larger view of the block diagram elements from Figure 8.2, it is easier to see the source blocks, the blocks for scaling by a constant, the complex spectral shifters, and the adders that make up the modulation and multiplexing system.

The hold signal is arranged so that the source block reads a new value every four cycles. The clock of the entire simulation is 46.08 MHz but the clock of the A/D converter is only 11.52 MHz, so one new value comes from the A/D converter every four ticks of the 46.08 MHz clock. This allowed smaller files to be generated. It would have been possible to generate files with the same data repeated four times, and that kind of data could have been read without the need for hold signals. However, that kind of file would have taken four times as much memory as the files used. The files used already took of the order of 20 megabytes each, which was a lot given the other memory required by the simulation files and the available memory on hard drives. Repeating each entry of the signal four times would have been a significant waste.

The block in the middle is for multiplying the data by a constant. That constant is stored in another file and controls the amplitude and power of the signal. The block entitled "file constant" is where the SPW simulation reads an external file to get a coefficient for multiplying the noise. This arrangement allows the coefficient in the external file to be changed without the need for recompiling the SPW simulation, which takes 14 minutes. Other simulation software such as Ptolemy offers an easy way to dynamically link the changed blocks to the rest of the simulation, but that possibility in SPW was not readily available to the user, and any hacking to get it to work was not explored. Instead, whenever information needed to be changed frequently, that information was relegated to external files.

At the right is the complex spectral shifter. It also has parameters that depend on the chart of parameters. The block multiplies its input by a digital complex exponential, so the output is shifted in frequency (the signal is convolved with an impulse in the frequency domain). The complex spectral shift block also required another parameter: a sampling

frequency. That was set to 46.08 MHz because the entire simulation is running at that frequency.

8.3.2 Gating the Output by the F_s Clock

The outputs of the complex spectral shifters were added together and then gated by the F_s clock, as is shown near the bottom right of Figure 8.2 and Figure 8.3. The gating was done by multiplying the signal by the complex number $0+0i$ when F_s was inactive, and by $1+0i$ when F_s was active.

The gating was done because otherwise, the multiplexed input would have had a sinc window on its spectrum. By gating, the multiplexed input is instead made into the upsampled version of the input. This introduces spectral images, but they are removed by the anti-aliasing filter. The fact that there are images can be seen by comparing two signals: one used as a reference and the other an upsampled version of the reference signal.

Derivation of Equation for Spectral Images Due to Upsampling

Consider a discrete time signal $x[n]$ which has a discrete time Fourier Transform (DTFT) denoted by $X(\omega)$. The idea is that if $y[n]$ is an upsampled version of $x[n]$, then $Y(\omega)$ is closely related to $X(\omega)$. The steps are to define upsampling precisely, to write an equation for the DTFT of the upsampled variable, to rewrite that DTFT by substituting in quantities related to the original signal, and lastly, to write the relationship between $X(\omega)$ and $Y(\omega)$.

First, a precise definition of upsampling is in order. If $y[n]$ is defined as the result of upsampling $x[n]$ by M, then the values for $y[n]$ are as follows:

$$y[n] = \begin{cases} x\left[\frac{n}{M}\right] & n \text{ is a multiple of } M \\ 0 & \text{otherwise} \end{cases}$$

The DTFT of the upsampled signal is as follows:

$$Y(\omega) = \sum_{n=-\infty}^{\infty} y[n]e^{-j\omega n}$$

This equation can be rewritten because $y[n]$ is zero except when n is a multiple of M. It is helpful to define a new variable, p, such that $n = Mp$. Then the equation for $Y(\omega)$ can be rewritten as follows:

$$Y(\omega) = \sum_{p=-\infty}^{\infty} y(Mp)e^{-j(\omega M)p}$$

Even though the sum does not include all the values in the signal $y[n]$, it does include all the values of $y[n]$ that are nonzero, so it is still a valid expression for $Y(\omega)$. With that sticky but important detail handled carefully, the rest of the work can proceed.

Using the fact that $y[Mp] = x[p]$, the equation for $Y(\omega)$ can be rewritten as follows:

$$Y(\omega) = \sum_{p=-\infty}^{\infty} x[p]e^{-j(\omega M)p}$$

As expected, this is very closely related to the DTFT of $x[n]$: $Y(\omega) = X(\omega M)$.

Note that the energy in $x[n]$ is the same as the $y[n]$, and similarly, the energy in $X(\omega)$ is the same as the energy in $Y(\omega)$. This can be seen from the figure by realizing that the images are all only $1/M$ as wide as the images in $X(\omega)$, but there are M times as many of them.

8.4 Model of the Channel and Analog Front End

The modulated, multiplexed data was combined and put together in an SPW block called "Frequency Plan," which appears in Figure 8.4. Noise is added to it, and the combined signal is passed first through anti-aliasing filter and then through A/D converters, which provide the simulated input to the demultiplexer.

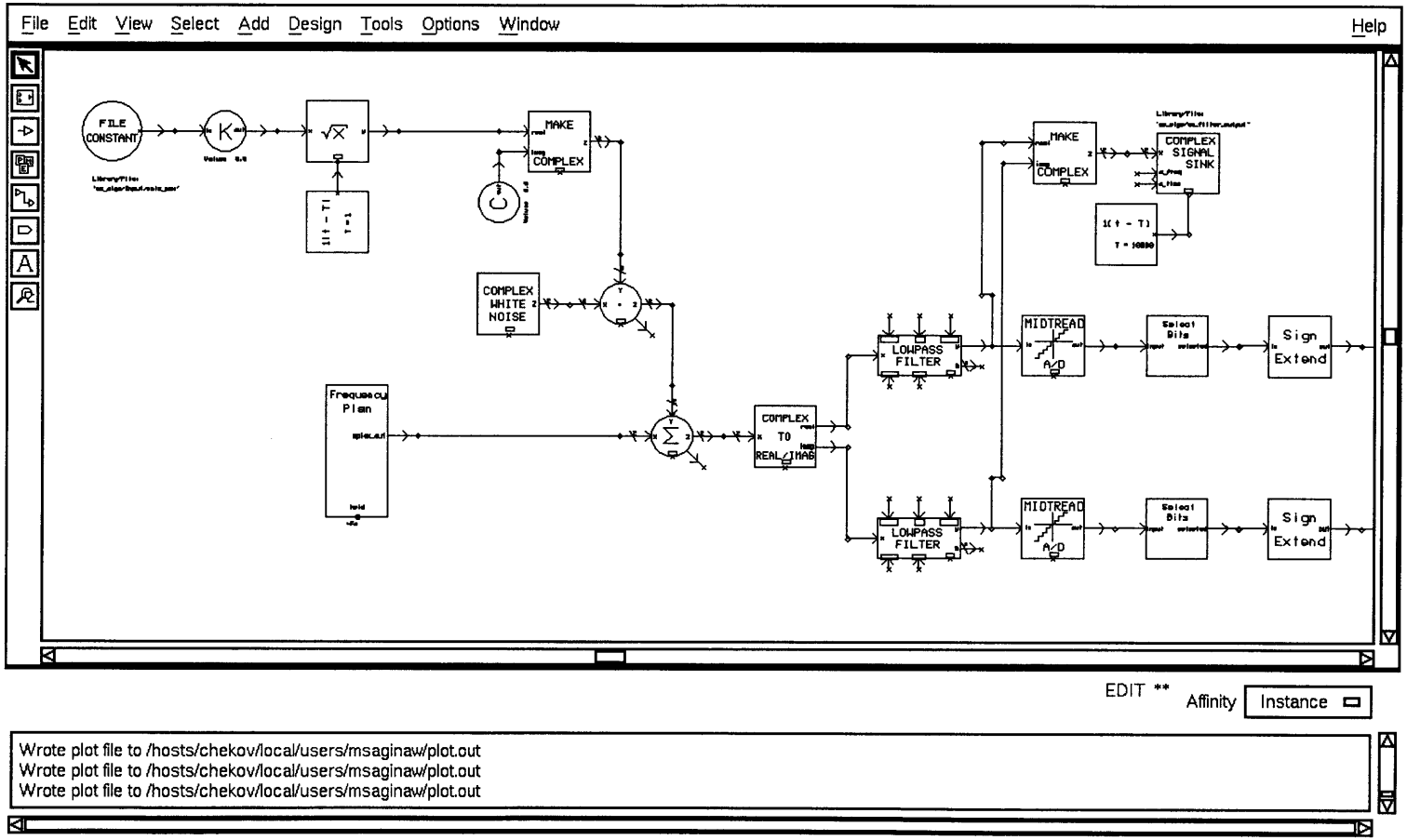
8.4.1 Generation of Noise

In Figure 8.4, the basic block that generates noise is the block entitled "Complex White Noise." However, there is a lot of other machinery associated with it. The function of those other blocks is to adjust the power of the noise being added to the signal.

The block entitled "file constant" is where the SPW simulation reads an external file to get a coefficient for multiplying the noise. This arrangement allows the coefficient in the external file to be changed without the need for recompiling the SPW simulation. It is very similar to the block in the modulating and multiplexing which multiplies the signals by a file constant in order to adjust the signal power.

There is a multiplication by a constant done by a circular block entitled "k." That fixed constant is 0.5. Then there is a square root function. The purpose of these is to make the data entry for the noise power intuitive. The total output power of the block entitled "Complex White Noise" is 2 units of energy per second: 1 unit in the real domain and 1 unit in the imaginary domain. The desired power is divided by 2 and then the square root is taken.

Figure 8.4: Model of Channel and Analog Front End



The modulated and multiplexed signal comes from the box called "Frequency Plan." White Gaussian noise is added to simulate the channel. The signal is passed through anti-aliasing filters and sent to A/D converters.

In an external file, a noise power N is written. The coefficient to the noise block output is $\sqrt{N/2}$. The noise from the noise block has power 2. Thus the variance of the product is $(N/2)2$, or N. Thus the user enters a number which corresponds to the noise power. The manipulations via SPW blocks are in place for the purpose of simplifying the input. That white Gaussian noise, with power controlled by a coefficient from an external file, is added to the signal. The carrier powers are also controlled by coefficients from an external file.

8.4.2 Anti-aliasing Filters

The anti-aliasing filters have the job of retaining signals in the 9 MHz band from -4.5 MHz to 4.5 MHz, and cutting off signals whose frequencies are less than -5.76 MHz or greater than 5.76 MHz. In software, the action of the filters was simulated by using digital filters. A built-in SPW block was chosen. It was a digital FIR filter with 55 taps. The large number of taps made possible a sharp amplitude response. The use of a symmetric FIR filter made possible a filter with a flat group delay. The extra images of the upsampled input signal were cut off by the filter, and only the central image remained.

