

Design, Implementation and Testing of a Digital Baseband Receiver for Spread Spectrum Telesensing

A Thesis Presented for the Degree of

Master of Science

The University of Tennessee, Knoxville

Brian Parker Chesney

December, 2000

Abstract

Telesensing involves receiving data wirelessly from a remote sensor. Generally, the sensor node is fixed and configured to transmit only or perform very basic reception. Because of their low power consumption, telesensors can be powered by a battery for long periods of time without a measurement or transmission interruption. This allows several nodes to be placed at strategic locations and creates a need to have all the individual data collected and processed at a centralized location. Frequency Division Multiple Access (FDMA) provides robust data transmission from multiple telesensors to the same receiver at the cost of added bandwidth.

This thesis focuses on the digital recovery of spread spectrum data in an FDMA system. A general digital spread spectrum receiver architecture is given (without transceiving capability) and each component is designed, implemented and tested in the receiver as a whole. A sliding correlator with a threshold is used to synchronize the pseudonoise (PN) code used to encode the data with the incoming data. System clocks are also recovered from the incoming data and distributed to the downstream modules. The design is implemented in an FPGA and tested with favorable packet error rate results in an FDMA system. The components of the digital receiver processor could be used in conjunction with a Costas Loop demodulator to provide CDMA for multiple sensors and its functionality and robustness are confirmed in this thesis. This would fit into a complete CDMA, allowing the demodulator to discriminate between various sensors.

Table of Contents

1.0 Introduction.....	1
1.1 Telesensing	1
1.2 Scope of Thesis	3
2.0 Background	6
2.1 Wireless Communication	4
2.2 Multiple Access	7
2.3 Spread Spectrum.....	10
2.4 PN Codes.....	13
3.0 System Overview.....	25
3.1 Data Acquisition Chip	26
3.2 Low-power Transmitter	27
3.3 FSK Receiver	29
3.4 Component Limitations.....	29
4.0 Digital Receiver Architecture	31
4.1 Clock Recovery and Chip Resolution	31
4.2 Digital Despreader	34
4.3 Embedded Protocol Removal.....	39
4.4 Packet Detector	40
4.5 Acquisition Processor	43
5.0 Implementation	45
6.0 Testing and Results	47

7.0 Conclusions and Future Work	58
References	63
Appendix - VHDL Code	66
Vita	103

List of Figures

- Figure 1: FSK in the Frequency Domain
- Figure 2: FDMA in the Frequency Domain
- Figure 3: A TDMA Frame
- Figure 4: Ideal FSK
- Figure 5: Spread Spectrum
- Figure 6: Code Division Multiple Access
- Figure 7: Linear Feedback Shift Register
- Figure 8: Autocorrelations
- Figure 9: Cross-correlation between Code 1 and Code 2
- Figure 10: Spreading with Code 1
- Figure 11: Spreading with Code 2
- Figure 12: Properly Recovered Data
- Figure 13: Improperly Decoded Data
- Figure 14: Sensor Data Link
- Figure 15: ACQ2 Block Diagram
- Figure 16: RFMD 2510 Block Diagram
- Figure 17: RFMD 2945 Block Diagram
- Figure 18: Direct-Sequence Spread Spectrum Receiver
- Figure 19: Digital Receiver Block Diagram
- Figure 20: Polarity Decoder Dither Conditions
- Figure 21: Polarity Decoder Block Diagram
- Figure 22: Despreader Correlator Block Diagram
- Figure 23: Despreader Correlator Histogram
- Figure 24: Packet Detector Block Diagram
- Figure 25: Acquisition Processor Block Diagram
- Figure 26: Initial Wireless Testing
- Figure 27: Repackaged Wireless Transmitter
- Figure 28: Internal Circuitry of Wireless Transmitter
- Figure 29: Repackaged Wireless Receiver
- Figure 30: RF Front-End of Repackaged Wireless Receiver
- Figure 31: Digital Spread Spectrum Baseband Processing Board
- Figure 32: Repackaged Wired Testing
- Figure 33: Repackaged Wireless Testing

Acknowledgements

I would like to thank the following people for their help and support throughout my education at the University of Tennessee. First, I would like to thank Jerry Stoneking for his role in getting me accepted to the University of Tennessee. Also, I would like to thank Charles Britton and James Rochelle for admitting me into the UT Joint Program. I would like to thank Charles Britton for his guidance and support in additional capacities, that of Principle Investigator on the research project that funded this work as well as a member of my thesis committee. I would also like to thank the rest of my committee, Danny Newport, my Major Advisor, and Daniel Koch for their direction and invaluable contributions to helping me finish my thesis.

I would like to thank the researchers and staff of the Monolithic Systems Group of ORNL's Instrumentation and Control Division, especially Gary Alley, Bill Bryan, Lloyd Clonts, Nance Ericson, Shane Frank, Gayle Jones, and Gary Turner. I would also like to thank Gary Turner for his patience and sharing with me his immense technical knowledge. I would also like to thank all of the members of the Joint Program, particularly, Derek Austin, Eric Bolton, Andrew Moor, Aaron Symko, and Stephen Terry for their friendship and support. Also, I would like to thank Jeff Falin for his pivotal role in my acceptance into the Joint Program.

I would like to thank my parents, Alan and Barbara Chesney, for their patience and support. I would also like to thank them for the care and advice that has always been plentiful and much appreciated. Similarly, I would like to thank my sister and brother-in-law, Sara and Elliot Tucker for their love, support, and excellent advice, as

well. I thank my grandparents, J. Dukehart and Marjory Chesney for their tireless and unconditional support. I would also like to thank Roger Borchers, Blair Brown and Greg Holloran as well as Benji and Lorna Wood, Alex Morris, Michael Ruff, Sheila Smitherman, and Meredith Novy for their continued support through the years.

This research was funded by a grant from Graviton, Inc. I thank them for their support and contribution, especially Larry Goldstein, Steve Tietsworth, and David Fern. The Principle Investigator for this research was Charles Britton. The majority of the work was performed at the Oak Ridge National Laboratory, managed by UT-Battelle for the U.S. Department of Energy under contract No. DE-AC05-00OR22725.

1.0 Introduction

Low-power wireless sensors can be used to efficiently report specific conditions at a remote location. Micromachined cantilever sensors fabricated on a silicon die can be used to provide the sensor data, providing a low-power solution to acquiring certain data [1]. Because these sensors can be so low-power that they could run off of a battery, the parts used to transmit the data need to be similarly low-power. Although, the wireless link provided by this project is independent of the data being transmitted on it, it was designed for a telesensing application. The transmitter aspect of this wireless solution is provided by an in-house, ORNL-developed analog-to-digital converter (ADC) chip and a commercial low-power transmitter. To provide data robustness and provide multiple access in future generations, spread spectrum communication is employed. This requires a digital receiver to recover the spread spectrum signal, despread it and interface with a Personal Computer (PC) to display the data. This thesis focuses on the design of the digital spread spectrum receiver to provide remote access for wireless data transfer.

1.1 Telesensing

Wireless sensor data transmission is free from certain limitations of cellular telephony. Transmission bursts do not need to be carefully coordinated to appear as a seamless stream of continuous speech to the human ear. Therefore, data can be transferred every few seconds or more as opposed to the millisecond increments required by cellular telephony. As an example, GSM, the Time Division Multiple Access (TDMA) cellular

standard used for wireless telephony in Europe, requires that the speech for a given call be transmitted every 4 ms in 125 us frames. This frequency of transmission would be overkill for a sensor that only needs to update its status on the order of seconds or minutes or even once an hour.

As a result of this long lag between transmission bursts, the RF front-end circuitry can be turned off since the period of time between transmissions greatly dwarfs the time required to turn it back on and transmit a burst of data. Also, the digital transmission processing core, responsible for relaying the sensor data to the RF circuitry, can be put into sleep mode to save power. Both of these power down features greatly reduce the power consumption of the system.

Additional power conservation can be attained if the transmitter is not configured for reception as well. The transmitter does not have to interrupt its sleep mode to spend power on receiving instructions from the host sensor data processor. However, even if this were a desired trait of a potential wireless telesensor implementation, sensor data does not require computationally-expensive and power-hungry digital signal processing to restore synthesized human speech, as with digital cellular telephones.

These three features of telesensing, RF and digital circuitry power conservation and simplified or nonexistent transceiving in the transmitter allow telesensing to have tremendous power savings over digital cellular telephony. Therefore, the data transmission circuitry can run autonomously using a single compact battery, instead of a traditional cellular telephone battery. These wireless telesensor transmitters can be left alone to transmit data reliably for months instead of needing to be recharged every couple of days as with a standard cellular telephone battery. The telesensors

transmit data less frequently and at lower rates than cellular telephones, but this is usually appropriate for their application to sensor data.

1.2 Scope of Thesis

The goal of this thesis is to move an analog voltage representing the output from an analog sensor wirelessly from the stand-alone sensor site to a host PC for display. Specifically, this thesis focuses on the design of the digital spread spectrum receiver used to recover the transmitted data. The architecture was designed for telesensing, but, since the actual transmission and reception of data is independent of the application, the wireless link was not tested with sensors as inputs. An appropriate background in digital communications is developed so that design considerations in the digital receiver can be appreciated. Then, the system overview is given, detailing the specific components used in the system and how this affects the design of the receiver. Next, the architecture of the digital spread spectrum receiver is detailed and its functionality explained. Later, the implementation of the design and the testing and results of the implementation are discussed. Finally, lessons learned from the project and conclusions on digital spread spectrum design are presented.

2.0 Background

2.1 Wireless Communications

The purpose of wireless communications is to transmit data through an ambient physical medium, such as air, so that a physical channel does not need to be built and maintained, such as a cable or twisted pair telephone line. This requires coupling the intended transmitted signal to the medium, something easily done with tethered communications since the channel is designed to support the transmitted signal. For wireless communications, this requires building an antenna to convert an incident voltage, the medium of the signal to be sent, to a radio frequency (RF) wave, a different signal but one that can propagate through the desired medium.

The size of the antenna required to couple a given signal to the air via RF waves varies inversely with the frequency of that signal. Lower frequency signals require prohibitively large antennas and so must be converted to a higher frequency signal. This is done by using a higher frequency carrier that is modulated by the signal to be sent.

The data to be sent can be transmitted as offsets from the carrier frequency. This is known as frequency shift keying (FSK) and is the method used to transmit the information in this system. In this case, the receiver/transmitter pair employ a binary frequency shift keying (BFSK) alphabet to transmit data. The general analytic expression for the alphabet is:

Equation 1:
$$s_i(t) = \sqrt{\frac{2 \cdot E_b}{T}} \cdot \cos(2\pi f_i t + \varphi)$$

where t is time-limited between 0 and T , i is a member of the binary alphabet $\{0,1\}$ and the energy transmitted in one bit is E_b . Different frequencies are used to transmit a 0 and a 1 usually a fixed frequency increment centered around the carrier frequency.

In order to ensure data fidelity at the receiver, the symbols of the BFSK alphabet must be orthogonal to each other, so that they do not interfere with one another. Since the signals are time-limited to T seconds, they can be expressed as

Equation 2:
$$s_i(t) = \sqrt{\frac{2 \cdot E_b}{T}} \cdot \cos(2\pi f_i t + \varphi) \cdot \text{rect}\left(\frac{t}{T}\right)$$

where $\text{rect}(t/T) = 1$ when $|t|$ is less than or equal to $T/2$ and 0 otherwise. The Fourier transform of $s_i(t)$ is

Equation 3:
$$S_i(f) = \mathfrak{F}\{s_i(t)\} = T \cdot \frac{\sin(\pi T(f - f_i))}{\pi T(f - f_i)}$$

For signals separated by multiples of $1/T$ Hertz,

Equation 4:
$$f_1 - f_0 = \frac{m}{T}$$

where m is an integer greater than or equal to 1. The value of $1/T$ is known as the minimum frequency separation for the two signals. The frequency domain representation of these signals evaluated at multiples of this minimum separation is nonzero for one signal while the other is zero and vice versa. This is evidenced by Figure 1. Therefore, the two signals do not interfere with each other at all and are orthogonal [2], [3], [4], [5].

For FSK digital communication, an FM superheterodyne receiver is usually employed. It mixes the incoming signal with a local oscillator to an intermediate frequency (IF). By downconverting to an IF instead of baseband, the receiver does not have to match the frequency in the local oscillator exactly with the incoming signal. This creates images at other frequencies, but these can be suppressed by appropriate filtering before the signal is mixed down to IF.

Since it is much easier to maintain a constant bandwidth in a fixed filter than a tunable one, the channel filtering is done at IF. Also, high-stable gain is more difficult to provide in a tunable amplifier than a fixed-frequency amplifier. This makes covering a wide frequency band easier and is why superheterodyne receivers are standard for FM radios [6].

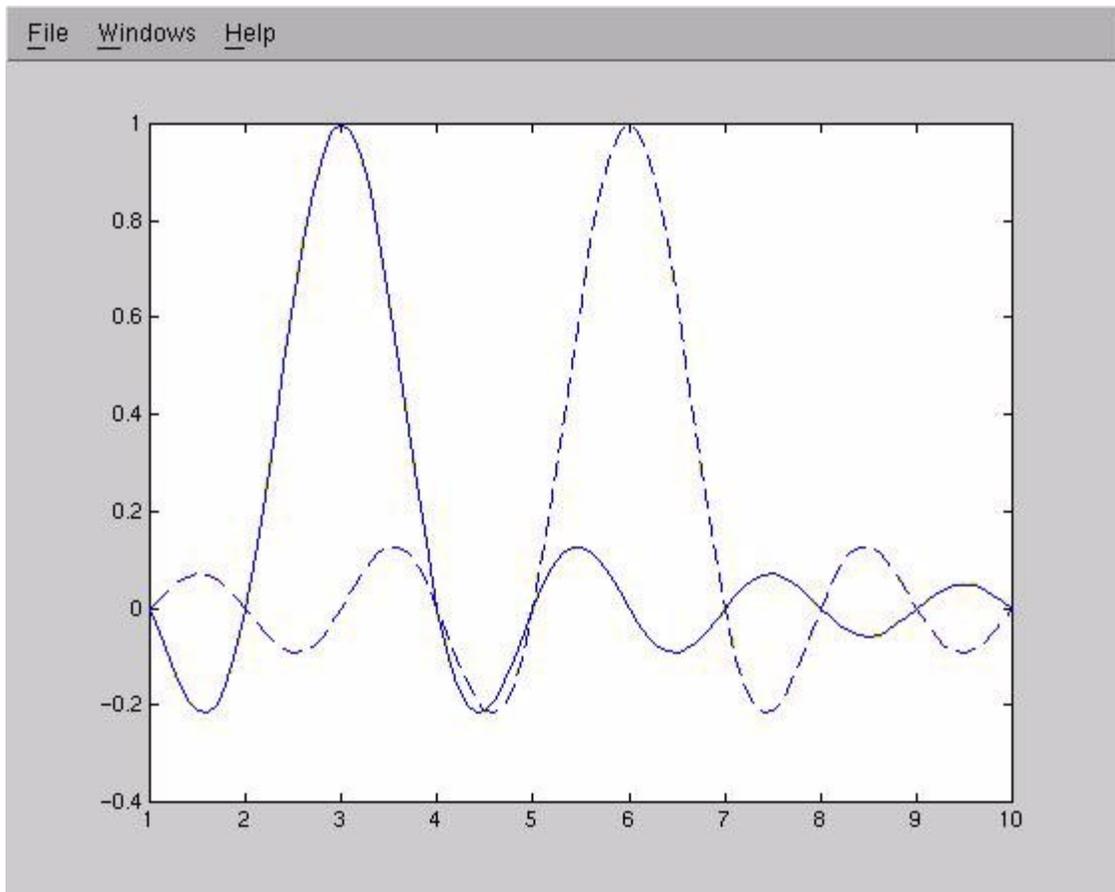


Figure 34: FSK in the Frequency Domain

2.2 Multiple Access

To employ wireless communication effectively, especially for telesensing applications, multiple users will need to be accommodated. For a sensor designed to detect the presence of a certain element, it usually would not be sufficient to just have one sensor in one place. Generally, an array of sensors would be used to cover a larger area.

These results would need to be coordinated and analyzed at a central location. If the receiver is able to listen to all of these sensors, then only one PC is needed to display and

assimilate the information gathered. This requires multiple sensors having wireless access to the receiver.