8.4.3 A/D Converter

The A/D converters were simulated by SPW blocks which are quantizers. The entire simulation is already working with digital signals. The simulation of the A/D converters do not sample a continuous time signal and convert the signal to a discrete time one; rather, they quantize the signal. Previous to the A/D converters in the simulation, the signals are all double precision floating point numbers. Afterwards, they are quantized, in this case to 8 bits. That parameter of quantization is easy to change, thus offering the opportunity to try out many different levels.

The A/D converters also have a setting for the maximum amplitude of their inputs, and they simulate clipping at saturation and cut off at low levels.

In the demultiplexer simulation, the A/D converters have an input amplitude maximum of 4.0 and 8 bits of quantization are used. Thus if the input is less than -4.0, then the output is -4.0 (0x80 in two's complement binary). If the input is more than 3.96875, then the output is 3.96875 (0x7f in two's complement binary). If the input magnitude is less than 0.00390625, then the output is 0.0 (0x00 in two's complement binary).

To avoid saturation or cutoff, a 2-bit backoff from the A/D converter's maximum was used. Thus the signal power and noise power are adjusted by the coefficients so that the total amplitude is about 1.0. In the real hardware system, the power of the signal and noise will be fixed but the gains of the electronics will be controllable, as will be the setting for the maximum amplitude accepted by the A/D.

8.4.4 Dithering

In the context of A/D converters, dithering refers to the phenomenon where small signals are not cut off even though they are below the A/D threshold, because they are combined with larger signals. For instance, if the A/D threshold is 0.00390625, a signal from a narrow band carrier might have a root mean squared amplitude of only 0.001 and would be cut off by the A/D converter. However, if the signal were combined with a second carrier that had a root mean square amplitude of 1.0, then the sum would pass the threshold. After the demultiplexer performs the FFT and separates and filters the carriers, the narrow band signal can be converted back to the time domain in the IFFT.

8.4.5 Calculating the Signal-to-Noise Ratio

The quantity E_b/N_o had to be computed to make BER plots, and a method for doing so is

given in [Feher]. For simplicity, suppose there was just one carrier. The noise was turned off and the carrier passed through the anti-aliasing filter. The power of the carrier, C, was measured after the anti-aliasing filter. Since the carriers contain unwanted images in their spectra due to being upsampled, it was necessary to look at the signals after the anti-aliasing filters took out the images. Note that $E_b = CT_b = C/f_b$.

Let N be the noise power that passes through the filter, which was measured by turning off all the carriers and passing the noise through the filter. Then $N_o = N/BW$, where BW is the noise equivalent bandwidth of the filter, which was measured in three different ways and know to be 9,635,668 Hz. The best measurement came from passing an impulse through the filter (since it is a digital filter in the simulation), computing the energy in the response, and using Parseval's theorem to equate that to the noise equivalent bandwidth. Finally, the equations can be divided to obtain: $\frac{E_b}{N_o} = \frac{C}{N} \cdot \frac{BW}{f_b}$.

Chapter 9

Chosen Architecture

In this chapter, the structure of the demultiplexer hardware is described, and block diagrams from the SPW simulation are discussed.

9.1 Clocks

The hardware has three different clocks: one at 46.08 MHz, one at 23.04 MHz, and one at 11.52 MHz. These clock rates were chosen based on three constraints: the requirements of the demodulator, ease of implementation of frequency filters, and hardware speed.

Requirements of the Demodulator

The 23.04 MHz frequency is the frequency at which output data samples are clocked into the demodulator ASIC. That is appropriate since the demodulator can handle clocks up to 25 MHz.

Ease of Implementation of Frequency Filters

Given a clock speed for the demodulator of a little less than 25 MHz, what is the best sampling rate for the A/D converters? The answer is intimately tied up with the frequency filters. They can be simplified by a wise choice of sampling frequency.

If the FFT samples are not spaced properly, then they will not span the allocated bandwidth slots snugly. For each frequency plan, it would be necessary to compute the filter coefficients, based on where the FFT samples fell. Those coefficients would have to

be downloaded to one piece of hardware, while an identical piece of hardware was read to obtain the coefficients of the frequency plan being used.

This can be simplified if the FFT samples span the allocated bandwidth slots just right. In that case, there is only the need for one memory unit to hold the coefficients for all types of carriers. In fact, there is another advantage too. If an FFT sample falls on the center frequency of the carrier, and if the filter coefficients are all real, then the filter coefficients (which are discretely spaced in frequency) are symmetric. They only need to be computed and stored in hardware once. They can be called twice, and this saves on the memory needed.

The bandwidths used in the INTELSAT system must be analyzed in order to figure out how to space the FFT frequency samples just right. In the INTELSAT system, allocated bandwidths are always multiples of 22.5 kHz and center frequencies are always multiples of 11.25 kHz. This is tied in with the spacing of FFT samples in the frequency domain. The samples are spaced one every $F_s/4096$ Hz. So if the center frequencies need to fall on the sample points, then the following equation must hold: $11,250 = k \cdot F_s/4096$, where k is an integer. If k is 4, then F_s is 11.52 MHz and $2F_s$ is 23.04 MHz. This is a good match to the demodulator. Given this choice of clock speed, the simplest way to divide up the 36 MHz band of frequency is to divide it into 4 slots of 9 MHz. Each demultiplexer, with an A/D sampling rate of 11.52 MHz, can handle a 9 MHz sub-band.

Hardware Speed

The 11.52 MHz sampling rate means that the fastest clock, for the IFFT, runs at 46.08 MHz. It is necessary to make sure that the hardware can run at that speed. Actually, this high speed has many implications for the hardware design and makes it difficult.

First, it is necessary to make sure that the FFT DSP chips can run at 46.08 MHz. Most commercial FFT chips can not run that fast. One special line of chips from BDSP runs at 50 MHz. It comes in a special package called a super ball grid array (SBGA), presumably because this package reduces lead inductance substantially and is therefore better for high speed operation. At this time, the SBGA package is new in the industry and very few people have experience with it. Because the pins on the SBGA package are not available on the sides of the chip, it is necessary to get special adapters for the chip for testing and debugging.

Furthermore, by the time all the path latencies are taken into account, it is difficult to meet the setup and hold times of the chips. For this reason, some parts which were planned as PROMs will actually be implemented as RAMs, which run faster. These will be loaded once with data and afterwards used as though they were PROMs.

Another step necessary to meet the hardware timing requirements was to add extra registers to break up some of signal flow paths between unclocked components. These were not simulated, so there will be small differences in the exact timing between the simulation and the final hardware. It is anticipated that these will not be a problem, however, because the timing differences can be easily done away with by tweaking the Altera programmable logic device (PLD) clocks by a few cycles once the actual hardware is being tested.

9.2 BDSP 9124 FFT Chips

The six different function codes of the BDSP 9124 chip which are being used in the demultiplexer, along with their latencies, are listed in Table 9.1.

Table 9.1: BDSP 9124 Function Code Latencies

Function Code	Latency (cycles)
BFLY2	18
BFLY4	18
BFLY16	68
BWND2	20
BWND4	20
MOVD	18

The BDSP 9124 has 24 bits of input each for real and imaginary data, and real and imaginary coefficients, all using two's complement notation. It also has a StartStop pin, an enable A pin, and an enable B pin.

Suppose a BDSP 9124 is to be used for a 32 point transform. Then the StartStop signal needs to go from inactive to active when the first data and coefficient values arrive. The StartStop signal remains active as the data and coefficients are clocked in. It needs to go inactive when the last data and coefficient values arrive. The enable A signal causes the BDSP 9124 to read the function code signal and needs to go active two cycles before the data and coefficients arrive. The enable B signal causes the BDSP 9124 to read the two scale factor input signals and needs to go high at least one cycle before the data and coefficients arrive.

9.3 Overlap Buffer

The 50% overlap buffer is between the A/D converters and the FFT operation. It is implemented as a circular buffer. The buffer needs to have at least 8192 locations. Let the

memory be divided into groups of 2048 locations labelled 'A,' 'B,' 'C,' and 'D.'

To illustrate how the circular buffer operates, imagine that 'A' and 'B' are full of data. As one sample is read into 'C,' two samples are read out of 'A.' After 4096 cycles of operation, 'C' is filled just as the last sample of 'B' is being read out. At that point, the writing begins in 'D' and the reading begins from the beginning of 'B.' The samples from 'B' are overlapped. This procedure continues, as shown in Table 9.2.

Table 9.2: Operation of 50% Overlap Buffer

Read	Write
none	A
none	B
A, B	C
B, C	D
C, D	A
D, A	B
A, B	C
etc.	etc.

This is the only stage of the demultiplexer where the RAM write addresses are sequential and the read addresses come from a pattern in a PROM. In all other stages, it is the other way around: the write addresses are according to a pattern and the read addresses are sequential. This is because in the early stages of the hardware design, the idea was to use RAMs with addressable write and sequential read, where those RAMs had an internal counter and no pins for a read address, and are therefore smaller. At a later stage in the hardware design, this idea was dropped due to the need to get any RAM that was fast enough and had small enough access time to meet the hardware timing constraints.

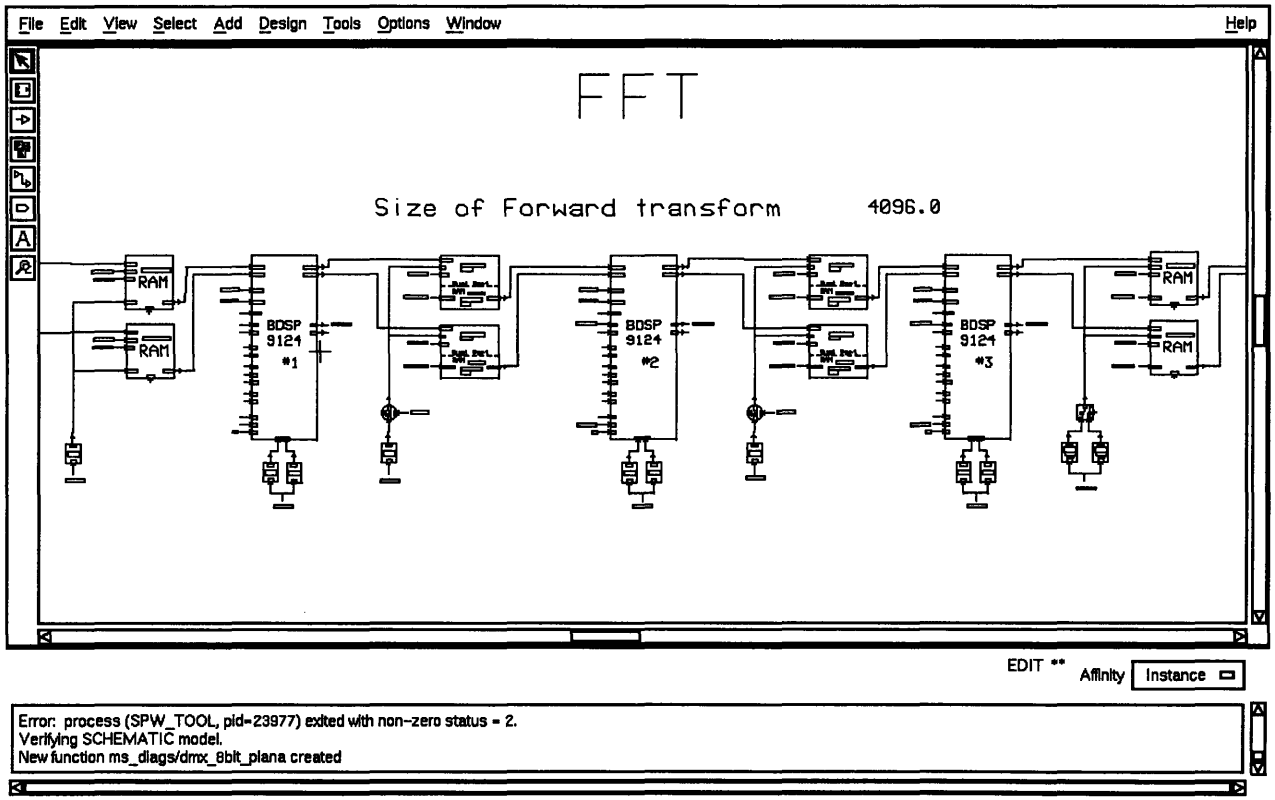
However, the hardware schematic wiring for variable write addresses and sequential read addresses was already underway, as was the software development effort. Therefore, the software development continued based on RAMs with addressable writing and sequential reading, even though this made the software somewhat more complicated.

As mentioned, this 50% overlap buffer is the one exception, and the RAMs have a sequential write address and a read address according to a pattern. It is necessary to have a read address according to a pattern because of the overlapping way in which the input data is used. That overlapping effect must come from read addresses, and the write address may as well be sequential. In addition, the read addresses also are arranged for digit reversal and for reshuffling, so they are different for the first use of a set of 2048 points than they are for the second use. This different arrangement could not be achieved by a write address according to a pattern coupled with a sequential read, so the read address had to be according to a pattern.

9.4 FFT

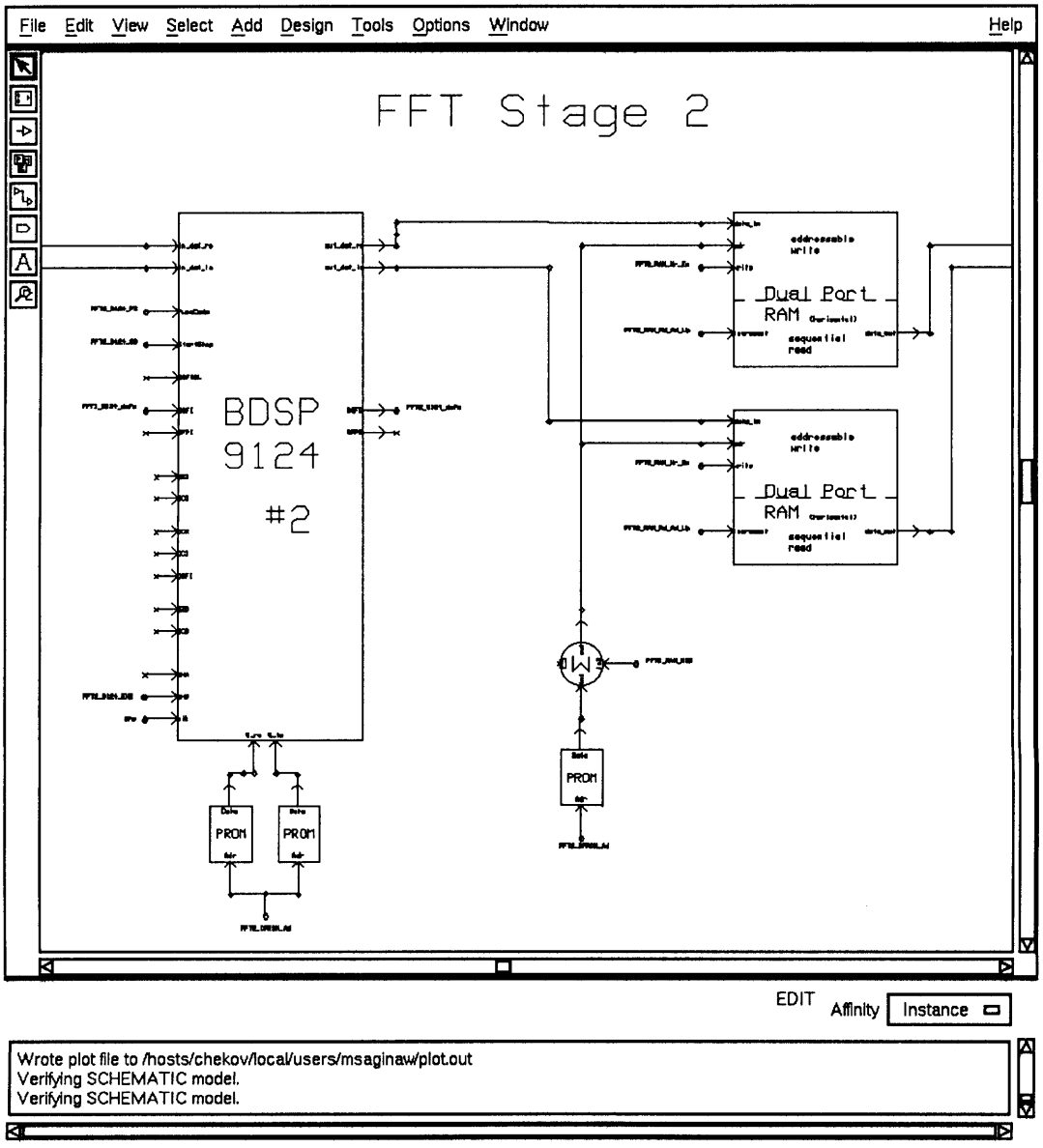
After the 50% overlap buffer, the next major section of the demultiplexer is the FFT. This section uses three stages with BDSP 9124 chips to take a 4096 point DFT of its input. The transformed data is then filtered at the first stage of the IFFT.

Figure 9.1: Simulation of the FFT



The 50% overlap buffer is implemented via the read addresses of the RAMs at the input. The data is passed through three radix 16 stages and the 4096 point FFT is computed. The patterns for digit-reversal, twiddle coefficients, and reshuffling addresses are independent of the particular frequency plan.

Figure 9.2: Stage 2 of the FFT



Data enters the BDSP chip figure at the left and twiddle coefficients enter at the bottom. The outputs are fed into RAMs. Since the patterns for twiddle coefficients and data reshuffling are independent of the frequency plan, they may be stored in PROMs.

9.4.1 The BDSP 9124 Chip in the FFT

Each stage of the FFT has a BDSP 9124 chip and PROMs that feed in twiddle coefficients.

These memory units can be PROMs because the FFT twiddles do not depend on the frequency plan. They are fixed, and the contents of the memory do not need to change for different frequency plans. Also, the FFT clock only runs at $2F_s$, which is slow enough to allow use of the PROMs, even though they have slower access times than the RAMs that are used in the IFFT. A sequential counter, to come from an Altera PLD in the actual hardware, feeds a sequential address to the PROMs.

In the FFT, all the signals to the BDSP 9124 chips are independent of the frequency plan. Each chip runs the BFLY16 function code all the time, so these signals can be hard-wired. The StartStop, enable A, and enable B signals are all periodic and occur once every 4096 of the $2F_s$ cycles. These can come from Altera PLDs and do not need to come from the host PC.

9.4.2 Preparing Data for the Next Stage

After the BDSP 9124 chips, the data enters memory RAMs: one for the real data and one for the imaginary data. The write address is determined by a PROM with 8192 points. As was the case with the twiddle coefficient address patterns, the data reshuffling address patterns are independent of the frequency plan. This fact, coupled with the fact that the FFT clock is only running at $2F_s$ means that the addresses can come from PROMs.

The 8192 addresses involved in data reshuffling work in ping-pong fashion. The first 4096 addresses all are addresses for the lower half of the data RAMs. The next 4096 addresses are for the upper 4096 points of the data RAMs. Just as was the case with the PROMs with twiddle coefficients, the PROM with data reshuffling addresses gets its address input from a counter that will come from an Altera PLD.

9.5 IFFT

9.5.1 Using the FFT to Compute IFFTs

In order to perform an IFFT operation, the same hardware and algorithms can be used as were used for the FFT, since the formulas are so similar. One way to do an IFFT is to change the twiddle coefficients so that the sign of the imaginary part is positive. Another way is to use the same twiddle coefficients as the FFT and also to use a trick. Take the complex conjugate of the input data, take the FFT, and then take the complex conjugate of the output. For these kinds of manipulations, the BDSP 9124 has one control pin for taking the complex conjugate of the input, and another control pin for taking the complex conjugate of the output.

The mathematics behind why this trick works involve a few manipulations of complex conjugates. Since the input signal is conjugated, the FFT twiddle coefficients are used, and the output signal is conjugated, the transform operation can be written as follows:

$$\left(\sum_{k=1}^{N}(X[k])^*\left(e^{\frac{-j2\pi kn}{N}}\right)\right)^*$$

The complex conjugate of the sum equals the sum of the complex conjugates:

$$\sum_{k=1}^{N}\left((X[k])^*\left(e^{\frac{-j2\pi kn}{N}}\right)\right)^*$$

The complex conjugate of each product equals the product of the each complex conjugate:

$$\sum_{k=1}^{N}X[k]e^{\frac{j2\pi kn}{N}}$$

This is indeed the inverse DFT of $X[k]$, so the method of computing it by using the FFT with conjugation manipulations is justified.

The simulation of the IFFT involves four stages. The data shaping filters are applied in the first stage, where the function codes BWND2 and BWND4 are used to multiply the incoming data by the appropriate filter coefficients. The subsequent stages of the IFFT transform the filtered data back to time coordinates, one carrier at a time. Finally, at the output, the addressing is arranged to provide a 50% discard of the data.

In each stage of the IFFT, there is a BDSP 9124 chip with coefficients, and data RAMs with reshuffling address patterns.