There are three primary methods of allowing multiple users access to wireless RF communication: FDMA, TDMA, CDMA. Frequency Division Multiple Access (FDMA) separates users by the carrier frequency they use to communicate. The separation between carriers must allow for the full spectrum of the signal to be communicated so that signals from adjacent carriers do not overlap as shown in Figure 2. This requires a certain amount of bandwidth, BW , for a given number of users, n , wishing to use f_b amount of the frequency spectrum, with signals separated by $f_{di} - f_{di-1}$.

Equation 5:

$$BW = \sum_{i=1}^n (f_{bi} + \Delta f_{di})$$

There are multiple access schemes that allow more effective use of this amount of bandwidth since each user gets their own amount of bandwidth to occupy, even when they are not using it. A more efficient way of allocating bandwidth would be allowing users to share a carrier. Time

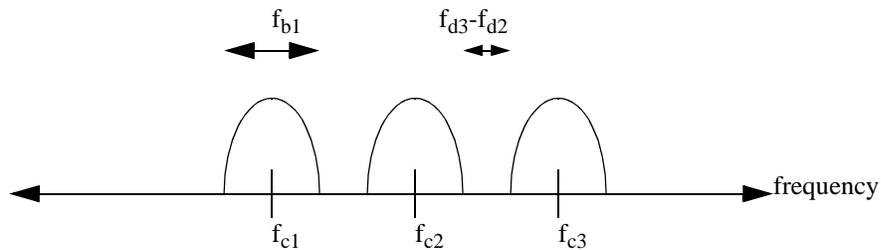


Figure 35: FDMA in the Frequency Domain

Division Multiple Access (TDMA), lets a certain number of users share a carrier equally. Each user is assigned a time slot in a frame that is transmitted on the carrier, repeating periodically. In Figure 3, for example, the frame has a period, T , supports m users, and each one transmits on the carrier for T/m seconds. For the same amount of bandwidth as above, $m*n$ users can be supported instead of n . This works well for applications where small delays in transmission bursts can go relatively unnoticed. An example of this is the GSM standard for cellular telephony, which specifies that a 4-ms frame accommodate 32 users for a transmission time of 125-us each. This 4-ms lag in speech is barely, if at all, perceptible by the human ear, so is adequate for relaying human voice [7].

Code Division Multiple Access (CDMA) allows multiple users to share the same carrier by encrypting each user's message. This requires a code that can only be decoded by the appropriate decryption key. To reduce computational complexity, the encoding algorithm should be easy to invert given the encoding key. The exclusive OR (XOR) function is an easily invertible binary function and is used for encoding a message for CDMA purposes. Since the encoding and decoding functions are well-known, the problem now reduces to finding a sufficiently strong encrypting code that does not give away the message sent and keeping that key secret.

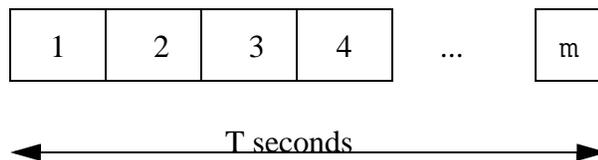


Figure 36: A TDMA Frame

CDMA uses a code in the transmitter that runs at an integer multiple of the data rate to encode data, called spread data. The rate of the code, called the chipping rate, must run faster than the data rate because each data bit is being encoded before it is transmitted. It must run at an integer multiple so that the receiver can recover the message data from the spread data, since the receiver has prior knowledge of the code but no knowledge of the timing (i.e., is not passed a clock from the transmitter). Making these encrypting codes, also called spreading codes, unique with respect to each other allows multiple users to share the same carrier without interfering with each other's data. The strength of CDMA lies in the generation of strong individual codes as well as a set of codes that are unique to each other [2], [8], [9].

2.3 Spread Spectrum

The frequency spectrum of a signal is spread when the signal is combined, through modulo-2 addition with a pseudorandom or pseudonoise (PN) sequence. As shown in Figure 4, the original data signal, in the case of FSK modulation, consists ideally

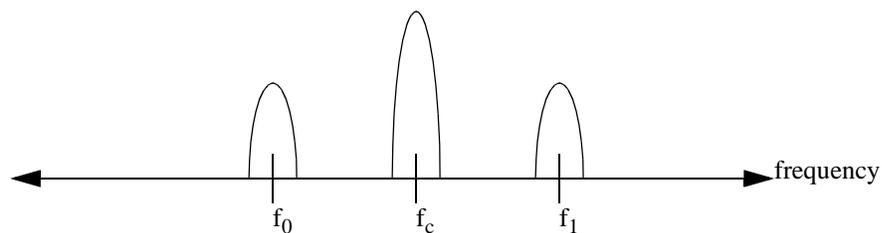


Figure 37: Ideal FSK

of narrow signals at the carrier frequency and also at fixed frequency increments from the carrier. The frequency spectrum of the ideal signal is limited to the bandwidth needed to include the small FSK increments. The idea behind combining this signal with a PN sequence is to create a signal that looks like noise when not properly decoded. This is achieved by making the signal appear to be random. A truly random signal contains all possible frequency components equally.

As a data signal is randomized, its frequency spectrum must be spread to include more of the frequency spectrum. However, it is only spread to a certain extent since the PN sequence is not truly random and repeats with some determined period.

Figure 5 shows a broader spectrum than the signals in Figure 4. CDMA is implemented using spread spectrum at the cost of added bandwidth. However, this cost is offset by the gain in multiple access afforded by the orthogonality of the PN codes. It is possible to use the PN code in different ways to encrypt communication. If the PN sequence is used to directly modulate the carrier, this is direct sequence spread spectrum. On the other hand,

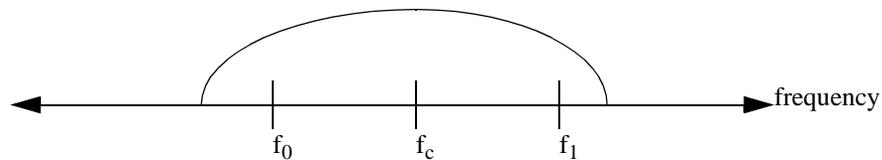


Figure 38: Spread Spectrum

if it is used to shift the carrier frequency in discrete increments, this is known as frequency hopping [8]. Each user is given their own PN code and may enjoy secure communication independent of other users on that same carrier frequency as shown in Figure 6.

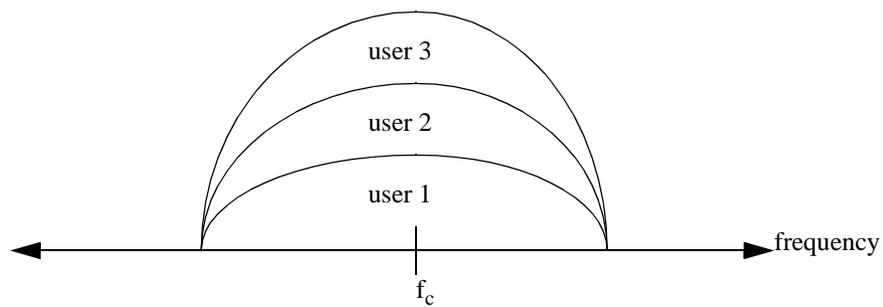


Figure 39: Code Division Multiple Access

2.4 PN Codes

If the encrypting code is a suitably selected sequence, then only the exactly aligned code reveals the message and a misaligned code reveals nothing. This can be measured by the digital normalized autocorrelation function. The autocorrelation, $R_x[n]$,

Equation 6:

$$R_x[n] = \frac{1}{N} \sum_{i=0}^{N-1-n} x[n+i] \cdot x[n]$$

measures the correlation between $x[n]$ and a time-shifted version of itself. This repeats at least with period N , since N is the length of the code. The ideal autocorrelation function for an encrypting code then is the Kronecker delta function which only has a nonzero value at 0. Transforming white noise from the frequency spectrum to the time domain also yields a similar function. Imitating white noise in an encrypting sequence is desirable because an improperly decoded signal will resemble white noise and give no useful information about how to properly decode the signal.

Since deterministic hardware is used to generate these codes, they cannot be truly random; but can approximate random binary strings. There are three general rules for analyzing digital codes to determine if they sufficiently resemble random bits. First, the number of ones and zeros in the code must not differ by more than one. Second, the autocorrelation must not exceed $1/N$, where N is the length of the code, when not exactly aligned with itself. Finally, the PN sequence must have balanced runs, i.e., $1/2$ of the runs of consecutive similar digits are of length 1, $1/4$ length 2, $1/8$ length 3, and so on [2], [9].

Pseudorandom sequences are efficiently generated from linear feedback shift registers (LFSR). These LFSRs are a string of one-bit registers cascaded together with connections to binary adders (XOR gates) at predetermined positions. The connections to the XOR gates are known collectively as the tap configuration and are determined by a generator polynomial. The output of the tap configuration is feedback into the first register and the feedback loop continues until the appropriate number of bits are shifted out of the output, which is the last register [10].

The generic k -stage LFSR in Figure 7 has $Y(x)$ as the output PN sequence. The seed, $m(x)$, is k -bits long and initially loaded into the LFSR. The tap configuration, given by $h(x)$, is a k -bit binary string of coefficients that determine which switches are closed to contribute to the sum feedback into the register holding the least significant bit. Since the k -stage shift register has 2^k possible states, after 2^k-1 transitions all have been exhausted and the LFSR starts repeating states again. Therefore the output PN sequence would start repeating again with a maximum periodicity of 2^k-1 .

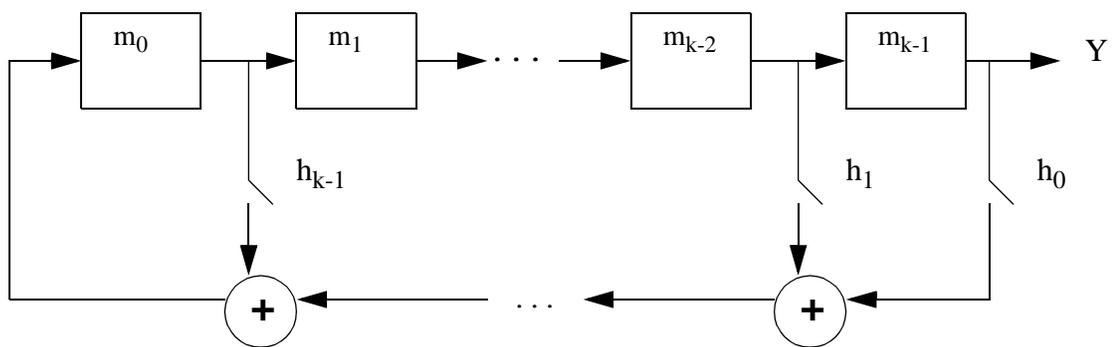


Figure 40: Linear Feedback Shift Register

Codes that repeat with maximum periodicity are maximal length (ML) codes and are desirable because they demonstrate the autocorrelation property described earlier. A code's periodicity is solely determined by the tap configuration as the choice of seed does not affect the length of the PN sequence as long as it is not all zeros [2], [8], [9]. The coefficients of the tap configuration, bit string $h(x)$, are determined experimentally. The ML tap configurations for a given length can be found by exhausting all the possibilities and checking the periodicity of the resulting codes. Tables for tap configurations that generate ML sequences can be found in [8].

To employ effective multiple access, the generated codes also must not interfere with each other and there must be enough codes to accommodate several users. For a k -stage LFSR, the number of ML sequences that can be generated is Euler's function divided by the LFSR length.

Equation 7:
$$ML_{num} = \frac{\phi(2^k - 1)}{k}$$

Euler's function gives the number of numbers that are coprime to, i.e., have no common factors with, and less than a certain number, including 1. Euler's function is maximized for prime numbers since all of the numbers less than it are coprime to it. Therefore, if $2^k - 1$ is a prime number, the corresponding k -stage LFSR will generate the maximum number of usable PN sequences [8].

However, these sequences must not be mistaken for each other if they are going to be a usable set. This is determined by the digital cross-correlation function.

Equation 8:
$$R_{xy} = \frac{1}{N} \sum_{i=1}^{N-1} x[n] \cdot y[n+i]$$

R_{xy} represents the cross-correlation between two codes, $x[n]$ and $y[n]$ [3], [11]. This is useful in examining the orthogonality of two codes, or the difficulty in mistaking one for the other.

As an example, consider two different LFSR tap configurations that use the same initial seed. They are chosen to be [0 0 1 1] and [1 0 0 1] and happen to be the only two configurations for a 4-stage LFSR that generate ML sequences. The resulting sequences for seed [1 0 0 0] are respectively,

0 0 0 1 0 0 1 1 0 1 0 1 1 1 1 and 0 0 0 1 1 1 1 0 1 0 1 1 0 0 1.

They meet the criteria for number of ones and zeros and runs listed above. Both have eight ones and seven zeros. Figure 8 shows that their autocorrelations only exceed 1/15 when they are exactly aligned with themselves. Finally, they both have balanced runs. Each has 8 runs of consecutive digits: four one-bit runs, two two-bit runs, a three-bit run, and a four-bit run. Their resulting autocorrelations are shown in the Figure 8 and are as close to an impulse function as a deterministic algorithm can be.

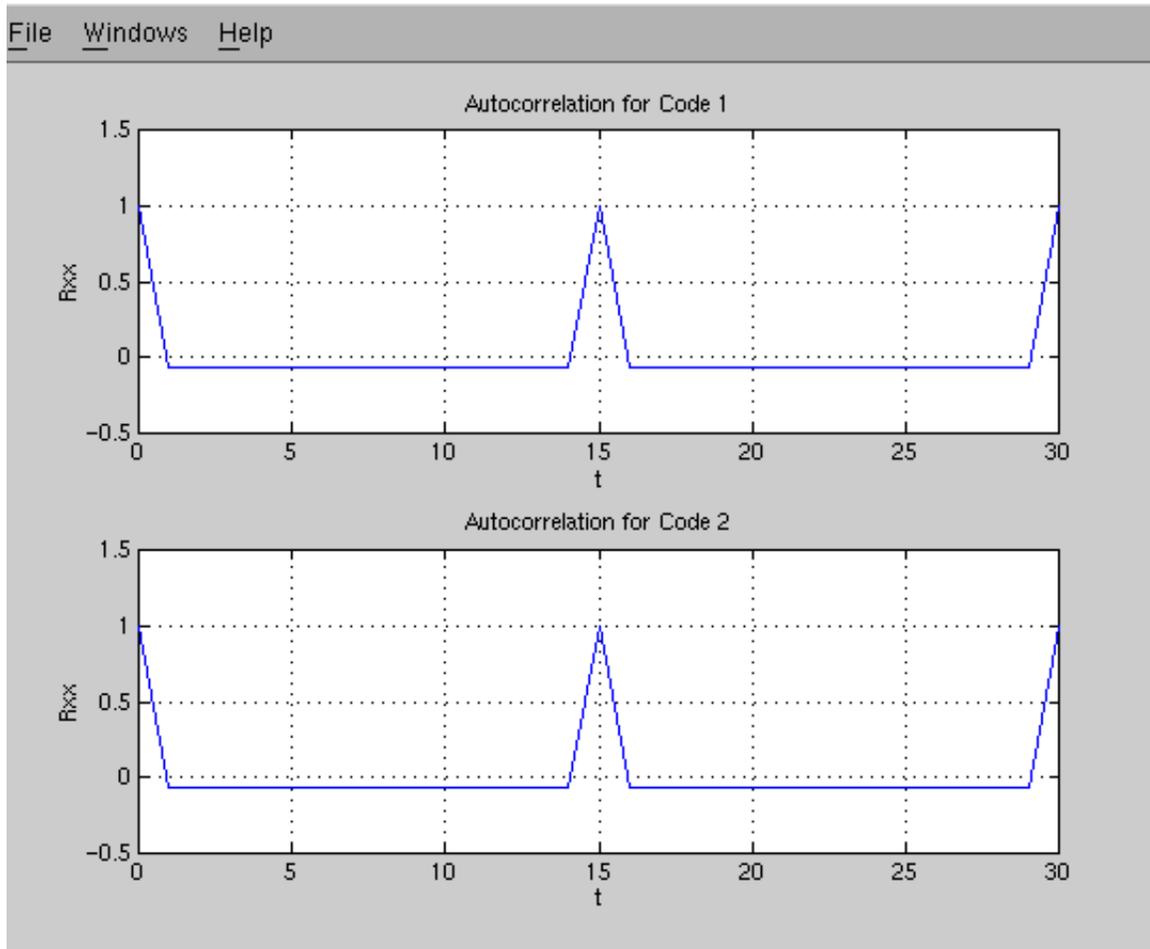


Figure 41: Autocorrelations

To use these two codes on the same carrier, they must be minimally correlated. The normalized cross-correlation is given in Equation 5 and graphed in Figure 9. If we are trying to send $[0\ 1\ 0\ 1]$ as data using these codes, this would result in Figures 10 and 11, respectively. Since the chip length is 15, the code spreads the data by 15, i.e.,