9.5.2 Filter Coefficients in First Stage of IFFT

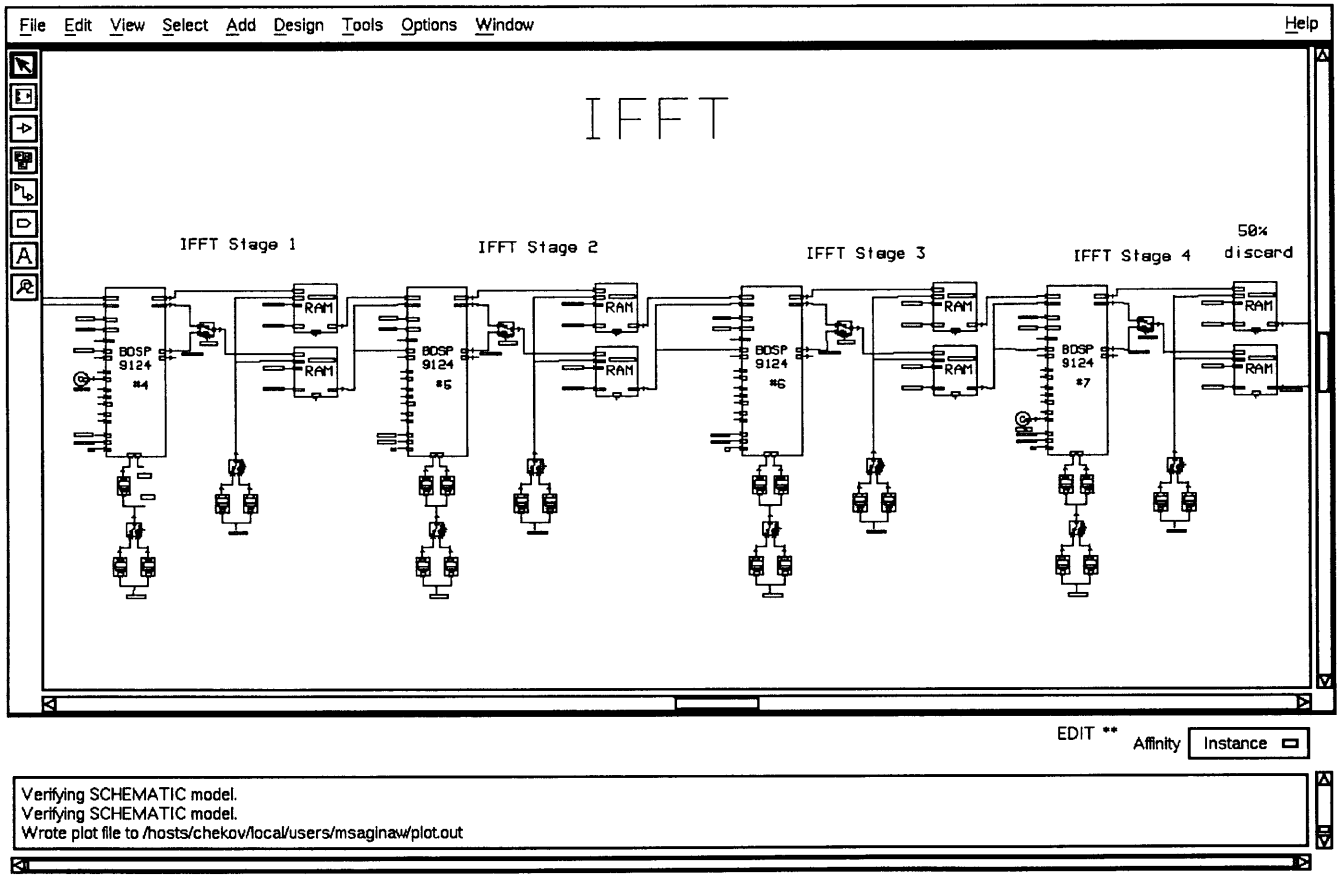
In the first stage, the coefficients to the BDSP 9124 are not twiddle coefficients but rather filter coefficients. In the simulation, there is only a set of real coefficients below the BDSP 9124 chip, since the imaginary coefficients are all 0. The memory holding the coefficients has all the coefficients for all the filters for all the INTELSAT carriers handled by this demultiplexer.

The subsequent stages are all similar to each other. They have the BDSP 9124 chip with twiddle coefficients. Then the data passes into data RAMs.

9.5.3 The BDSP 9124 Chip in Each Stage of the IFFT

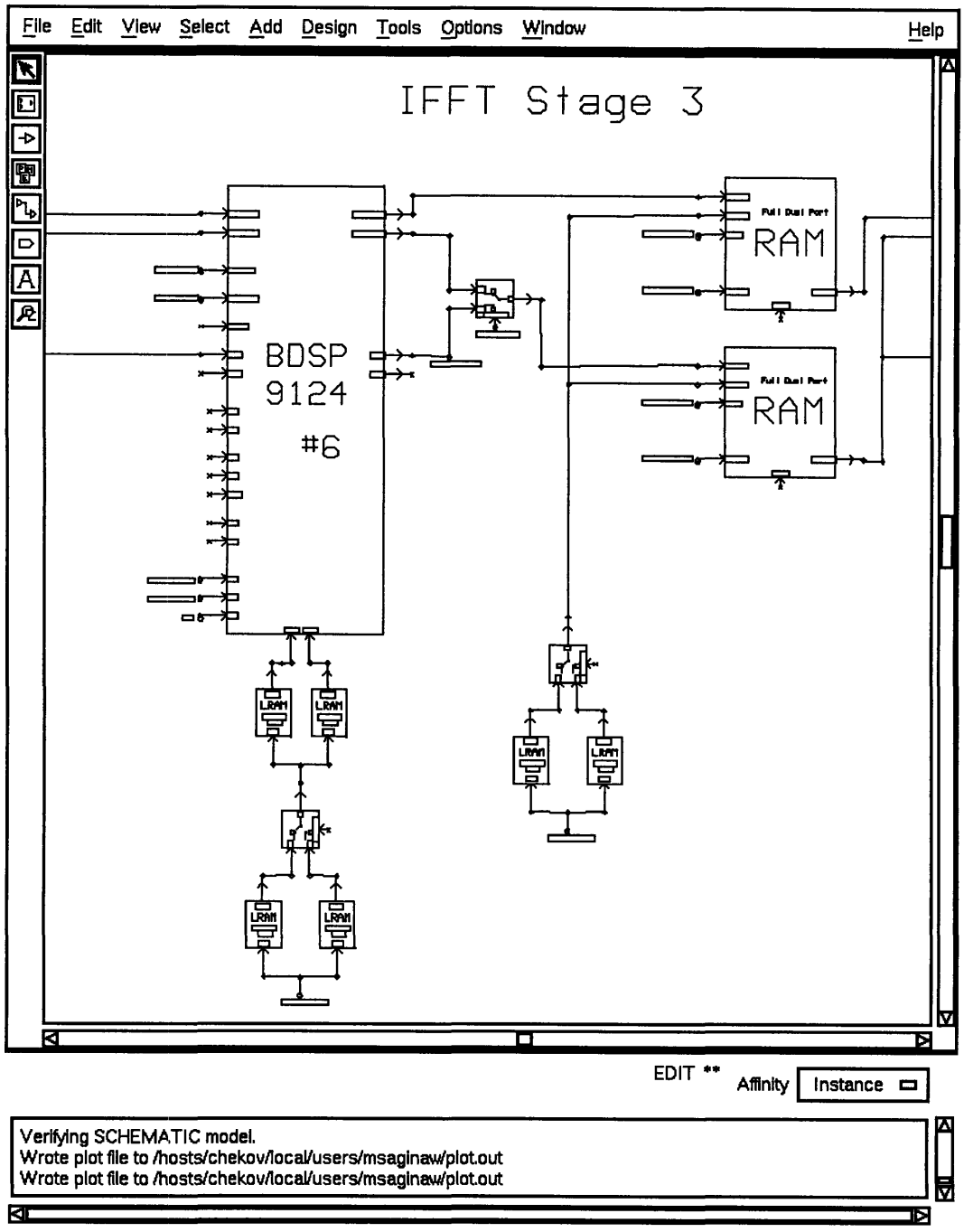
In each stage of the IFFT, there is a BDSP 9124 chip along with apparatus to provide it with coefficients. For stages 2, 3, and 4, these coefficients are twiddle factors for the transform operation. The memories below the BDSP 9124 chip contain the 2048 twiddle coefficients in order.

Figure 9.3: Simulation of the IFFT



The data shaping filter is implemented by multiplication in the first stage of the IFFT. The four stages put together perform the IFFT, and the addressing at the output discards the 50% of the samples that are invalid.

Figure 9.4: Stage 3 of the IFFT



In the FFT, the patterns for twiddle coefficients and data reshuffling were independent of the frequency plan but in the IFFT they are not. The twiddles operate in a two tier structure. First, all the 2048 complex coefficients that could possibly be needed are stored in one set of memories, shown under the BDSP 9124 chip. Underneath of those memories, other memories have the addresses for the twiddle coefficients for the particular frequency plan. A switch is in place for changing between two frequency plans on-the-fly. There are also two RAMs with different patterns for data reshuffling, which also depend on the frequency plan. Each RAM holds the patterns for one plan, and they can be switched in and out on-the-fly. Finally, there is one extra switch to funnel the BDSP 9124's DSFO output signal into the data RAM. However, this is no longer part of the hardware design because it introduced too much propagation delay.

In the FFT, there is just one set of memory with twiddle coefficients in the order needed for the transform, but in the IFFT, the twiddles needed depend on the size of the transform. Therefore the structure of memories is different: it is a two-tier structure. The first stage is the memories with the lookup tables of trigonometric functions. The second stage is the RAM that is programmed by the host PC. It contains the patterns describing which twiddles are needed in the IFFT.

Furthermore, there are two of these RAMs with patterns of twiddle coefficients. One is for the frequency plan being used by the hardware, the one which is "at bat." There is also another RAM with patterns for another frequency plan, and it can be switched in when a change in the frequency plan occurs. This second RAM holds the twiddle coefficients for the frequency plan that is "on deck." Each of these two RAMs receives a sequential address from a counter, which in hardware will be provided by an Altera PLD.

9.5.4 Data Storage and Reshuffling in Each Stage of the IFFT

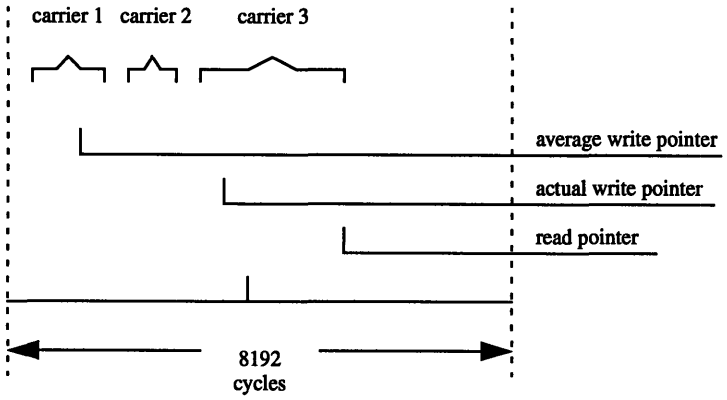
The output of the BDSP 9124 chip feeds into data RAMs for the real and imaginary data. The write address for these RAMs is controlled by patterns from a RAM that is programmed by the host PC. These reshuffling patterns must match the frequency plan. As with the twiddle coefficient addresses, there are two RAMs with data reshuffling coefficients. Only one controls the actual write address, and this is dictated by a switch. The control signal for the switch comes from the programmable logic which generates control signals for the hardware.