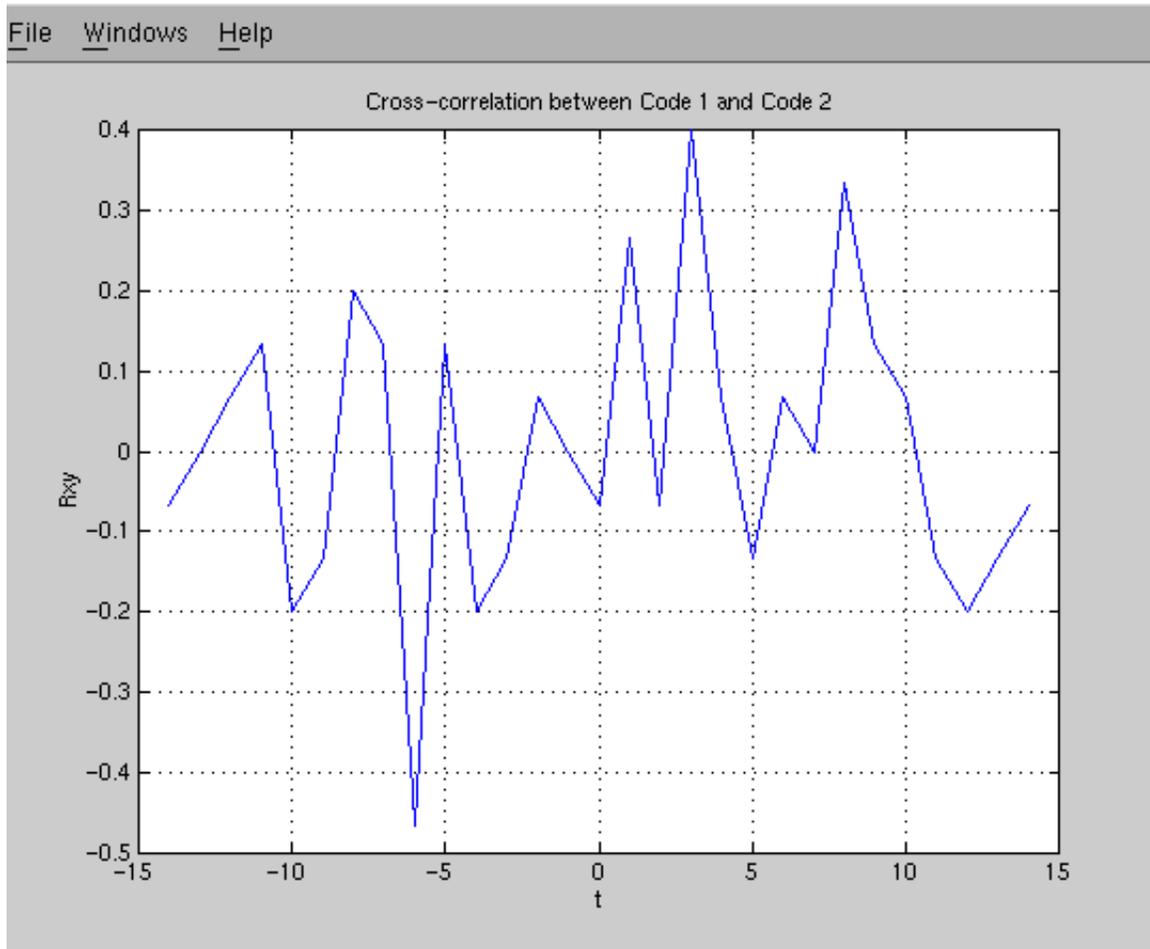


Figure 42: Cross-correlation between Code 1 and Code 2

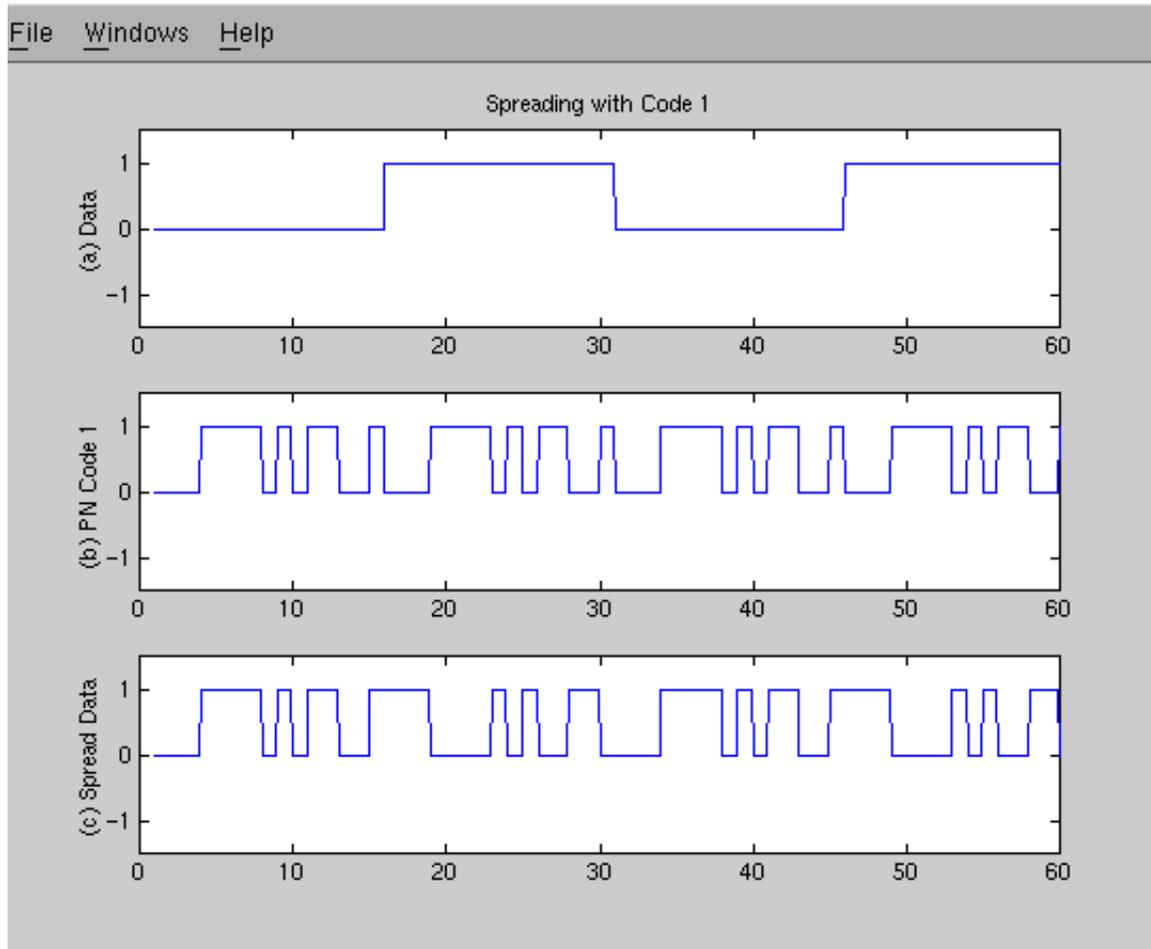


Figure 43: Spreading with Code 1

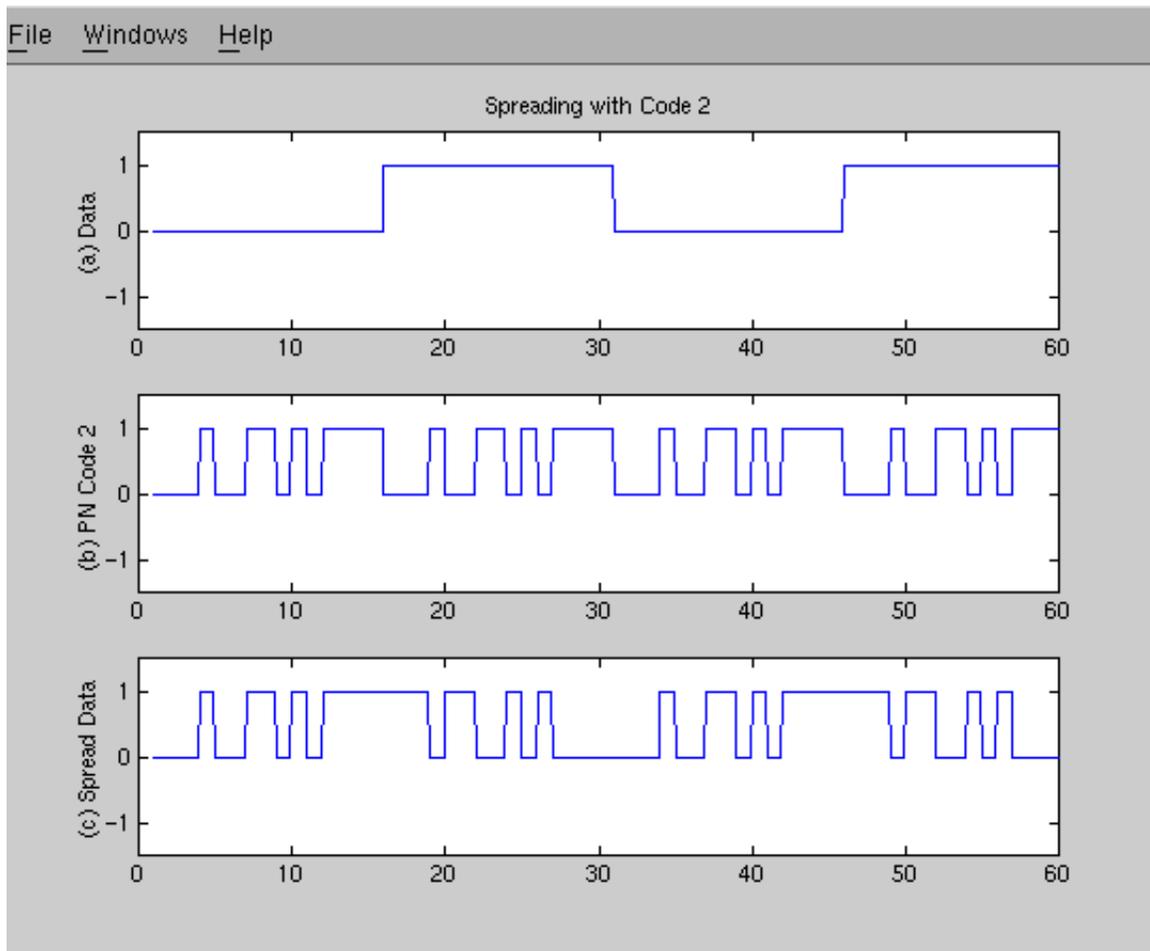


Figure 44: Spreading with Code 2

every bit is turned into 15 bits by the PN code. The original data can be recovered again by performing the XOR function again with the correct code, as shown in Figure 12. However, if the wrong PN code is used, the data is not recovered and the result looks like noise, as shown in Figure 13.

Figure 9 shows that the correlation between Codes 1 and 2 never reaches 50%. If one of the codes is compared with the incoming spread data stream and 8 or more

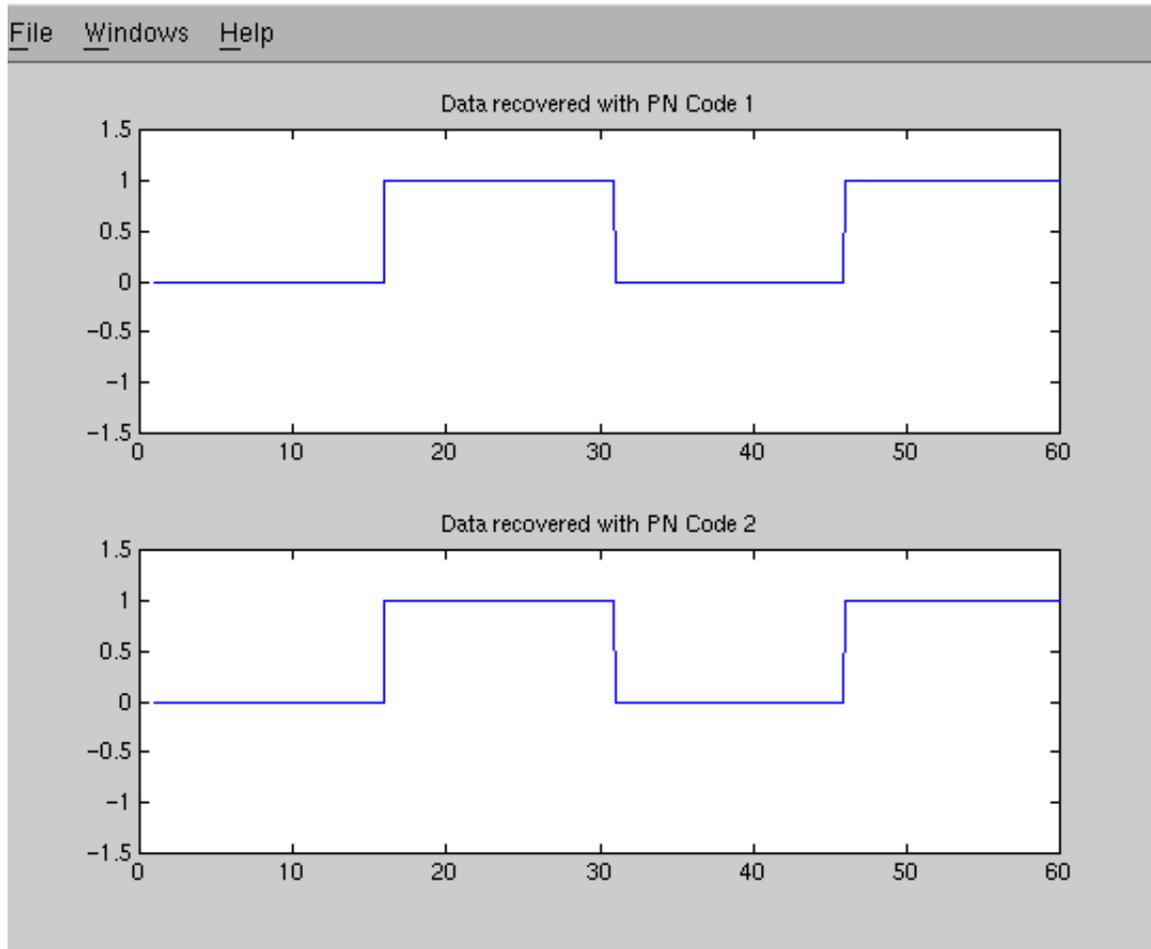


Figure 45: Properly Recovered Data

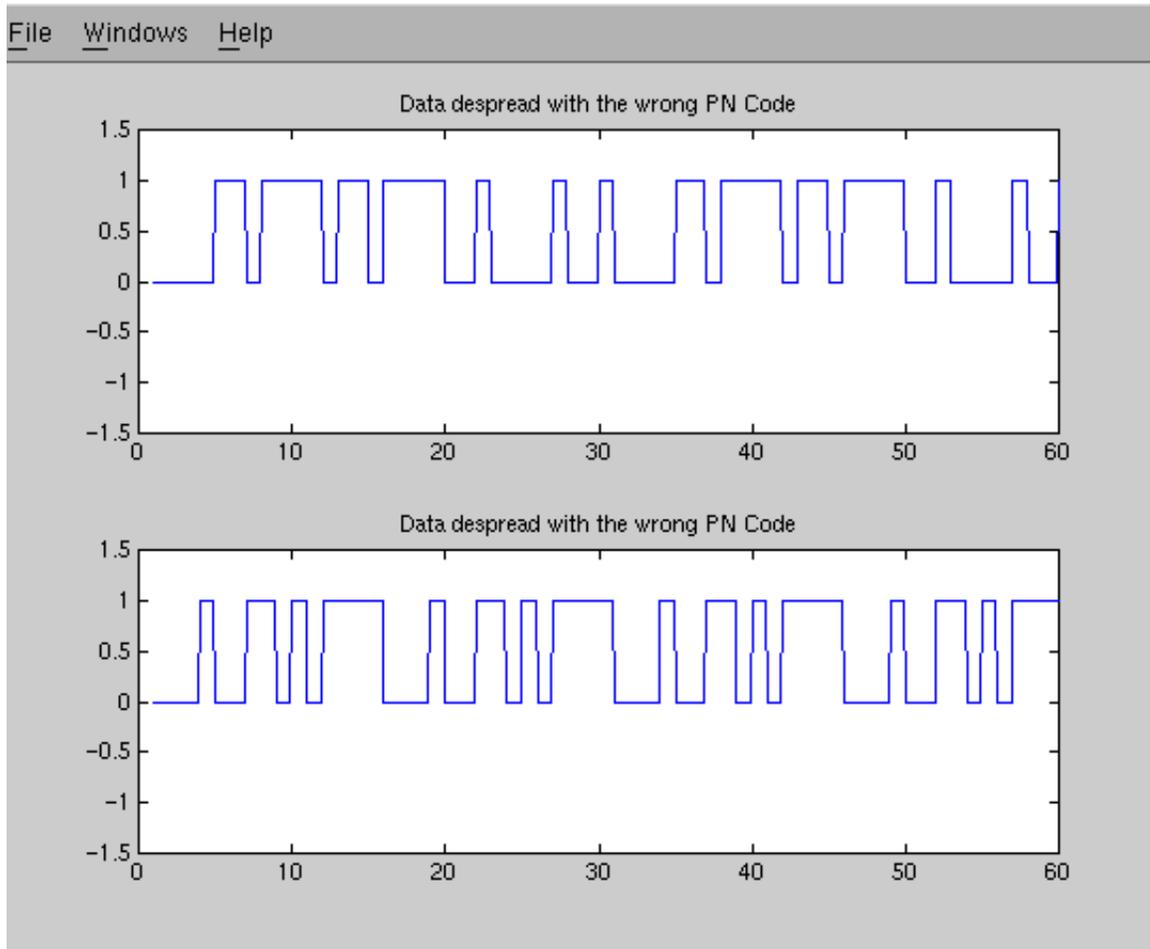


Figure 46: Improperly Decoded Data

of the bits in a window of 15 consecutive bits are positively or negatively correlated, then the chosen code is probably the correct code for despreading the incoming data.

Therefore, a threshold of 8 matches can be established for determining whether a code is being properly despread or not. If fewer than 8 bits are positively or negatively correlated, then either the wrong data is trying to be decoded for that PN code or more of the data needs to be acquired. Later, this will prove useful in the despreader module of the digital receiver for decoding the spread data without a synchronous clock.

It is possible to generate ML codes that are not the output of an LFSR but still have a low cross-correlation with other codes of the same set. A k -stage LFSR resulting in n -bit ML codes can only produce a subset of all the possible n -bit strings. Therefore as n and k increase, more and more sequences are available that comply with the randomness properties, but are not attainable with an LFSR. One way of generating these codes is by XORing two ML codes together. This results in an ML code known as a Gold code.

Some research focuses solely on how to generate more PN sequences of a given length. Lately, research has focused on using chaotic signals to generate new PN sequences. For a certain sequence length there are only a certain number, b , of ML sequences that can be generated. Gold codes create more sequences, but only b^2 at the most. For a 6-stage LFSR generating 63-bit codes, according to Equation 8, fewer than 6 ML codes can be generated. They yield at most 36 different possible Gold Codes. There are 2^{63} possible bit strings and many of them could be ML sequences as well.

The signals generated by a chaotic source inherently have minimal non-zero shift autocorrelation and generally have good cross-correlation properties. This is because a chaotic source is very sensitive to initial conditions and can produce a variety of outputs. Therefore, it intrinsically has a broad spectrum. Heidari-Bateni and McGillem first proposed and studied chaotic sequences generated by a logistics map [12], [13], [14], [15]. Adler and Rivlin used Chebyshev polynomials to generate PN sequences and Chen et al studied their performance [16], [17]. They found that the chaotic sequences slightly outperformed the Gold Codes in terms of error performance over signal to noise ratio.

They also found that chaotic sequences allowed significantly more users similar bit error rate compared to Gold Codes or conversely, that the bit error rate was improved for chaotic sequences over Gold Codes for the same number of users.

3.0 System Overview

The goal of this thesis is to design a digital receiver processor to reliably transfer sensor data to a PC monitor display. The digital receiver processor is integrated with a receiver RF front-end circuit (RF MicroDevice's RFMD 2945) to receive signals from a digital transmission processor (an ORNL-developed chip called ACQ2) and a transmitter RF front-end circuit (RF MicroDevice's RFMD 2510). The ACQ2 and 2510 chips generate a direct sequence spread spectrum signal for reception and decoding by the 2945 chip and the digital receiver. The sensor data link is shown in Figure 14.

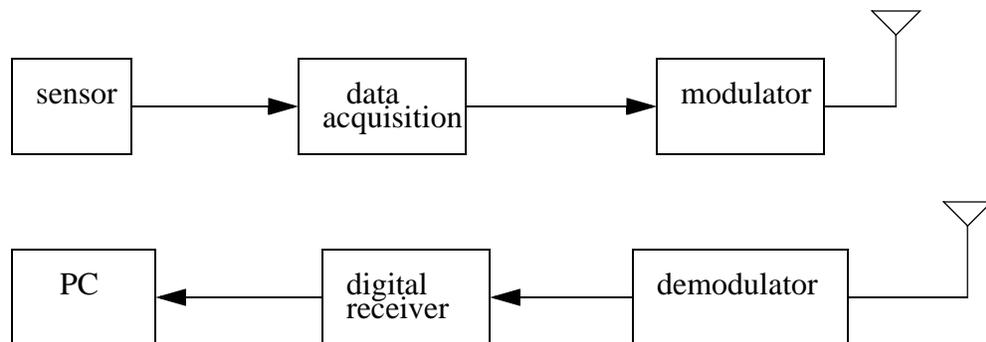


Figure 47: Sensor Data Link

3.1 Data Acquisition Chip

The ACQ2 chip was developed at the Oak Ridge National Laboratory to provide baseband digital data for wireless monitoring of mouse vital signs. It samples its four sensor inputs, as shown in Figure 15, and creates a serial data packet and produces a spread spectrum digital stream for wireless transmission in the digital controller. It employs a 10-bit successive approximation analog-to-digital converter (ADC) and a 2.5 V bandgap reference to digitize the sensor inputs. The digital controller is responsible for the front-end electronics, RAM, PN engine, packet builder, and spreading control. It also has a differential encoder to ensure a robust data stream and can choose from two maximal length sequence 63-chip PN codes as well as a 63-chip Gold code to ensure secure spread spectrum communication. Off-chip RF front-end circuitry can be put into sleep mode between sampling periods to save power.

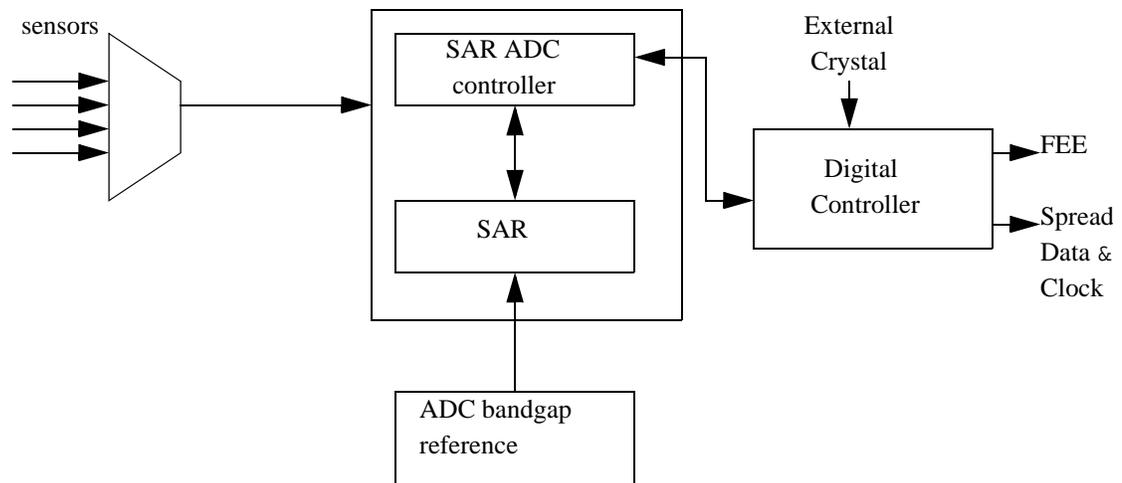


Figure 48: ACQ2 Block Diagram

3.2 Low-power Transmitter

The RFMD 2510 is a low-power wireless transmitter that can operate in the US 915 MHz band. It has an on-chip voltage-controlled oscillator (VCO) consisting of a phase detector and charge pump as well as a programmable phase-locked loop for frequency synthesis. The loop filter for the VCO is off-chip and included in the evaluation board in addition to the reference crystal needed. It also has power-down capability and only consumes 1 uA when in sleep mode. It is this low-power feature that made the 2510 attractive since wireless transmitters generally spend more time sleeping than transmitting. A basic diagram of the transmitter is given in Figure 16.

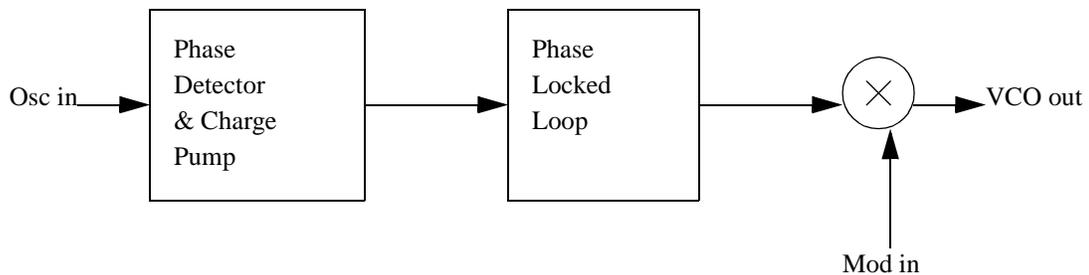


Figure 49: RFMD 2510 Block Diagram

[18]

3.3 FSK Receiver

The RFMD 2945 receiver converts an input RF signal into a digital output signal using a frequency modulated feedback demodulator. A block diagram is shown in Figure 17. The VCO output provides the RF carrier reference, which is mixed with the incoming RF signal. This tracks, through two filters, the incoming RF signal and holds it at the discriminator center frequency. If the input frequency falls below this carrier frequency then a 0 is output and if it is above, then a 1 is output [8], [19].

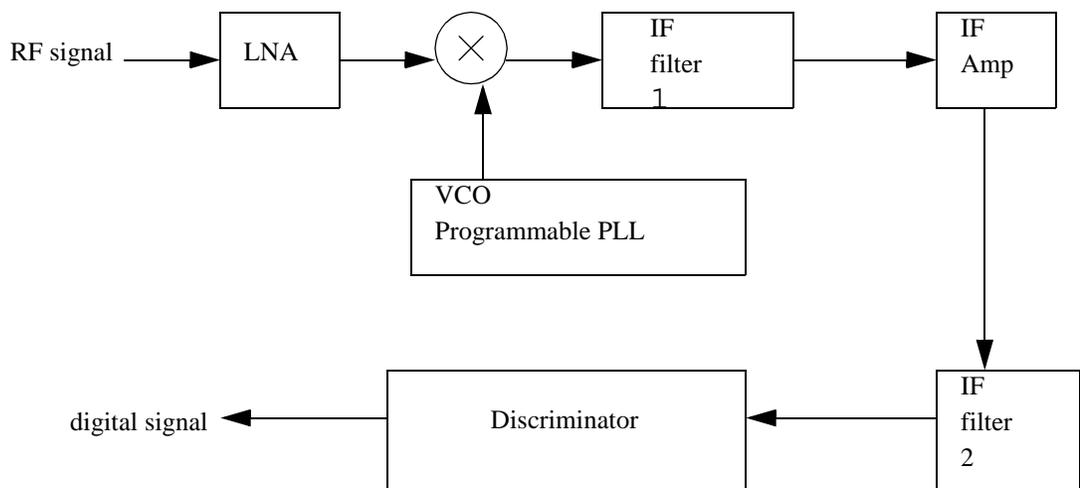


Figure 50: RFMD 2945 Block Diagram

3.4 Component Limitations

Multiple access was required to allow more than one sensor node to transmit data to a single digital receiver to be displayed on the PC. Since the ACQ chip has a PN engine in it and can build spread packets, originally, the system was to have multiple sensors chirping on the same frequency separated by different PN codes -- CDMA. However, due to the limitations of the RFMD 2945 receiver, this was not possible and the sensors instead had to be separated by different carrier frequencies -- FDMA. The 2945 chip only allows, as inputs, an RF signal and a carrier frequency. The 2945 demodulates frequency-shift keyed (FSK) data from the given carrier frequency leaving a digital signal. However, to strip out the PN code and recover the data for a given transmitter, the matched filter and correlation functions necessary to do this need to be performed before the RF signal is demodulated. Since this could not be done with the components chosen to provide the RF link, an FDMA scheme was employed instead.

Figure 18 shows the necessary components of a direct sequence spread spectrum receiver. The correlation needs to be calculated before the IF mixing and filtering, which is impossible given the constraint of the RFMD 2945 receiver. The PN reference code needs to be discovered in the RF signal before it is demodulated and brought down to baseband. The clock generation and synchronicity decision need to be made while demodulation is occurring and not segregated to a separate digital processor because at baseband multiple PN codes will concatenate and their information will be utterly unrecoverable.

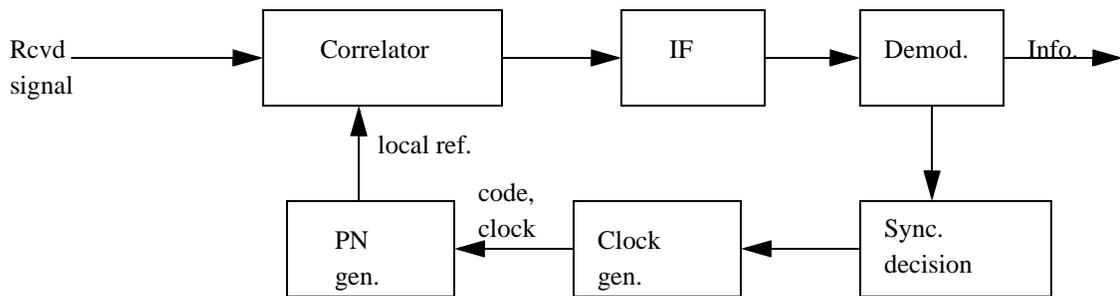


Figure 51: Direct-Sequence Spread Spectrum Receiver

Two transmitters were built and tested, Tx101 and Tx104. They consisted of an evaluation board for the 2510 chip and ACQ chip. Each transmits three data channels, a temperature channel, and a sequential packet counter. Each can use multiple length PN codes, but for this demonstration only 63-chip Gold codes were considered when building the digital receiver. In fact, since the transmitters were separated in frequency, one PN code was shared by both transmitters. Even though it did not provide multiple access, the PN code did provide data robustness and a measure of confidence in the fidelity of the received data. For display purposes only the temperature, the first two data channels and the sequential packet counter were shown on the PC. However, all data channels were analyzed and demonstrated reliable transmission. A LABVIEW program coordinated the results display on the PC.

4.0 Digital Receiver Architecture

Once the RF signal has been demodulated and the baseband digital signal has been recovered, the serial data stream is sent to the digital receiver for despreading and stacking for display on the PC. The digital receiver has no prior knowledge of the phase of the clocks used to generate the baseband digital data stream in the transmitter and so must deduce this information from the inbound data stream. Also, this clock recovery must be done in real time so that received digital data can continue to stream through the receiver. To present stacked parallel words to the PC from the inbound digital bit stream, the digital receiver must perform five main functions, separated as design modules: chip polarity decoding, despreading, protocol stripping, packet validation and packet processing. The partitioning of these functions is shown in Figure 19.

4.1 Clock Recovery and Chip Resolution

Clock recovery and chip resolution are performed by the first module, the polarity decoder. First, the polarity of each inbound data chip must be resolved. This first module takes the demodulated data as input (DEM0D) as well as a reference clock (SMPCLK). It uses an oversampling scheme to resolve the logic polarity of each chip (SPDA), derive the PN clock (DPNCLK), and align the PN clock.