Now the flow of one particular frame of data during 8192 cycles is described, using stage 3 of the IFFT as an example. At first, data for all the carriers is stored in the memory that precedes the BDSP 9124. One carrier at a time, the data is passed through the BDSP

9124 chip and collected into the memory which follows. After the 8192 cycles, data for all the carriers for that frame are stored in the memory following the BDSP 9124.

In the FFT, it was clear that a memory size of 8192 points between each BDSP chip was adequate. The memory was used in ping-pong fashion. In the IFFT, this is less apparent but still true: the number of storage locations needed for each memory between the BDSP 9124 chips is 8192.

Figure 9.5: Data RAM Memory Requirements in the IFFT



Each bracket represents the IFFT samples associated with a carrier. Even though there are 8192 IFFT samples, there is no need to do ping-pong addressing and only 8192 memory locations are needed in the data RAMs in the IFFT. Even in a worst case scenario, the actual write pointer would never be less than the read pointer

There are three pointers to think about, as shown in Figure 9.5: the average write pointer, the actual write pointer, and the read pointer. The average write pointer starts at 0 and proceeds sequentially to 8191, and then rolls over to 0. The actual write pointer does not proceed so smoothly. It bounces forward and backward around the average write pointer, because it reshuffles the data from each carrier to a different order. However, all the data from one carrier remains in a clump with the other data from the same carrier.

Finally, there is the read pointer. It starts 4096 clock cycles after the write pointer. It lags the average write pointer by 4096. The key question to ask is whether the actual write pointer will ever be less than the read pointer. In fact, it will not even come close, because

the largest IFFT size is 2048. The largest jump backwards that the actual write pointer could take, relative to the average write pointer, is a jump of 2048. However the read pointer lags the average write pointer by 4096, so there is no danger of the actual write pointer being less than the read pointer. To summarize, the read pointer will never try to read something which has not yet been properly written.

Similarly, after the writing of a batch of 8192 points is done, the average write pointer rolls over to 0 and proceeds forward again. It lags the read pointer by 4096. The actual write pointer jumps around the average write pointer, but only by 2048 points at most. Thus it will never be greater than the read pointer. To summarize, the actual write pointer will never jump ahead so far as to overwrite something that the read pointer has not read yet.

9.5.5 Timing Issues

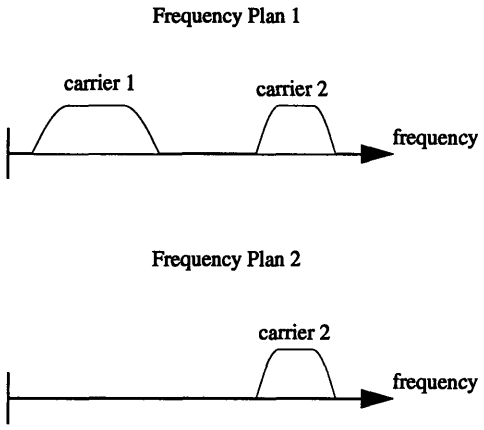
Writing to and Reading from the Final RAM

At the final RAM, data is clocked in at a frequency of $4F_s$. However, half of that is effectively discarded since it is written to an address that is never read. The read address to the RAM only increments at a rate of $2F_s$, and the signals are sent to the demodulator at that rate.

Switching on a Rollover Boundary

For each memory unit for twiddles and data reshuffling, the plan switches must occur as the address to the memory rolls over from 8191 to 0.

Figure 9.6: Timing of Frequency Plan Changes



The switch in frequency plan must come as each RAM -- with patterns of twiddle coefficient addresses and data reshuffling addresses -- rolls over from 8191 to 0. If it happened in the middle of the address sequence, the continued carrier would not be handled properly.

Suppose that carrier 2 in Figure 9.6 is supposed to be continued as the frequency plan changes. For the first plan, there is a group of addresses for carrier 1, and another group of addresses for carrier 2. For the second plan, there is only a group of addresses for carrier 2. Suppose that while frequency plan 1 was active and carrier 2 was being processed, there was an attempt to switch in plan 2. Then the addresses in plan 2 would be way beyond the clump of values for carrier 2. Frequency plan changes can only occur as the counters roll over from 8191 to 0.

The demodulator is able to accept on-the-fly switches also at this boundary, so the two hardware units are consistent.

9.5.6 Radix Structure

At each stage of the IFFT, the BDSP 9124 chips may be called upon to perform certain operations by giving them certain function codes.

Table 9.3: Operations and Function Codes at Each Stage

Stage 1		Stage 2		Stage 3		Stage 4	
Radix	Function Code	Radix	Function Code	Radix	Function Code	Radix	Function Code
2	BWND2	16	BFLY16	1	MOVD	1	MOVD
4	BWND4			4	BFLY4	4	BFLY4
				16	BFLY16		

These function codes are put together to perform each size of transform needed, from 32 to 2048, as shown (repeated from Table 6.2 for convenience) Table 9.4.

Table 9.4: Radix Combination for Achieving Each IFFT Transform Size

size	stage 1	stage 2	stage 3	stage 4
32	2	16	1	1
64	4	16	1	1
128	2	16	1	4
256	4	16	1	4
512	2	16	4	4
1024	4	16	4	4
2048	2	16	16	4

Because the filtering occurs in the first stage of the IFFT, the function codes there involve multiplication of the input data by a set of coefficients. Conveniently, the first stage of the IFFT has twiddles that are all 1, so there is no need to combine the coefficients with any non-trivial twiddle coefficients. (In the FFT, the first stage would have had twiddles that were all 1. However the BFLY16 function code was used, and that function

code combines two stages of BFLY4 operations into one stage, so the twiddle coefficients for the first pair of radix-4 stages were not all 1.)

9.6 Discarding Data

At the end of the IFFT, all the samples have been converted back to time coordinates. However, some of them are invalid due to aliasing, and must be discarded. This occurs at the end of stage 4 of the IFFT. The addressing is different. The valid samples are put in the first 4096 points of the memory and the invalid samples are put elsewhere. Only the first half of the memory is read. The read address is sequential but it only runs from 0 through 4095. The counter only advances at a frequency of 23.04 MHz.

9.7 Hardware Bits

9.7.1 Bit precision

Every stage of the project works with two's complement notation for positive and negative numbers. For instance, the A/D converters with 8 bits of precision have outputs that range from -128 to 127. (However, the control signals do not use two's complement.) The demultiplexer was planned with an A/D converter with 8 bits of precision for the real and imaginary parts. The BDSP 9124 chips can carry out transforms with 24 bits of precision. In the demultiplexer, only 16 bits of precision are used. This includes every stage of data in the FFT, the twiddles in the FFT, the handling of data in the RAMs, the transfer of data from the FFT to the IFFT, the filter coefficients, and the twiddles in the IFFT, all of which are precise to 16 bits.

At the output, however, only 8 bits are passed on to the demodulator. The ideal signal level for the demodulator is 32, for the height of a binary 1. The demodulator can handle a range of input signal levels. It accepts 8 bits of input, so only 8 bits of the demultiplexer's output are passed on to the demodulator.

The issue of which 8 bits to pass on to the demodulator is a thorny one. For each IFFT size, there is a path of radices of transforms. Each radix operation involves multiplication and addition of the input and makes the output generally bigger than the input. There is a barrel shifter at the output of the demultiplexer to pass on a selected 8 bits from the demultiplexer to the demodulator. If the wrong group of bits is selected, the demodulator will be saturated or will cut off. In either event, it will not demodulate properly. Since the host PC knows what kind of transform is being executed, it should be able to send the right control signal to the barrel shifter. Exactly which amounts of shifting for which carriers must be determined empirically through use of the simulation.

9.7.2 Control Bits

At each stage of the IFFT, there are two RAMs which receive their contents from the host PC. The primary purpose of the RAMs under the BDSP 9124 chips is to provide addresses for the twiddle coefficients. Similarly, the primary purpose of the RAMs under the data RAMs is to provide reshuffling patterns for the write address. However, these RAMs that are programmed by the host PC have 16 bits, and not all of those are needed for the primary purposes. The extra bits are needed as control bits, mostly for the BDSP 9124 chip.

Bit Allocation for IFFT Stage 1

The StartStop bit of the BDSP 9124 is either 0 or 1, so its value can come in

straightforward fashion from the RAM. However, the function code bit is encoded. There are only two options for the function code bit at this stage: BWND2 and BWND4. So all that is needed to indicate which function code is appropriate is a single bit. However, this bit must be decoded in an Altera PLD and then sent to the BDSP 9124 chip.

This stage has two unusual details: an unusual enable B signal and a schizophrenic bit that serves as a control signal for the previous stage.

Table 9.5: Bit Allocation for IFFT Stage 1

Coefficient RAM		Data RAM	
Signal	Bits	Signal	Bits
select from 5048 filter coeffs	13	write address for data reshuffling; 8192 locations in RAM	13
schizophrenic bit: FuncCode and stage 3	1	unused	3
BDSP 9124 StartStop	1		
BDSP 9124 Enable A	1		
total	16	total	16

For this stage, the enable B signal is independent of the frequency plan. The signal only needs to go active once every 4096 FFT clock cycles, when an FFT is completed and a new scale factor is ready. Since it is periodic, it can come from an Altera PLD.

The schizophrenic bit serves two purposes. It is both the function code bit for the BDSP 9124 and also a signal for the RAM in stage 3 of the FFT. Fortunately, there is no offset in the counter between these two functions, however there may be low level hardware registers which will make a difference in the timing of these two functions.

Bit Allocation for IFFT Stage 2

No bits are needed for the function code of the BDSP 9124 chip because it is always BFLY16 in this stage of the IFFT.

Table 9.6: Bit Allocation for IFFT Stage 2

Coefficient RAM		Data RAM	
Signal	Bits	Signal	Bits
select from 2048 twiddle coeffs	11	write address for data reshuffling; 8192 locations in RAM	13
BDSP 9124 StartStop	1	unused	3
BDSP 9124 Enable A	1		
BDSP 9124 Enable B	1		
unused	2		
total	16	total	16

Bit Allocation for IFFT Stage 3

In this stage, two bits are needed for the function code for the BDSP 9124 chip because it can have three different values: MOVD, BFLY4, or BFLY16.

Table 9.7: Bit Allocation for IFFT Stage 3

Coefficient RAM		Data RAM	
Signal	Bits	Signal	Bits
select from 2048 twiddle coeffs	11	write address for data reshuffling; 8192 locations in RAM	13
BDSP 9124 StartStop	1	unused	3
BDSP 9124 Function Code	2		
BDSP 9124 Enable A	1		
BDSP 9124 Enable B	1		
total	16	total	16

Bit Allocation for IFFT Stage 4

Table 9.8: Bit Allocation for IFFT Stage 4

Coefficient RAM		Data RAM	
Signal	Bits	Signal	Bits
select from 2048 twiddle coeffs	11	write address for data reshuffling; 4096 locations in RAM	12
BDSP 9124 StartStop	1	unused	4
BDSP 9124 Function Code	1		
BDSP 9124 Enable A	1		
BDSP 9124 Enable B	1		
unused	1		
total	16	total	16

In this stage, there are two possibilities for the function code for the BDSP 9124. Therefore one bit is needed to distinguish between them. In the programmed RAM under

the data RAMs, there are only 12 bits required for the address because there are only 4096 points being used in the data RAMs.

9.7.3 Delay Through Project

The goal here is to trace a data point as it moves through the project to see how long a delay is introduced by the project. Imagine that 2048 points from the A/D converter have been stored in the overlap buffer. Then another 2048 points from the A/D converter need to be pumped into the overlap buffer. This takes 2048 cycles at F_s. Then 4096 points from the overlap buffer are pumped through the BDSP 9124 chip into another memory chip. This takes 4096 cycles at the $2F_s$ clock.

Table 9.9: Delay Through Demultiplexer Stages

Stage of hardware	Number of samples to handle	Clock speed
from A/D converter into overlap buffer	2048	F_s
FFT stage 1: through BDSP 9124, into data RAM	4096	$2F_s$
FFT stage 2: through BDSP 9124, into data RAM	4096	$2F_s$
FFT stage 3: through BDSP 9124, into data RAM	4096	$2F_s$
IFFT stage 1: through BDSP 9124, into data RAM	8192	$4F_s$
IFFT stage 2: through BDSP 9124, into data RAM	8192	$4F_s$
IFFT stage 3: through BDSP 9124 into data RAM	8192	$4F_s$
IFFT stage 4: through BDSP 9124 into data RAM	8192	$4F_s$

Once a sample is input, it takes 8 clumps of 2048 cycles of the F_s clock, or 1.42 milliseconds before it is ready to be read by the demodulator.

Chapter 10

Results

Once the end-to-end simulation was completed and the signal-to-noise ratio computed, data could be collected for making bit error ratio (BER) curves. Noise was added at the input to the demultiplexer and the demultiplexer outputs were fed into the demodulator, which came up with an estimate of the original bit stream for each carrier. This estimate was compared with the original signal to find the number of errors. BER curves were then plotted as a function of signal-to-noise ratio.

10.1 Description of Trials for Results

10.1.1 Types of Plans Tested

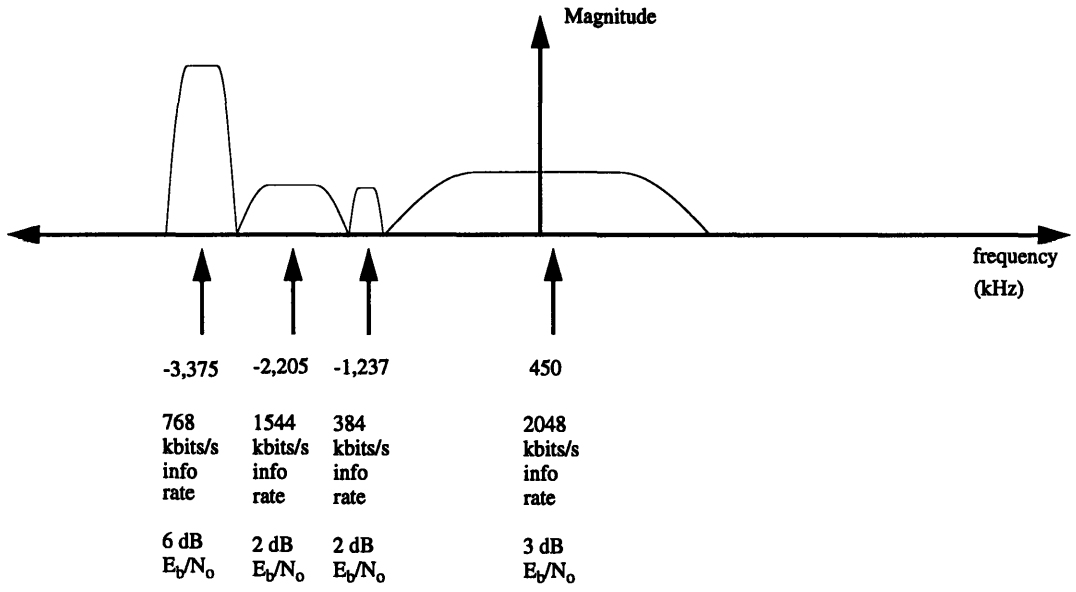
The tests were performed with a variety of carriers at a variety of power levels. The allocated bandwidths of the carriers were tightly packed, to maximize the effects of interchannel interference. For example, one of the plans tested is shown in Figure 10.1.

10.1.2 Criteria for Good Points on Result Curves

In the BER curves, some of the points are labelled as "good points" and the rest as "other points." There are two criteria for a point to be a "good point." First, there had to be at least 100 errors between the input and demodulator output. That ensured good statistics for the error rate. Second, the "good points" had to be from a carrier that was surrounded on both sides. For example, in Figure 10.1, the carriers b1544 and c384 had interchannel

interference from both sides, but the other two channels only had interchannel interference from one side.

Figure 10.1: A sample of a Test Plan



Carriers were spaced so that their allocated bandwidths were adjacent, in order to maximize effects of interchannel interference.

10.1.3 Limits on Sizes of Tests

It was difficult to test carriers at low bit rates. When the bit rate is low, it takes a long time for a significant number of errors to occur, and this taxes the computer's time. In order to run simulations over such long stretches, many samples of input data must be generated for each carrier, and these large files tax the computer's memory. For the demultiplexer simulation, it took 80 minutes to run 2 million samples into the simulator. For each carrier, the files with 2 million samples took about 20 MBytes. To work with five carriers at once meant reading from 100 MBytes of hard drive space. Hard drive space became a scarce commodity.

There are three regimes of signal-to-noise ratios. The first is where the ratio is so low that the demodulator can't lock on. The second is intermediate, where the demodulator can lock on and many errors can be recorded. The third is where the ratio is so high that it is difficult to measure a large number of errors.

The specification on the demodulator is that it should be able to lock onto the signal at signal-to-noise ratios of 2 dB, where signal-to-noise ratios are measured as the energy per bit, divided by the noise power per Hz (E_b/N_o).

For the sizes of the simulations chosen, the highest rate carriers (e2048) were about 4.3 Mbits/s transmitted bit rate and could give good points with more than 100 errors at signal-to-noise ratios as high as 6 dB E_b/N_o. At the other extreme, the low bit rate carrier (c384) had a transmitted bit rate of about 614 kbits/s, and good points with more than 100 errors at signal-to-noise ratios only up to 4 dB E_b/N_o.

10.2 Result Graphs

For these four types of carriers, data was run through the demultiplexer and demodulator end-to-end simulation. The carriers were chosen so as to span a wide range of bit rates. The "b1544" carrier was included because the 1544 kbit/s information rate is the only one in the system that is not a multiple of 64 kbits/s.

The carrier with an information rate of 768 kbits/s was always used on the edge, at the lowest or highest frequency. It never had interchannel interference from both sides, so there are no "good points" for that carrier. The results are shown for comparison.

Table 10.1: Carrier Types That Were Tested

Info Rate (kbits/s)	Trans. Bit Rate (kbits/s)
64	45.5
384	614.4
768	1092.3
1544	2186.7
2048	4288.0

The hash-marked triangle in all graphs is the specification that the combined demultiplexer and demodulator system has to meet. There is an allowable degradation of 1 dB E_b/N_o at BERs of 10^{-2}, 10^{-3}, and 10^{-4}.