Three consecutive five-sample windows of the input data are analyzed: early, middle, and late. The windows are compared to each other and the derived PN clock (DPNCLK) is dithered according to which one has the greatest magnitude. The

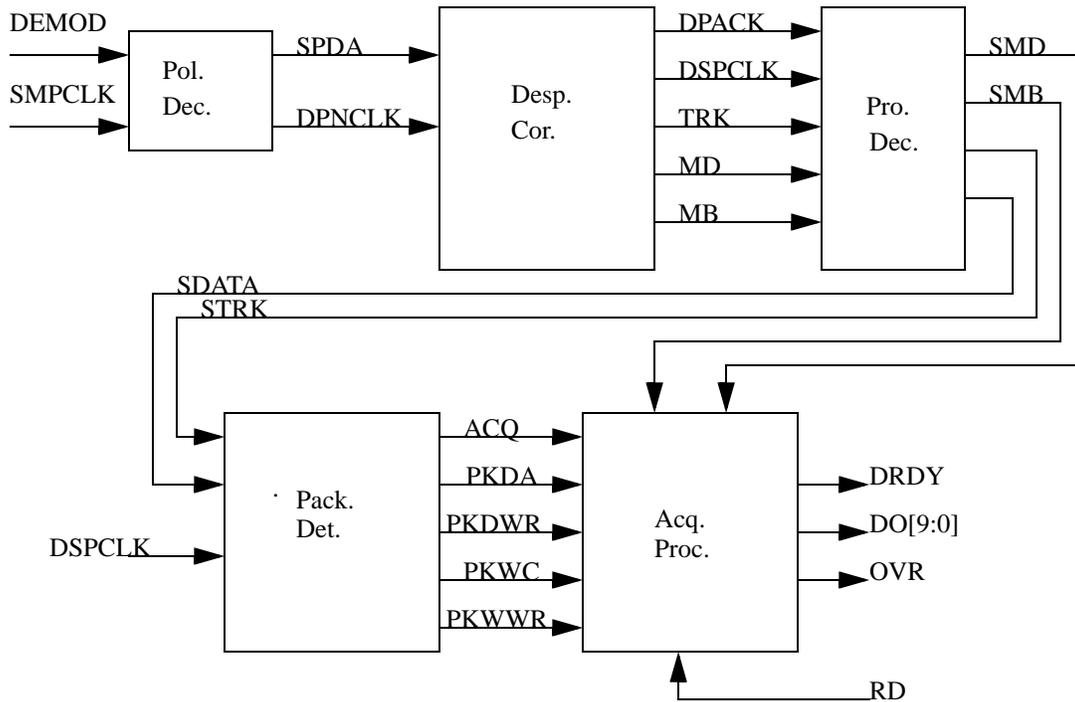


Figure 52: Digital Receiver Block Diagram

clock is dithered on the falling edge meaning that the time between the last falling edge and the next rising edge is always the same. The periodicity of the clock is determined by the falling edge. Figure 20 shows the polarity decoder in the default state so no dithering is performed. If the early window had the largest sampled magnitude of the three, the falling edge would be advanced one sample clock (**SMPCLK**) cycle as indicated by the dashed lines to the left of the default falling edge. If the late window were largest, the falling edge to the right of the default edge would be used to retard the clock.

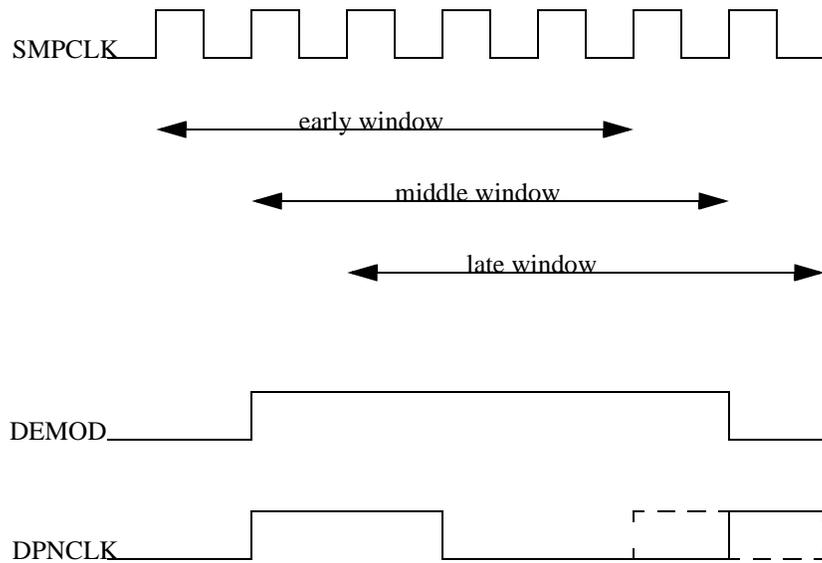


Figure 53: Polarity Decoder Dither Conditions

As shown in Figure 21, the polarity decoder consists of a 7-bit shift register to form the three consecutive oversampling windows, a comparator to decode the polarity of the oversampled bit and the dithering logic. The dithering logic compares the magnitude of the three windows and adjusts the clock generation and synchronization logic according to the method described above.

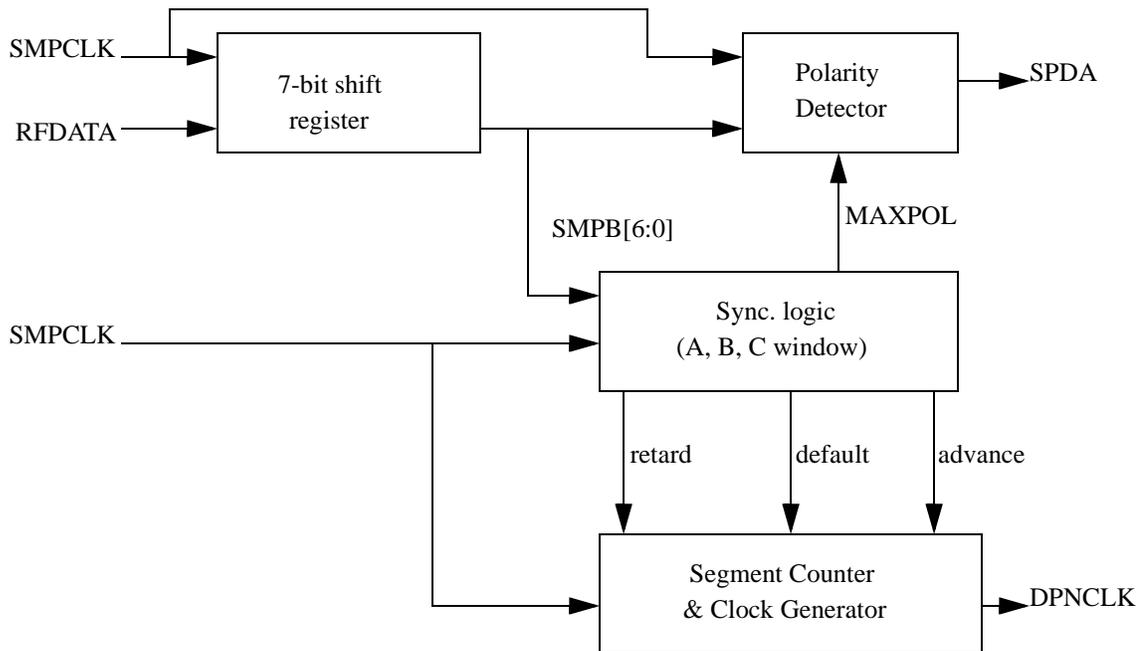


Figure 54: Polarity Decoder Block Diagram

4.2 Digital Despreader

The second module measures the correlation of the spread code to the inbound data, compares it to a stored reference code and determines the polarity of the despread data bit. Also, track mode is determined and parameters for establishing it and disengaging it are updated. The despread clock is derived and aligned with the data. The despreader is shown in Figure 22.

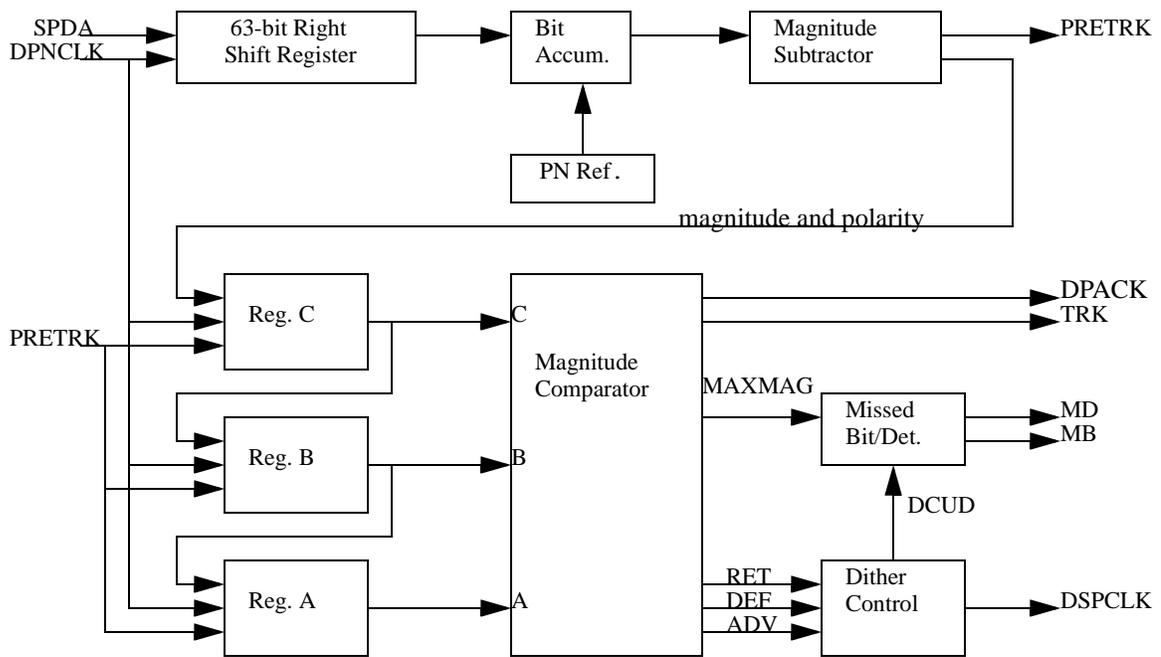


Figure 55: Despreader Correlator Block Diagram

Earlier, correlation was presented as normalized, that is, as a fraction of the maximum correlation. However, this correlation calculation requires the computationally-expensive operation of division and is not used in the implementation of the digital receiver, since it is not necessary. The earlier derivation provides a way of analyzing codes of various lengths, but since the length of the PN code for this design is known and fixed, scaling the correlation is unnecessary. As long as the comparison thresholds are understood to be valid for 63-bit PN codes, then the absolute correlation calculation without the normalization suffices.

The first job of the despreader correlator is to calculate the correlation of the inbound data stream with the reference PN code. This is done using three consecutive

63-bit windows of the inbound chips, a 63-bit right shift register and summing the bits in each window. Then, the inbound chip stream is analyzed using comparators to perform two functions: dithering the derived despread data clock (DSPCLK) and deriving the polarity of the serial despread data (SDATA).

Dithering the derived despread data clock is accomplished by comparing the correlation windows. Like the polarity decoder dithering in Figure 20, the rising edge of the derived PN clock always stays the same, relative to the last falling edge and the following falling edge is dithered according to the output of the sliding correlator. If the first window has the largest correlation, then the falling edge of DSPCLK occurs 1 clock cycle of DPNCLK earlier with respect to the last rising edge than it did the previous cycle. If the last window has the largest correlation, then the falling edge of DSPCLK occurs 1 clock cycle of DPNCLK later with respect to the last rising edge than it did the previous cycle. Otherwise, the middle window is assumed to have the largest correlation and the periodicity of the previous clock cycle is repeated.

The derivation of the polarity of the despread data involves analyzing the inbound chip stream for correlation with the stored PN reference and periodicity of the signal. The correlation dimension statistics determine the polarity of the current bit, while the temporal statistics indicate a measure of confidence of how well the inbound spread data is being despread. Figure 23 shows a histogram of how this works.

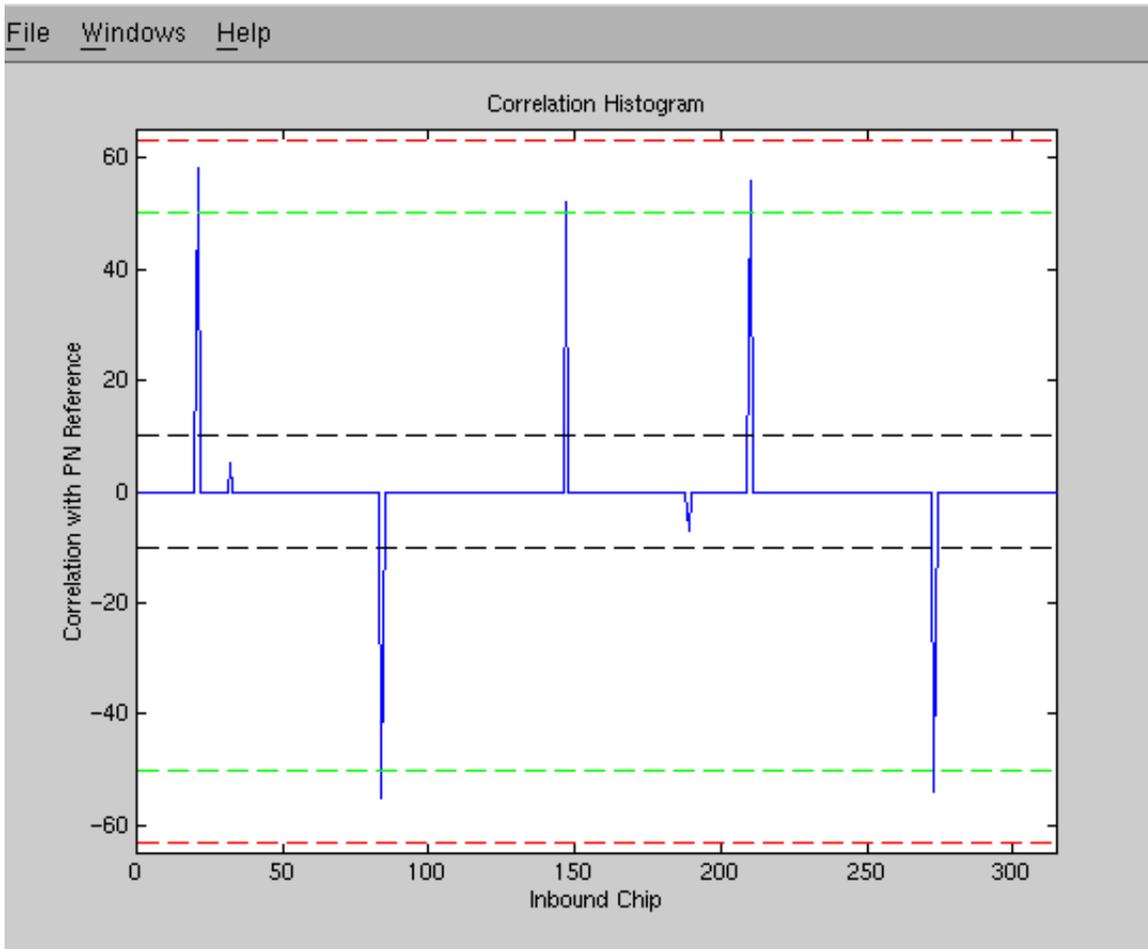


Figure 56: Despreader Correlator Histogram

The red, green, and black dashed lines represent thresholds for correlation decisions. They are calculated with respect to a baseline of 0, which would indicate that the inbound signal is not at all related to the stored reference. Positive values of correlation indicate that the data bit is a zero, since XOR with 0 is a transparent operation. Negative correlation values indicate that the data bit was a one since XOR with a one results in inversion.

The red dashed lines indicate the maximum positive and negative value of correlation possible for a 63-bit code. This occurs when the inbound chips exactly match the stored reference; and for this implementation, that value is 63. The green dashed lines indicate the detection threshold (DSMDTH[5:0]) set for the despreader correlator and are set in Figure 23 at 50 as an example. The black dashed lines indicate the bit threshold (DSMBTH[5:0]) set for the despreader correlator and are set in Figure 23 at 10 as an example.

If the absolute value of the correlation calculated lies above the detect threshold, the serial data is aligned closely enough with the reference PN code and the despread data bit polarity is declared. This is shown in Figure 23 as the blue peaks in correlation. The decoded data is 01001. If the absolute value of the calculated correlation lies between the bit and detection thresholds, the inbound chips are not yet correctly aligned with PN reference meaning there is a missing polarity detection but a data bit has been detected. If it falls below the bit threshold, it is assumed that the inbound data has no correlation with the PN reference and that neither a data bit nor a polarity detection has been detected. This is shown in the two blue peaks near the baseline. This can be caused by one of the chips becoming corrupted in the transmission. However, due to the robustness of the PN codes, this minor correlation is rejected and the data can still be correctly decoded.