Figure 10.2: BER Curve for Carrier with Information Rate of 2048 kbits/s

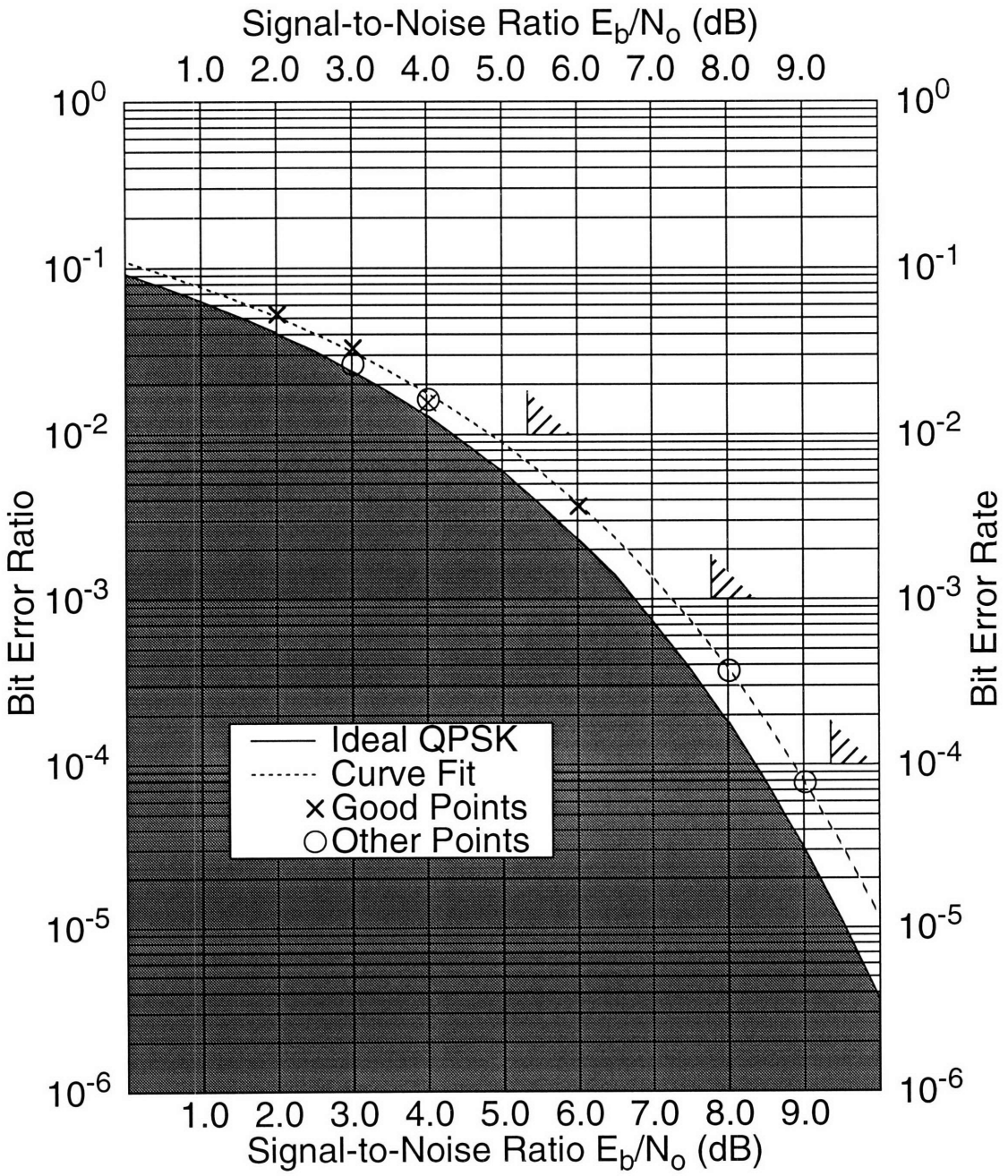


Figure 10.3: BER Curve for Carrier with Information Rate of 1544 kbits/s

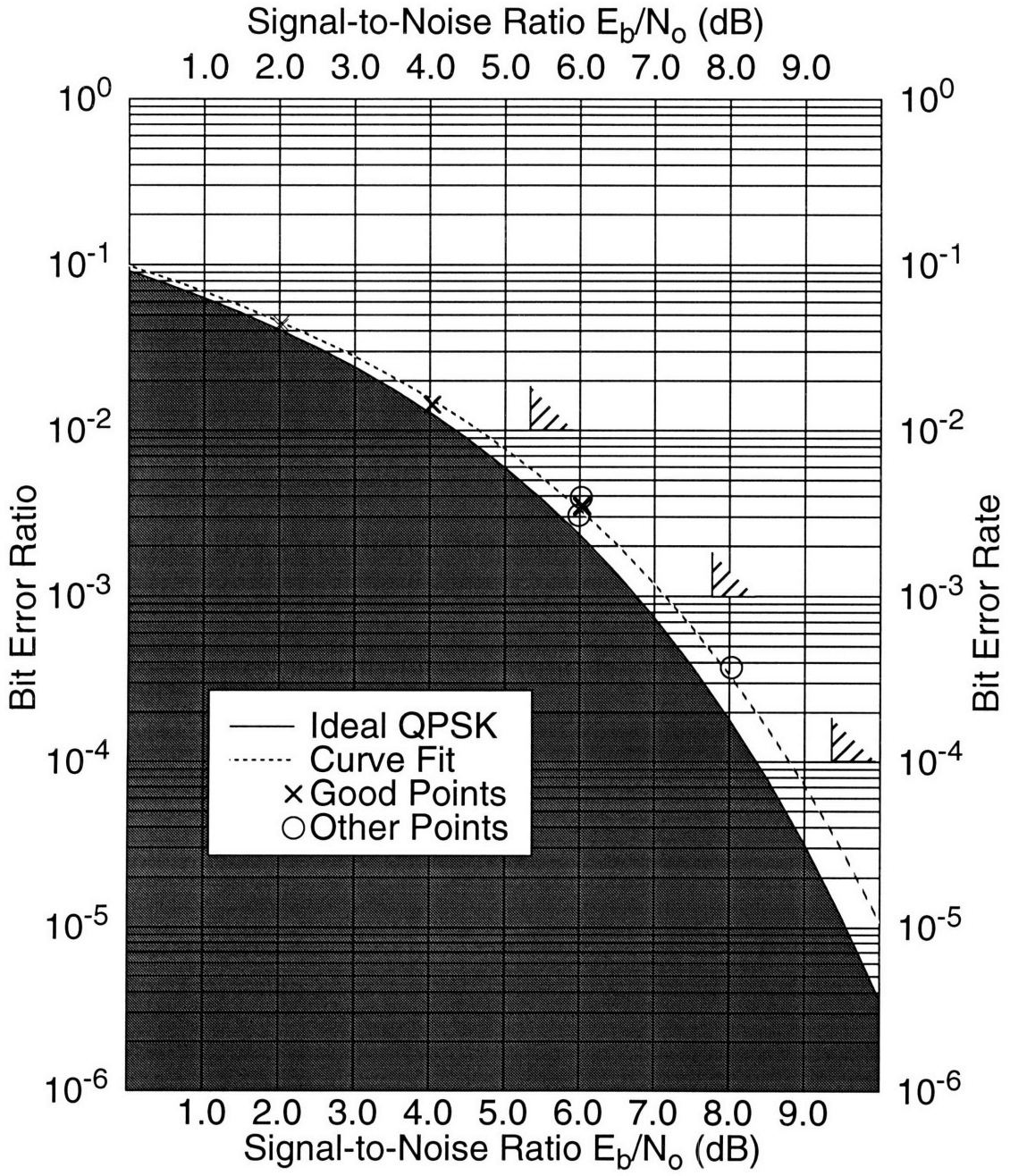


Figure 10.4: BER Curve for Carrier with Information Rate of 384 kbits/s

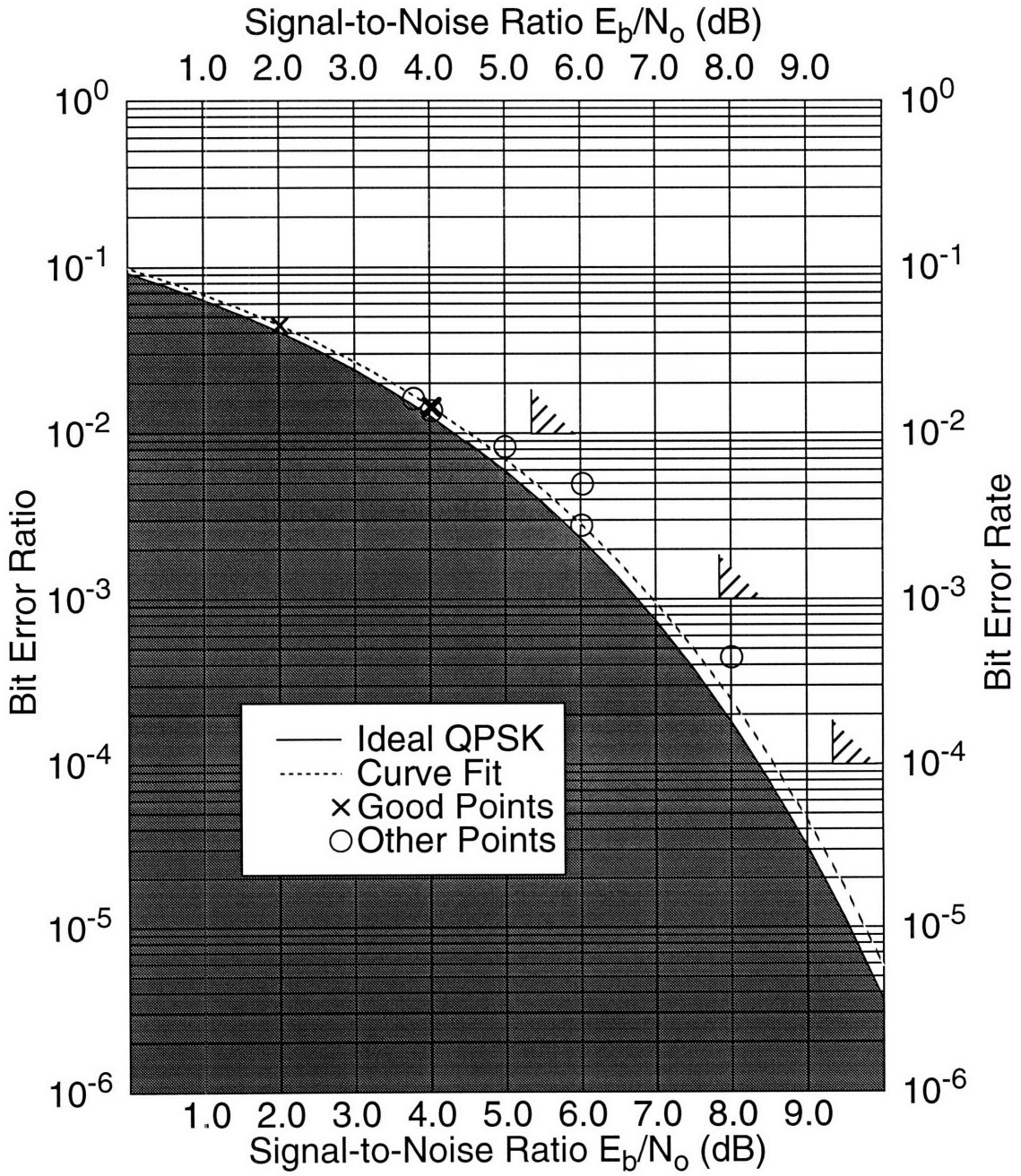
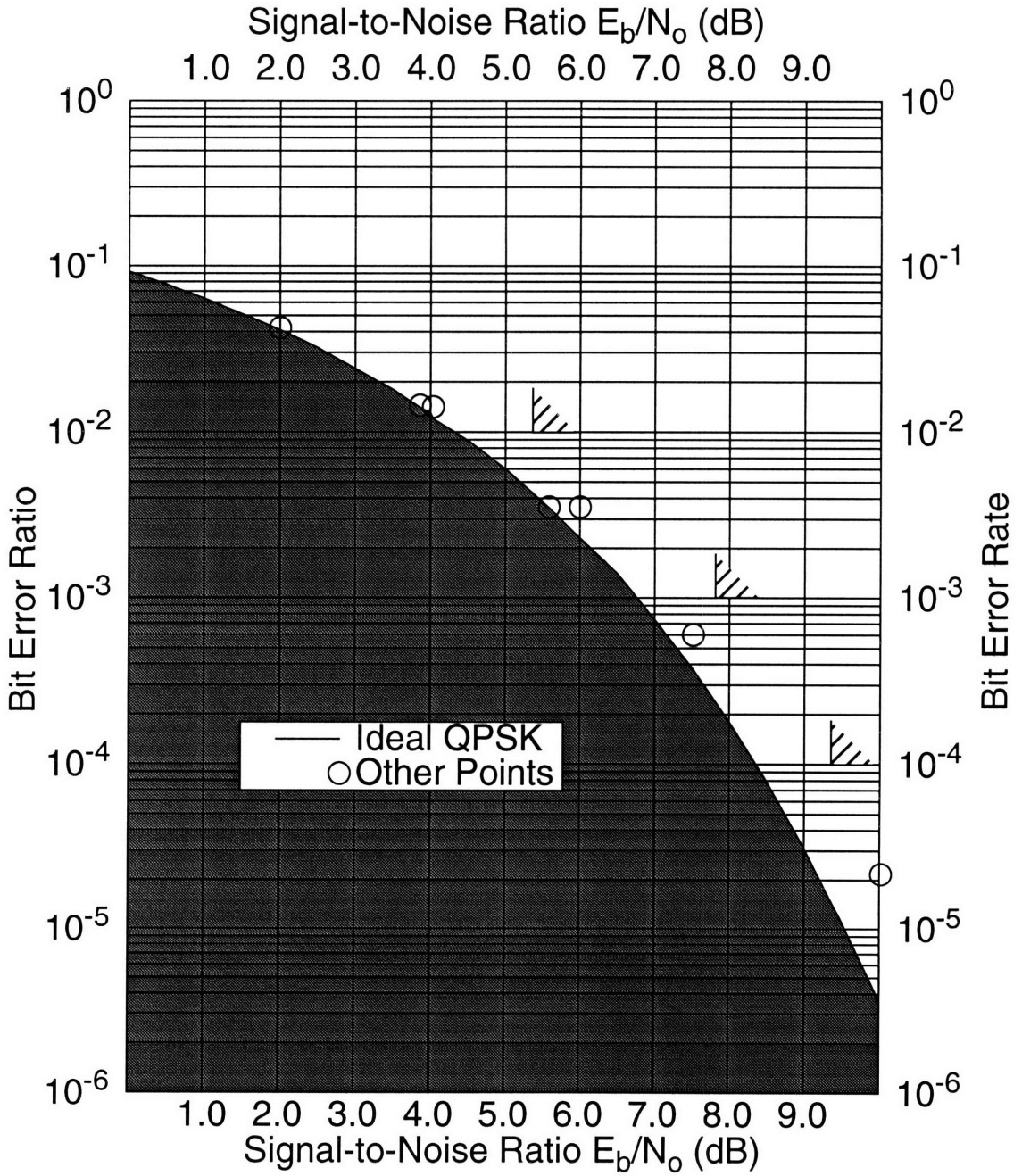