Correlation statistics must be tracked temporally, that is, over consecutive PN chip length bit cycles (for this project, 63 bits) to determine how well the sliding correlator is performing. These statistics are used to declare the pre-track and track modes, which enable the downstream processing circuitry as well as provide the

acquisition processor information for determining the fidelity with which an entire packet of data was decoded. The number of consecutive detection cycles must exceed DSTKTH before track can be declared. The key is that the detection cycles need to repeat with a periodicity equal to the PN chip length. If consecutive failures to repeat exceed DSNTTH, then the track is disabled and establishing a new track must begin again. However, if for instance, there is only one cycle where the maximum correlation failed to repeat with the correct periodicity, then information indicating which thresholds it passed is sent to the acquisition processor, as the missed detect (MD) and missed bit (MB) statistics, and the track mode is kept enabled. The acquisition processor compares these accumulated values for a packet to a threshold (PKMDTH[5:0] and PKMBTH[5:0], respectively) and this contributes to the decision of whether the packet is considered good and cached or bad and discarded.

4.3 Embedded Protocol Removal

The third module removes the embedded communications protocol from the despread data stream. The embedded protocol is differential encoding calculated recursively as

Equation 9:
$$enc[k] = in[k] \oplus enc[k - 1]$$

where $enc[0] = 1$. To decode

Equation 10: $dec[k] = enc[k] \oplus enc[k - 1]$

where $enc[0] = 1$.

Equation 11: $dec[k] = in[k] \oplus enc[k - 1] \oplus enc[k - 1]$

Equation 12: $dec[k] = in[k]$

Therefore, the protocol remover is simply an exclusive OR gate with a clocked output. However, this is not just the encoder implemented in reverse. Unlike the encoder where the output is fed back to encode the next bit, the decoder uses a sliding window of two incoming encoded bits to determine the next decoded bit instead of feeding back the last decoded bit.

4.4 Packet Detector

Now that the incident serial chip stream has matched the stored pseudorandom reference to the satisfaction of the first two modules and the embedded communications protocol has been removed, the despread bit stream is assumed to be a serial data packet stream built in the manner configured in the data acquisition chip (ACQ2). The fourth module validates the preamble words at the beginning of the despread serial packet, measures the acquisition mode parameters, converts the serial

words to parallel, and counts the number of words in each packet. For the Graviton proof-of-concept demonstration the 8-word packet followed the form of Table 1.

The first two words, RF sync and Frame sync, are the preamble words and are stripped out as the packet detector converts the serial data to 10-bit parallel words.

Once these two words have been found, the packet detector skips the sequential counter,

Table 1: Packet Format

Word #	Function	Hex Value
1	RF Sync.	333
2	Frame Sync.	01F
3	Seq. Counter	001
4	Tx ID	005
5	Temp.	1FD
6	Data 1	014
7	Data 2	019
8	Data 3	02F

and validates the transmitter identification number, word four. If the two sync words and the identification number all match, then the packet detector enables the acquire mode and sends the now 6-word packet (PKDA[9:0]) to the acquisition processor with a data strobe (PKDWR). The packet detector also counts the number of words in the packet (PKWC[5:0]) and sends that to the acquisition processor as well with a word count strobe (PKWWR).

The values of the last six words in Table 1 are given only as examples and represent the pinned-down settings used to test the digital receiver. The sequential counter

increments between 1 and $3FF_{\text{hex}}$ and indicates the order in which the packets were sent. The transmitter identification can be used to rotate through different transmitters and identify which data came from which transmitter. In a true CDMA scheme this would be helpful, but not so much so for an FDMA setup, so those values were pinned. The temperature channel ranges from 40°F to 120°F and is sensed on the ACQ chip. The other three data channels range from 0 to 2.5 V with a precision of about 0.002 V. For ease of testing, all four data channels were pinned to the values given in Table 1.

The Packet Detector is shown in Figure 24. The serial data from the protocol decoder is converted to 10-bit parallel words. The first two words of the packet are verified and the third is skipped so that the fourth, the identification number can be

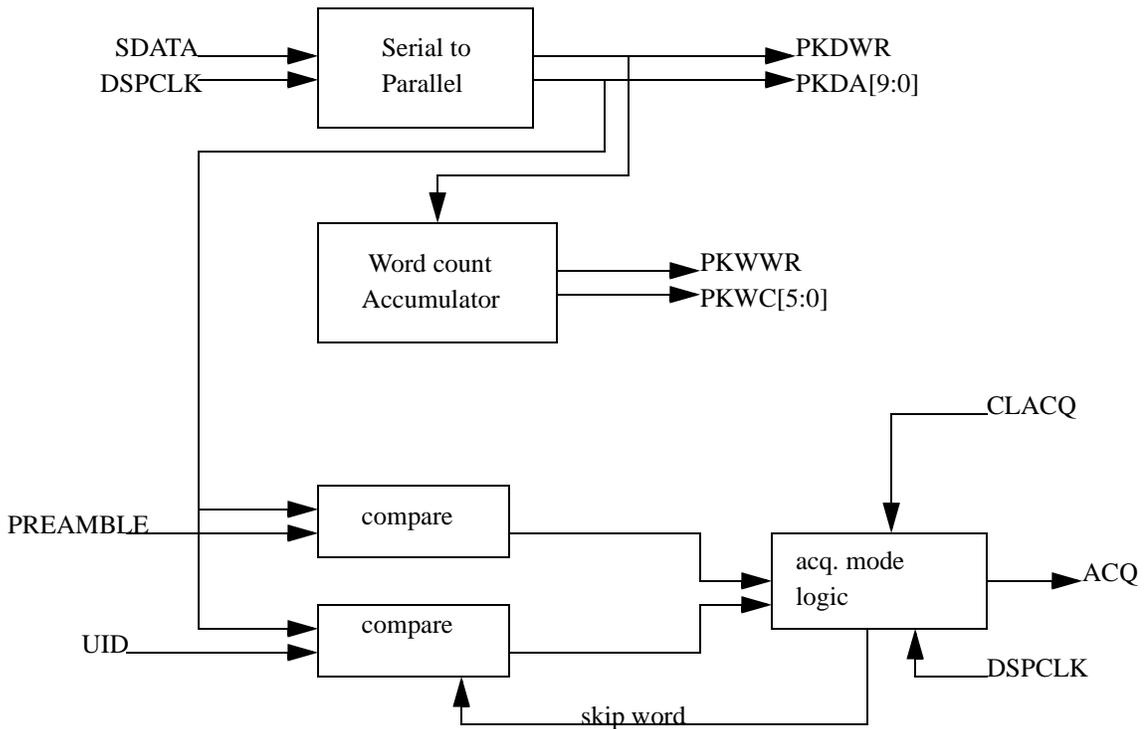


Figure 57: Packet Detector Block Diagram

verified. A word count is generated for the acquisition processor as well as strobes for the parallel data and the packet word count.

4.5 Acquisition Processor

The fifth module validates the packet by evaluating the flags generated in the previous modules. These flags include number of words in the packet and number of missed bits and detects. This module is also responsible for stacking the good data and interfacing with the host. The acquisition processor uses two FIFOs to store the inbound data. The first is a temporary FIFO to hold all the words in a packet together as the packet error logic validates the packet. The packet error logic will throw out a packet if it has too few words or too many missed bits or missed detects from the despreaders. If the packet is deemed good, it is transferred to the data FIFO, otherwise, it is cleared from the temporary FIFO. In either case the acquisition mode (ACQ) is cleared and the processor waits for the next packet. If the data FIFO is full, then the acquisition processor alerts the host PC via the data ready signal (DRDY) and awaits a clock from the PC to clock the data out of the FIFO to the output port pins. The data FIFO holds 16 6-word 10 bits/word packets when it is full. The acquisition processor is shown in Figure 25.

When the acquisition processor is transitioning a good packet from the temporary FIFO to the data FIFO, it cannot accept an incoming packet. For this project, the time between packets is about 3-5 seconds, which is fast for telesensing. Usually the

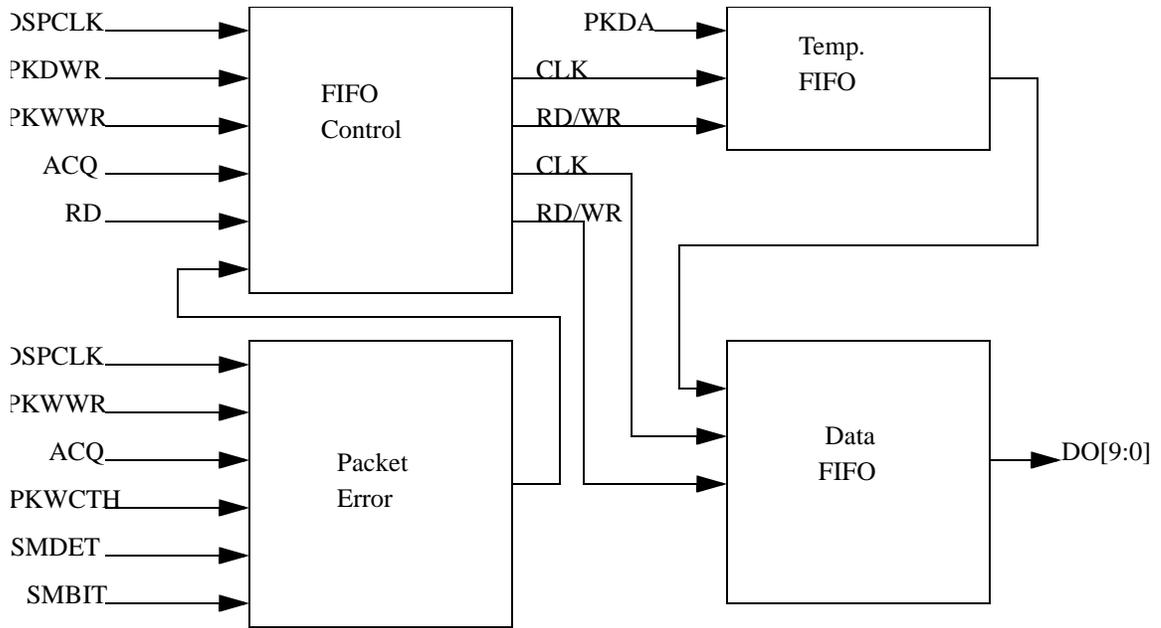


Figure 58: Acquisition Processor Block Diagram

sensor readouts need only be updated a few times an hour. This limitation helped simplify the processing logic and since there is no threat of a packet arriving while the previous is being processed, it is an appropriate design trade-off.

5.0 Implementation

The digital receiver was implemented in an Altera FLEX 10K50RC240-3 Field Programmable Gate Array (FPGA) using the Altera Max+Plus II version 9.3 software. Modules 1 and 2, the polarity decoder and the despreaders correlator, respectively were written in a different VHDL environment and so had to be converted to the Max+Plus II environment, along with their simulation stimuli. The changes to the code were mostly semantic changes and did not affect the structure or functionality of the code.

All of the modules were written in VHDL except for the data cache FIFO and the temporary packet FIFO, which were implemented using Altera's Library of Parameterized Modules (LPM). LPMs are technology-independent modules that conform to industry-wide conventions for implementing common functions in gate arrays [20]. The LPMs used were sections of the chip optimized to implement data storage functions. The synthesis tool had control over the physical placement of the design, but with some limitation. In order to ensure accuracy in the polarity decoder's sample clock (SMPCLK) and the derived PN clock (DPNCLK), these two clocks were placed on the device's two clock trees. Also, each module was assigned as a clique, which meant that the fitting tool would route each individual module in a physically confined area.

The thresholds used to determine the dithering conditions were brought off-chip to programmable pins on the test board along with other programmable thresholds. This allowed flexibility in determining the optimal settings for reliable reception. A LABVIEW program was written to clock the data out of the cache FIFO

when it was ready and display it on the screen as graphs of individual data channels. A digital receiver test board was built that allowed programming of the dithering thresholds to test the device. It had a socketed Electrically Erasable Programmable Read-Only Memory (EEPROM) so that the design could be iterated without removing the device.

6.0 Testing and Results

Initially, the design did not fit into the device chosen, meaning that the compiler could not find a way to implement the design given the resources of the device specified. Originally, it contained a monitor multiplexer to allow analysis of internal signals as testing was performed. This was taken out and the design fit. The device could be reprogrammed to bring different internal signals to the output pins as necessary, and while this made analysis slower, it significantly reduced the amount of resources the design required and allowed fit into the originally chosen part.

The design was tested both cabled and wirelessly. The cabled test was performed with 40 dB of signal loss and allowed initial functional testing of the device without interference or multipath so that the primitive functionality of the device could be confirmed. Wireless testing was then performed once basic functionality was confirmed. Interference from other RF transmitters was not a concern since the building in which the testing was performed acted as an RF shield. The threshold settings were not immediately optimized: functional settings were found and the testing was performed holding these constant. Figure 26 shows the results. Occasionally, packet displayed on the PC would be incorrect. This was considered a data drop out and its frequency is shown in Table 2. The results in Table 2 were taken over the 1024-sample window shown in Figure 26 and are typical for each device.

The cause of the data dropouts was traced back to the downstream processing of the despread serial packet. The data (SPDA[9:0]) out of the packet

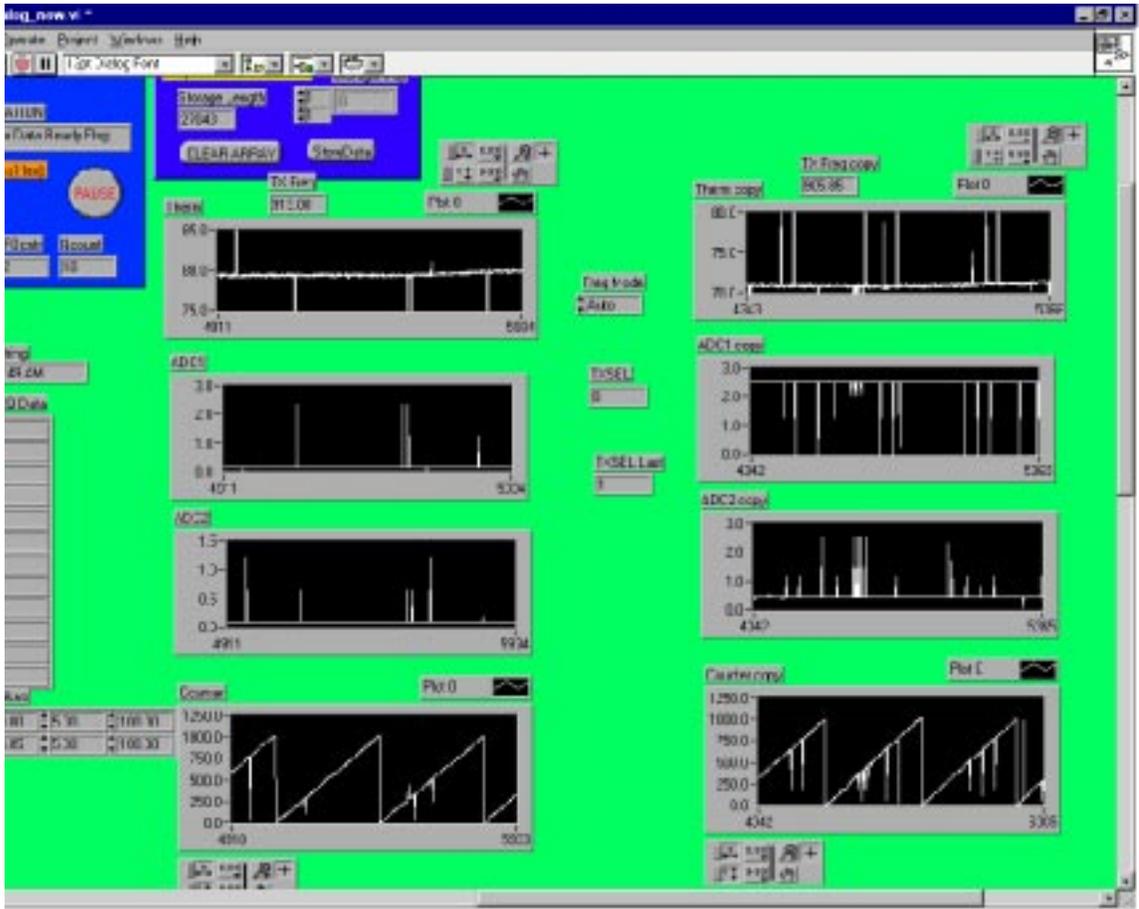


Figure 59: Initial Wireless Testing

Table 2: Data Acquisition Error Rate

Channel	Tx101	Tx104
Temp.	5.87E-3	1.466E-2
Data 1	5.87E-3	4.564E-2
Data 2	5.87E-3	1.662E-2
Counter	4.89E-3	1.075E-2

detection module going into the temporary storage buffer was not resembling the serial data out of the protocol remover. It worsened the more the despread clock (DSPCLK) had to dither. Therefore, the incorrect data was probably due to incorrect values being latched into the serial to parallel converter in the packet detector module as a result of the dithered clock being routed inefficiently on the chip. As a result of the limitation of resources internal to the device, DSPCLK could not be put on a clock tree and had to be routed as a regular signal.

The test environment was shielded from outside RF signals, but included several nearby solid objects. These objects could reflect RF waves causing interference at the receiver, a phenomenon known as multipath. This also degrades error rate performance, but is not something that could have been easily measured. Therefore, the effect of multipath on the error performance has not been directly quantized. These two real world implementation issues, internal clock routing and RF multipath, probably account for most of the data errors and explain why the system demonstrated worse error performance than is generally predicted by digital communication theory.

The transmitters were also tested for range. They were able to range 16 yards non-line-of-sight and still the receiver was able to decode the information. This setup was not tested for packet or word error rate. This error rate testing was performed at ranges of 6 to 10 feet.

The transmitter and the receiver were both repackaged for portability and demonstration purposes. The repackaged transmitter is shown in Figures 27 and 28 and the repackaged receiver is shown in Figures 29, 30 and 31. The repackaging of the transmitter caused shielding problems that degraded performance. The design was



Figure 60: Repackaged Wireless Transmitter

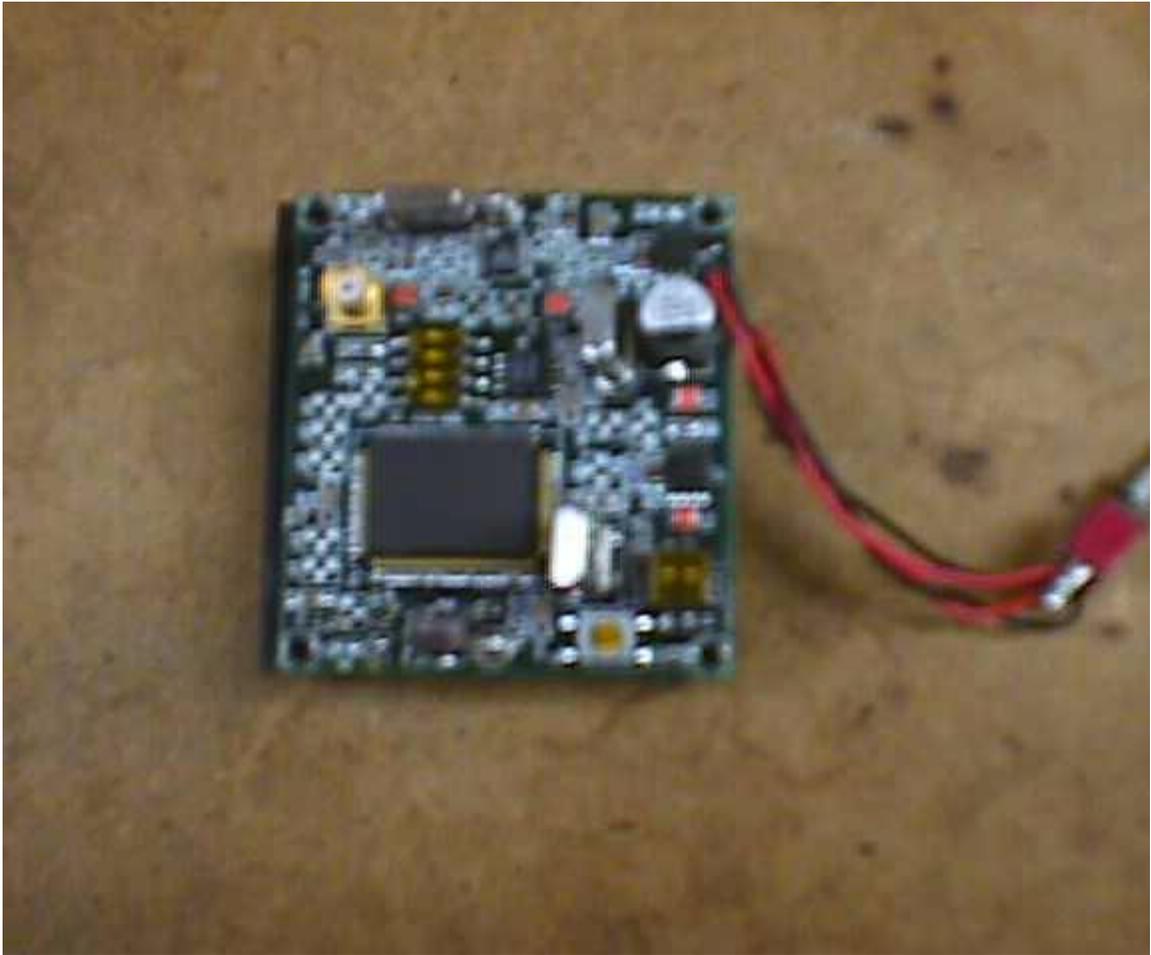


Figure 61: Internal Circuitry of Repackaged Wireless Transmitter



Figure 62: Repackaged Wireless Receiver

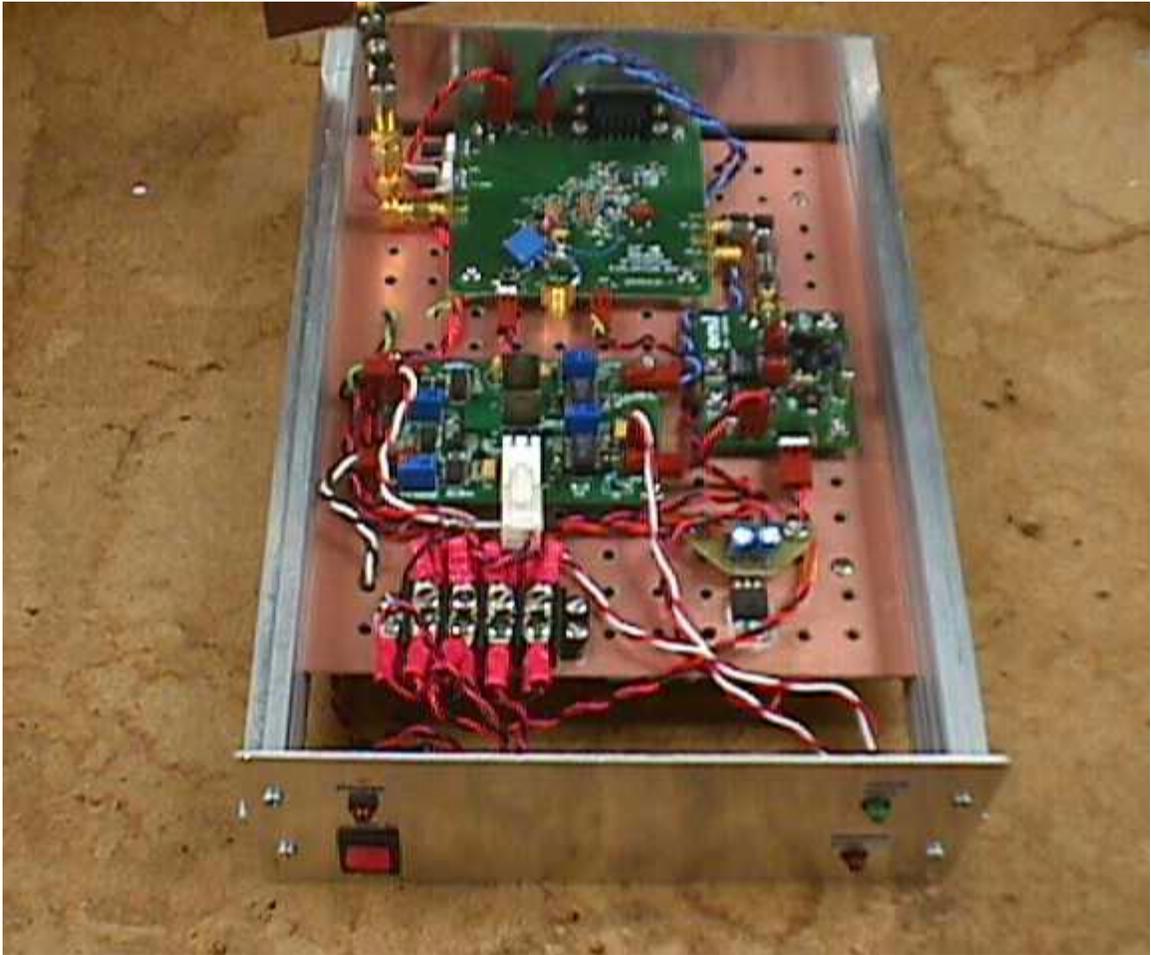


Figure 63: RF Front-End Circuitry of Repackaged Wireless Receiver

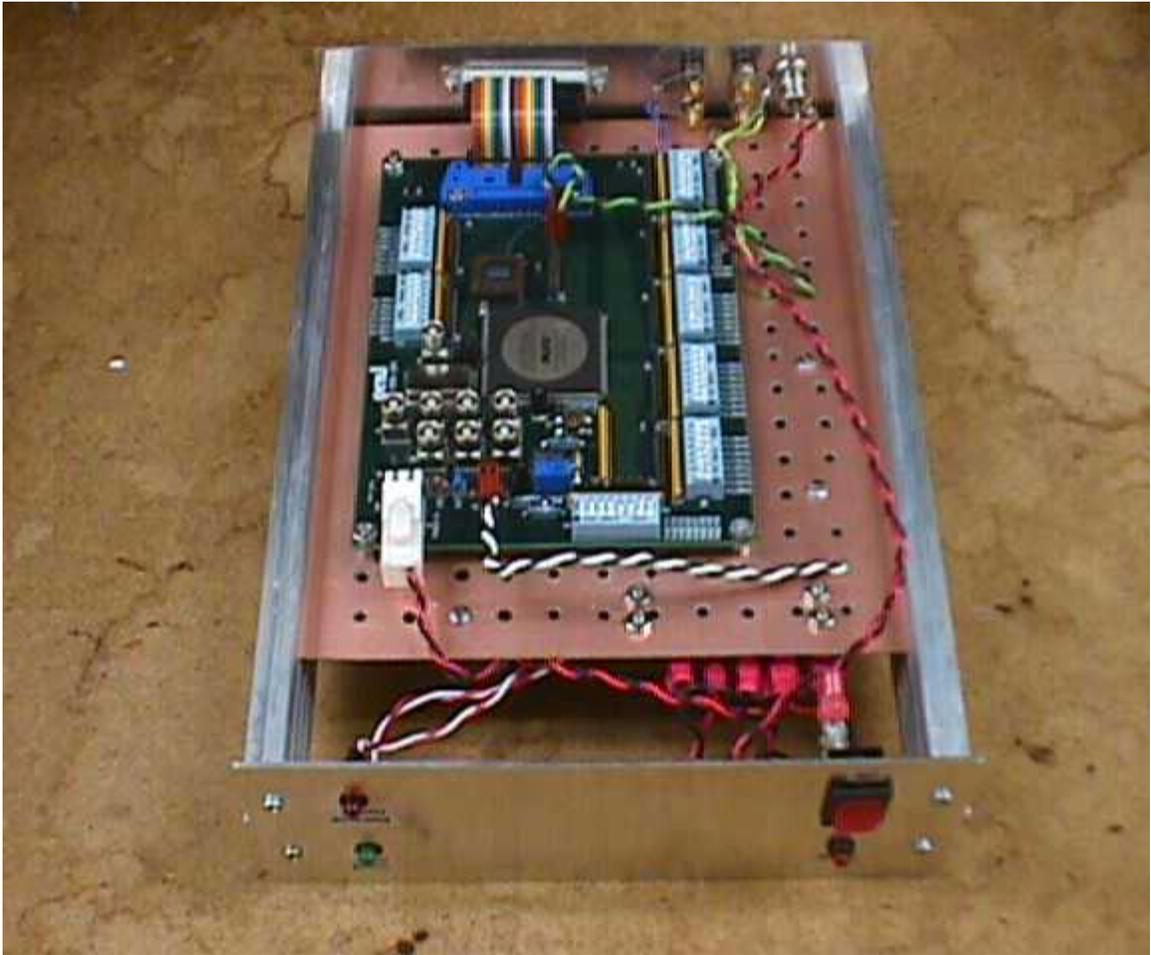


Figure 64: Digital Spread Spectrum Baseband Processing Board

iterated to try to alleviate the DSPCLK distribution problem. Instead of clocking the logic in the downstream modules on the falling edge of DSPCLK, the edge that is being dithered, the logic was clocked on the rising edge. This helped some but not much and only vindicated the notion that the clock distribution problem needs to be addressed in an ASIC environment where clock routing is well-defined or in a programmable part with more clock trees to ensure the fidelity of the clock signal.

To further optimize the design, the thresholds were optimized and entered into the design as constants to free up logic cells for more efficient routing. The transfer between the data storage FIFO and the temporary packet FIFO was cleaned up and simplified. The wired and wireless results are shown in Figures 32 and 33, respectively. They do not show performance comparable to the initial testing, but provide the design in a compact package that is portable.



Figure 65: Repackaged Wired Testing

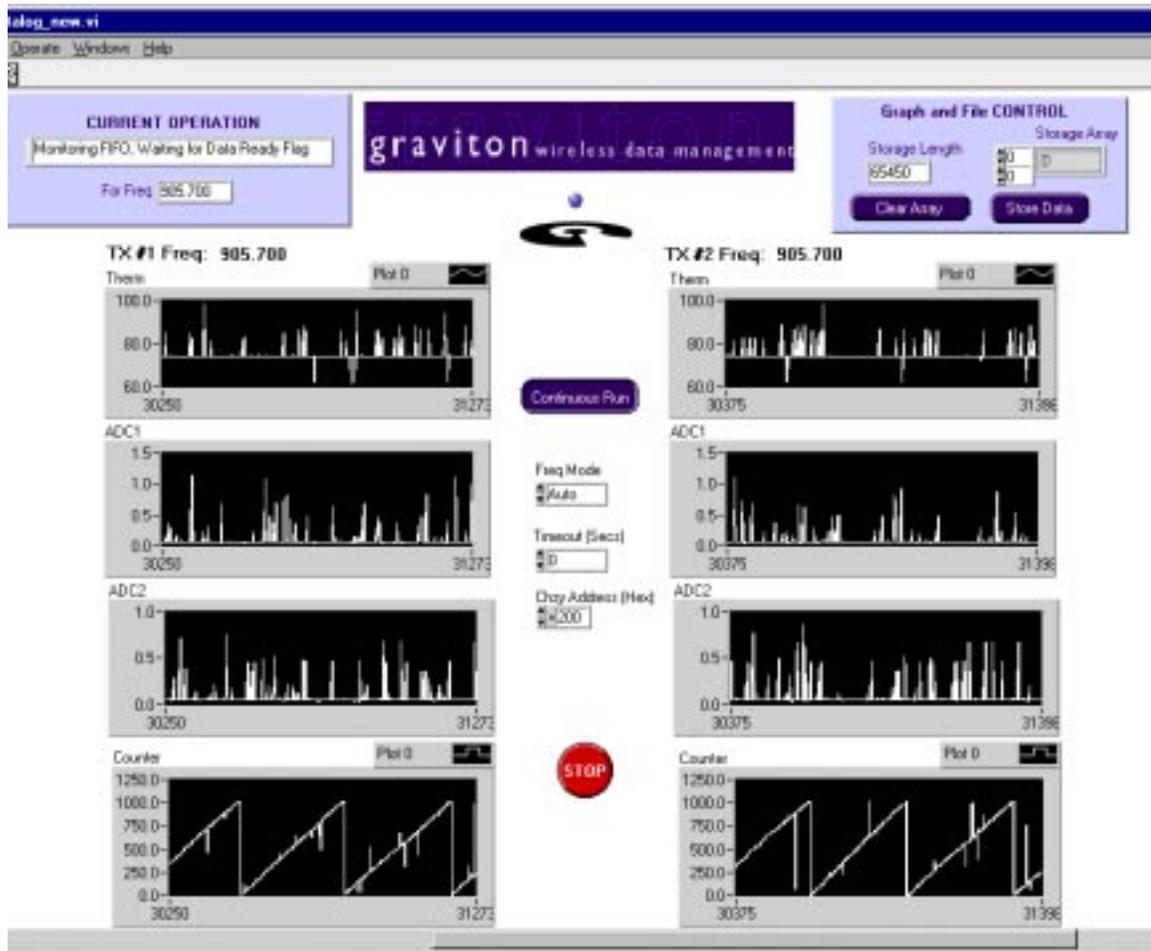


Figure 66: Repackaged Wireless Testing

7.0 Conclusions and Future Work

The monitor multiplexer was unnecessary. It was included in the original design as a carryover from a design in the ASIC environment where it was useful in analyzing devices whose designs could not be easily iterated. This was not necessary for an FPGA implementation since the device can be easily reprogrammed to bring out any necessary signals. Additionally, setting the programmable thresholds as constants in the VHDL code helped free up logic cells. While changing these constants is significantly slower than changing dip switches on a test board, the free logic cells allowed easier and more efficient routing of the design. This was especially important since the design was done in VHDL and not hand-placed schematic capture, which would have allowed the compiler less flexibility in fitting the design.