Figure 10.5: BER Curve for Carrier with Information Rate of 768 kbits/s



10.3 Frequency Offset

If there is a frequency offset due to an offset error in the local oscillator of the transmitter or the receiver, then there will be a small beat frequency in the data at the end of the IFFT. The demodulator can handle small offsets, up to ±2.5% of the symbol rate or 5.5 kHz, whichever is less. The demodulator is known to meet these specifications for single carriers with no demultiplexer. Two sample test runs with a maximum permissible frequency offset showed no extra degradation.

Chapter 11

Remaining Items

Seven items remain. They are described here in roughly decreasing order of importance.

11.1 On-the-Fly Switching

This is a major requirement of the demultiplexer. The demodulator is capable of handling it and the demultiplexer must be also. Although blocks for this were built into the simulation, there was not enough time to test this capability.

In order to add this capability to the simulation, the output of the demultiplexer simulator must be made more sophisticated. Currently, it feeds the signal from each carrier into a separate file, for examination by the demodulator. However, to test on-the-fly switching, the demultiplexer would have to send its output to one file and also send control signals to the demodulator, marking off the time divisions between blocks of data for different carriers.

Furthermore, the blocks which have been placed in the simulation for testing on-the-fly switching would have to be provided with control signals for switches and put to use. The necessity of thinking through these control signals in the simulation would provide background for implementing them in hardware.

Currently, the thinking is that on-the-fly switching will work, but the whole point of the simulation is the test things out before unexpected difficulties arise in the hardware.

11.2 BDSP Chip Latency

The C source code simulator for the BDSP 9124 chip works with data in groups of 16 for the BFLY16 function code and in groups of 4 for all other function codes. The SPW simulation is built around this. For the BFLY16 function code, 16 data points must be put into the C source code simulator and at the moment when the last point is put in, the first point of output is ready. The latency for BFLY16 in the software is therefore 15. Similarly, the latency for all other functions is 3.

The lookup table in the software that generates address patterns for data reshuffling actually uses these skewed values. The "latency" is 15 for the BFLY16 function code and 3 for all other function codes. These values do not match the hardware.

The software that generates actual addresses for the hardware must have the values in the lookup table changed. This is a very simple matter; in fact, the real values that correspond to the hardware latency are in comments in the code, next to the spot where they belong. However, it does mean that there is a discrepancy between what code is used for simulation and what code is used for hardware.

Although it seems difficult to modify the parallelizer and serializer to make the simulation of the BDSP 9124 act more like the actual chip in terms of latency, it may be worthwhile because then at least the same code used for simulation can be reused - without modification - for the hardware. That way, debugging can be simplified, and there would be no worries that any errors that come up in the hardware debugging are due to this discrepancy.

11.3 Order of Data Entry for BFLY2 and BWND2

In [9124 UG], the figure on page 3-6 for the BFLY2 (and the similar diagram for

BWND2) shows the data going in and coming out in the following order: A, C, B, D. This is different from the order used in simulation, which was A, B, C, D. However, this may be a misinterpretation of the diagram. The diagram may mean that A and B are indeed the first two data samples to enter the chip, and the interpretation of the drawing may be wrong. These is likely because the user's guide for the BDSP 9124 and the BDSP 9320 do not mention anything about the order of data entry. Their patterns for reshuffling and twiddles do not make it seem that the order is anything but sequential. However, it is something to keep in mind when debugging of the hardware, in case it slipped by the SPW simulation and the simulation using the BDSP 9124 C source code.

11.4 Scaling Issue

Fortunately, this is an issue that seems to be solved now. To illustrate the scaling issue, consider two sets of data which enter an inverse transform. One is slightly larger than the other and the DSFO values tell the BDSP chips to kick the level down to prevent overflow. At the output, this slightly larger signal appears only about half as large as the subsequent block of data. The worry was that these sudden changes in amplitude would cause bursts of errors in the demodulator.

One idea for fixing this was to use not only the DSFO and DSFI signals but also the BFPO and BFPI signals (block floating point input and output). The DSFO and DSFI values specify how much to reduce the input to a BDSP 9124 chip in order to prevent overflows in calculations. On the other hand, the BFPO and BFPI values are designed to keep track of the accumulated exponent, and can be used for normalization at the end of the demultiplexer. These values are used in a barrel shifter that controls the input levels of the signals entering the demodulator.

The FIFO in the hardware for the DSFO values has 9 bits, and this is perfectly matched to the DSFO and BFPO values, which have 3 and 6 bits respectively. Thus both can be passed from stage to stage and used to control the signal levels. Simulations using both of these signals appear to be working and eliminate the scaling problem.

11.5 Frequency Offsets

As mentioned in the chapter about results, this issue has been explored in a couple of simulations. Frequency offsets within the allowable specification appear to pose no problem to the demultiplexer/demodulator system. However this exploration has been tentative and a more methodical exploration would be more satisfying.

11.6 Different Precision Levels in the A/D

For this project, the precision levels at all stages including the A/D converter appear to be completely satisfactory. However, it may be a good idea to run simulations with different precision levels in the A/D converters for two reasons. First, actual A/D converters are not ideal and according to some data sheets, an 8 bit A/D converter at 11.52 MHz only provides $7\frac{1}{2}$ bits of resolution. Thus it might be a good idea to simulate a 7 bit A/D converter to get a lower bound on performance. Second, it would be interesting to see how sensitive the demultiplexer/demodulator system is to variations in the A/D converter precision. Since the degradation with an 8 bit A/D converter is only a few tenths of a dB E_b/N_o, it may not be worthwhile to consider a 10 bit A/D converter because that may only provide negligible improvement. However, if a 7 bit A/D converter proves to be much worse than an 8 bit A/D converter, then it may be wise to use a 10 bit A/D converter to stay far away from a region of bad performance.

However the issue of various precision levels is somewhat removed from this demultiplexer and more applicable to other projects. This is a ground based project where power is not a concern. For other projects slated to go on board satellites, power is a precious commodity. The idea is to use the simulator for these other projects and see how many bits can be cut away without degrading performance unacceptably. For instance, maybe the IFFT needs less than 16 bits of precision. Every bit that can be cut is a power savings.

11.7 User-friendly software

Finally, the current software for inputting the frequency plan is awkward and based on text files. Worse, it requires the user to understand the constraints on the number of carriers that can be handled by the demultiplexer. It is easy to imagine a windows-oriented program in which a user can use a mouse and get feedback from a graphical screen display in order to input a new frequency plan. The program could automatically check the user's inputs against the constraints and inform the user if the selections were okay or not, and if not, why not. This software could then generate a text file in the form that is currently used by the software that generates the addressing patterns. The windows program could call the program and have it generate addressing files automatically.

Chapter 12

References

12.1 Cited References

[9124 UG] *LH9124 Digital Signal Processor User's Guide.* SHARP Electronics Corporation, 1992.

[Feher] Feher, Kamilo. *Digital Communications; Satellite/Earth Stations Engineering.* Prentice-Hall Inc., NJ, 1983.

[IESS 308] "Performance Characteristics for Intermediate Data Rate (IDR) Digital Carriers." INTELSAT Document IESS-308 (Rev. 8).

[IESS 309] "Performance Characteristics for International Business Service (IBS) Digital Carriers." INTELSAT Document IESS-309 (Rev. 6).

[OS] Oppenheim, Alan. V. and Schafer, Ronald W. *Discrete-Time Signal Processing*, Prentice Hall, New Jersey, 1989.

[Poklemba] Poklemba, J. J. "Pole-zero approximations for the raised cosine filter family." *COMSAT Technical Review*, Volume 17, Number 1, Spring 1987.

[Shenoi] Shenoi, Kishan. *Digital Signal Processing in Telecommunications.* Prentice Hall, NJ, 1995.

[Snyder] Snyder, J. "9-MHz Subband Architecture for MCDD Demultiplexer." Memorandum MD-96-021, July 20, 1996.

[Thomas] E-mail discussion with Jim Thomas, Scientist, COMSAT Laboratories.

12.2 Consulted References

"18.310: Introduction to Discrete Applied Math, Massachusetts Institute of Technology, Chapter V: Generating Functions and Fast Arithmetic." Last revised by Professor Jim Propp, November, 1995.

"Advanced On-Board Digital Processing." Final Report 920300/92-42, COMSAT Laboratories, July 1992.

Agarwal, R. and Cooley, J.W. "Algorithms for Digital Convolution." IEEE Transactions on ASSP, Vol. 25, 1977, 392-410.

Bergland, G. D. "A guided tour of the fast Fourier transform." IEEE Spectrum, July 1969.

BDSP9320 DSP Memory Management Unit User's Guide. Butterfly DSP, Inc., 1995.

Brigham, E. Oran. *The Fast Fourier Transform.* Prentice-Hall, Inc., New Jersey, 1974.

LH9124 Digital Signal Processor Real Time Simulator User's Guide. Sharp Electronics Corporation, 1993.

LH9320 Address Generator Real Time Simulator User's Guide. Sharp Electronics Corporation, 1993.

Multicarrier Demultiplexer/Demodulator Presentation by COMSAT Labs. July 1995.

Siebert, William M. *Circuits, Signals, and Systems.* McGraw-Hill, NY, 1986.

Snyder, J. "Demultiplexer Architecture for MCDD." Memorandum MD-96-006, March 12, 1996.