There were not enough clock trees on the Altera part, which meant that the third clock (DSPCLK) had to be routed as a regular clock signal. This proved costly since this clock was already being dithered and occasionally caused problems in the downstream modules latching data correctly. Some of the data lines may have been violating setup and hold times, but this was hard to determine due to the aperiodic nature of a dithered clock routed inefficiently. This scenario is possible but not as likely as data lines in the same parallel word being latched at different times creating erroneous outputs. This is likely since the problem became much worse with the wireless testing as the clock had to do more dithering.

There were some routing problems with the software tool trying to optimize the VHDL conversion by using internal chip resources called embedded array

blocks (EAB). These blocks were physically in the middle of the chips floor plan and occasionally interfered with requiring each individual module to be routed tightly together. For instance, a module may use an adder and be implemented in one corner of the chip, except for that adder, which is placed halfway across the chip in the EAB. This may not have been the main routing problem, but did not help keep the design close together. This problem can also be solved by the three reasons given above for the next generation design.

These were limitations of the part chosen, but will not be a problem in future generations of the digital receiver design for three reasons. First, the next-generation is being composed of fundamental modules that are being physically hand-placed in the device. Second, the choice of FPGA has changed to a device that has more clock trees and will ensure a tighter design with more predictable clock delays. Third, once the system specifications are decided for the wireless system, the digital receiver can be implemented in an ASIC with well-defined timing for clocks.

Sheng and Broderon's book on low-power wideband CDMA helped illuminate the problems encountered on clock distribution. They designed a system to support asymmetrical multipoint transceivers transmitting to a centralized base station receiver. The book focuses mainly on the downlink (base station receiving from the transceivers) design and shows how to build a CMOS implementation of the transmitter baseband modulator, RF transmitter, RF analog receiver, and the baseband digital signal processor (DSP). They used a matched filter correlator for both the I and Q channels but the DSP is segmented from the RF front end as in this system. They paid a significant amount of attention to clock buffering. They minimized skew by balancing the capacitive

load seen by the buffers. With a knowledge of the process parameters, transistors were sized appropriately and predictable clock distribution and propagation resulted [21].

The digital spread spectrum receiver worked well. The receiver was able to demonstrate reliable wireless spread spectrum transmission across a room. The initial testing showed low error rate for the unpackaged design and was able to show Graviton, Inc. how spread spectrum telesensing can be achieved. Repackaging made the receiver look more like a viable product, but exacerbated some problems the receiver was having with timing. However, these are issues that will be addressed and remedied in the next generation. The current digital receiver was not able to demonstrate CDMA, but this was due to component limitations in the rest of the system and not a result of a flaw in the design. However, the receiver did demonstrate the fundamental digital circuitry needed to implement CDMA and its reliable reception of sensor data proves that it will be a fundamental building block of the next-generation design.

CDMA communication requires intricate interface between complex digital and analog functions. Therefore, to implement a CDMA receiver both have to be considered. The digital portion of the receiver must be able to adjust the analog front-end circuitry responsible for demodulating the incident RF signal. Also, the timing and distribution of the clocks must be tightly controlled in the digital portion of the receiver has this has a tremendous effect on the data being recovered by the receiver. With CDMA's sustained popularity and wide range of design topics to explore, it will continue to be a widely-researched topic.

References

References

- [1] Britton, C. L. et al. "Battery-powered, wireless, MEMS sensors for high-sensitivity chemical and biological sensing." Presented at *The 1999 Conference on Advanced Research in VLSI*. Atlanta, GA. March 1999.
- [2] Sklar, Bernard. *Digital Communications: Fundamentals and Applications*. New Jersey: Prentice-Hall, 1988.
- [3] Proakis, J. and M. Salehi. *Communication Systems Engineering*. New Jersey: Prentice Hall, 1994.
- [4] Poor, H. and G. Wornell. *Wireless Communications*. New Jersey: Prentice-Hall, 1998.
- [5] Glisic, S. and P. Lappanen. *Wireless Communications: TDMA vs. CDMA*. Boston: Kluwer, 1997.
- [6] Rohde, U., J. Whitaker and T. Bucher. *Communications Receivers: Principles and Design, Second Edition*. New York: McGraw-Hill, 1996.
- [7] Mehrotra, A. *GSM System Engineering*. Boston: Artech, 1997.
- [8] Dixon, Robert. *Spread Spectrum Systems with Commercial Applications, Third Edition*. New York: Wiley and Sons, 1994.
- [9] Viterbi, A. *CDMA: Principles of Spread Spectrum*. Reading, MA: Addison-Wesley, 1995.
- [10] Peterson, W. and E. Weldon. *Error-correcting Codes*. Cambridge: MIT Press, 1972.
- [11] Yang, S. *CDMA RF System Engineering*. Boston: Artech, 1998.
- [12] Heidari-Bateni and C. McGillem. "Chaotic sequences for spread spectrum: An alternative to PN-sequences," *Proceedings of the IEEE International Conference on Selected Topics in Wireless Communications*. p. 437-440, 1992.
- [13] Heidari-Bateni and C. McGillem. "A chaotic direct-sequence spread spectrum communication system," *IEEE Transactions on Communication*. vol. 422. num. 2/3/4, p. 1524-1527, 1994.
- [14] Chen, C. et al. "Design of chaotic spread spectrum sequences using ergodic theory."

- [15] Shaw, R. "Strange Attractors, Chaotic Behavior, and Information Flow." *Zeitschrift fur Naturforschung*. 36a, 1981. p. 80-112.
- [16] Adler, R. and T. Rivlin. "Ergodic and mixing properties of Chebyshev polynomials," *Proceedings of the American Mathematics Society*. vol. 15, p. 794-796, 1964.
- [17] Chen, C. and K. Yao. "Basic issues in chaotic communication systems."
- [18] <http://www.rfmd.com/DataBooks/db97/2510.pdf>
- [19] <http://www.rfmd.com/DataBooks/db97/2945.pdf>
- [20] <http://www.altera.com>
- [21] Sheng, S. and R. Broderick. *Low-power CMOS Wireless Communication: A Wide-band CDMA Systems Design*. Boston: Kluwer, 1998.
- [22] Otte, R., L. de Jong, A. van Roermund. *Low-power Wireless Infrared Communication*. Boston: Kluwer, 1999.

Appendix

8.0 VHDL Code

8.1 Clock Divider

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity clk_div is
port
(
    signal smpclkx8:in std_logic;
    signal smpclk:out std_logic
);
end clk_div;

architecture behavior of clk_div is
    signal smpclk_1:std_logic;
    signal smpclk_2:std_logic;
    signal smpclk_3:std_logic;
begin

div_by_8: process(smpclkx8)
begin
if rising_edge(smpclkx8) then
    smpclk_1 <= not(smpclk_1);
else
    smpclk_1 <= smpclk_1;
end if;
if rising_edge(smpclk_1) then
    smpclk_2 <= not(smpclk_2);
else
    smpclk_2 <= smpclk_2;
end if;
if rising_edge(smpclk_2) then
    smpclk_3 <= not(smpclk_3);
else
    smpclk_3 <= smpclk_3;
end if;
smpclk <= smpclk_3;
end process div_by_8;
```

```
end behavior;
```

8.2 Polarity Decoder

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.std_logic_arith.all;  
use ieee.std_logic_unsigned.all;
```

```
entity poldec_alt2 is  
port  
(
```

```
--*****  
-- Declare I/O interface  
--*****
```

```
signal rfdata: in std_logic;  
signal smpclk: in std_logic;  
signal mrstb: in std_logic;
```

```
signal sysclrb: out std_logic;  
signal ldvar: out std_logic;  
signal dpnclk_o: out std_logic;  
signal bitdith: out std_logic_vector(1 downto 0);  
signal bitsmp: out std_logic_vector(6 downto 0);  
signal spda: out std_logic;  
signal Apol_o: out std_logic;  
signal Bpol_o: out std_logic;  
signal Cpol_o: out std_logic;  
signal bitseg: out std_logic_vector(2 downto 0)  
);  
end poldec_alt2;
```

```
--*****  
-- Declare architecture behavior body  
--*****
```

```
architecture behavior of poldec_alt2 is
```

```
--*****
-- Declare the internal signals
--*****
```

```
signal ldvar_sig1, ldvar_sig2, sysclrb_sig:std_logic;
signal bitsmp_d,bitsmp_q:std_logic_vector(6 downto 0);
signal bitseg_d,bitseg_q:std_logic_vector(2 downto 0);
signal bitdith_d,bitdith_q:std_logic_vector(1 downto 0);
signal dpnclk_d,dpnclk_q:std_logic;
signal spda_d,spda_q:std_logic;
```

```
signal Amagh:std_logic_vector(2 downto 0);
signal Bmagh:std_logic_vector(2 downto 0);
signal Cmagh:std_logic_vector(2 downto 0);
signal Amagl:std_logic_vector(2 downto 0);
signal Bmagl:std_logic_vector(2 downto 0);
signal Cmagl:std_logic_vector(2 downto 0);
signal Amag:std_logic_vector(2 downto 0);
signal Bmag:std_logic_vector(2 downto 0);
signal Cmag:std_logic_vector(2 downto 0);
signal Apol:std_logic;
signal Bpol:std_logic;
signal Cpol:std_logic;
signal temp:std_logic_vector(2 downto 0);
```

```
--*****
-- Declare the required constants
--*****
```

```
constant DEFAULT: std_logic_vector(1 downto 0) :="00";
constant RETARD: std_logic_vector(1 downto 0) :="01";
constant ADVANCE: std_logic_vector(1 downto 0) :="10";
```

```
constant RETARDPT: std_logic_vector(2 downto 0) :="011";
constant DEFAULTPT: std_logic_vector(2 downto 0) :="100";
constant ADVANCEPT: std_logic_vector(2 downto 0) :="101";
constant MAXPT: std_logic_vector(2 downto 0) :="101";
```

```
constant GND : std_logic_vector(1 downto 0) :="00";
```

```
--*****
```

```
-- If in DEFAULT: BITHDIT=0, BITSEG roll point=4, SPDA=Bpol.
--
-- If in RETARD: BITHDIT=1, BITSEG roll point=3, SPDA=Cpol.
```

```

--
-- If in ADVANCE: BITHDIT=2, BITSEG roll point=5, SPDA=Apol.

--*****

--*****
-- Declare the operational sequence definitions (text only)
--*****

-- This module is used to recover an accurate and robust data stream
-- with a flexible data clock. The recovered input (RFDATA) data
-- stream is oversampled (5 to 1 ratio) to enable an accurate and
-- flexible data sample window. Because of the times five oversample
-- five bits are sampled plus to enable a sliding (Tau-dither) scheme
-- two additional bits are sampled. The seven sampled bits are divided
-- into three windows (A,B,C) each with five bits. Magnitudes are
-- detected and compared for optimum sampling decisions, bit polarity
-- alignment, and to recover an accurate data clock from the data stream.

--*****
-- Begin architecture behavior application
--*****

begin
--The operation controller sets the system reset (sysclrb) and programmable
--settings load (ldvar)
opcon: process(smpclk, mrstb)
begin
if mrstb = '0' then
    sysclrb_sig <= '0';
    ldvar_sig1 <= '0';
    ldvar_sig2 <= '0';
elseif falling_edge(smpclk) then
    sysclrb_sig <= '1';
    ldvar_sig1 <= ldvar_sig2;
    ldvar_sig2 <= '1';
end if;
sysclrb <= sysclrb_sig;
ldvar <= ldvar_sig1;
end process opcon;

flops: process(smpclk, sysclrb_sig)
begin
    if sysclrb_sig = '0' then
        bitsmp_q <= "0000000";

```

```

        bitseg_q <= "000";
        bitdith_q <= "00";
        dpnclk_q <= '0';
        spda_q <= '0';
        elsif falling_edge(smpclk) then
            bitsmp_q <= bitsmp_d;
            bitseg_q <= bitseg_d;
            bitdith_q <= bitdith_d;
            dpnclk_q <= dpnclk_d;
            spda_q <= spda_d;
        else
            bitsmp_q <= bitsmp_q;
            bitseg_q <= bitseg_q;
            bitdith_q <= bitdith_q;
            dpnclk_q <= dpnclk_q;
            spda_q <= spda_q;
        end if;

--*****
-- Declare the port outputs for test outputs only,remove later
--*****

Apol_o <= Apol;
Bpol_o <= Bpol;
Cpol_o <= Cpol;

--*****
-- Declare the port outputs from a flip flop
--*****

spda <= spda_q;
dpnclk_o <= dpnclk_q;
bitsmp <= bitsmp_q;
bitseg <= bitseg_q;
bitdith <= bitdith_q;

end process flops;

--*****
-- Declare the process sensitivity list (inputs to the process)
--*****

process(bitsmp_q)

```

```

--*****
--*****

-- Begin the process

-- This process is used to detect the magnitude of logical low's
-- and the magnitude of logical high's for each of the three sample
-- windows (A,B,C) which are set to contain sampled bits 0-4, 1-5,
-- 2-6 respectively. A magnitude value for each of the six signals
-- (Amagh,Amagl,Bmagh,Bmagl,Cmagh,Cmagl) is output in 3 bit fields.

--*****
--*****

begin

--*****
-- Declare the flip flops which require feedback
--*****

-- no flip flops in this particular process

--*****
-- Declare the default states for the flip flops in this process.
--*****

Amagh <= (GND & bitsmp_q(0)) + (GND & bitsmp_q(1)) + (GND & bitsmp_q(2))
        + (GND & bitsmp_q(3)) + (GND & bitsmp_q(4));

Amagl <= (GND & not(bitsmp_q(0))) + (GND & not(bitsmp_q(1))) + (GND &
not(bitsmp_q(2)))
        + (GND & not(bitsmp_q(3))) + (GND & not(bitsmp_q(4)));

Bmagh <= (GND & bitsmp_q(1)) + (GND & bitsmp_q(2)) + (GND & bitsmp_q(3))
        + (GND & bitsmp_q(4)) + (GND & bitsmp_q(5));

Bmagl <= (GND & not(bitsmp_q(1))) + (GND & not(bitsmp_q(2))) + (GND &
not(bitsmp_q(3)))
        + (GND & not(bitsmp_q(4))) + (GND & not(bitsmp_q(5)));

Cmagh <= (GND & bitsmp_q(2)) + (GND & bitsmp_q(3)) + (GND & bitsmp_q(4))
        + (GND & bitsmp_q(5)) + (GND & bitsmp_q(6));

```

```
Cmagl <= (GND & not(bitsmp_q(2))) + (GND & not(bitsmp_q(3))) + (GND &
not(bitsmp_q(4)))
      + (GND & not(bitsmp_q(5))) + (GND & not(bitsmp_q(6)));
```

```
end process;
```

```
-- *****
-- End process
-- *****
```

```
-- *****
-- Declare the process sensitivity list (inputs to the process)
-- *****
```

```
process(Amagh,Bmagh,Cmagh,Amagl,Bmagl,Cmagl)
```

```
-- *****
-- *****
```

```
-- Begin the process
```

```
-- This process is used to resolve the maximum magnitude of the
-- logical low's and the magnitude of logical high's for each of
-- the three sampled windows (A,B,C). The results are output as
-- Amag,Bmag,Cmag and are represented in three bit fields. The data
-- polarity (logic low or logic high) is also resolved and is output
-- as Apol,Bpol,Cpol with each represented by a single bit.
```

```
-- *****
-- *****
```

```
begin
```

```
-- *****
-- Declare the flip flops which require feedback
-- *****
```

```
-- no flip flops in this particular process
```

```
-- *****
-- Declare the default states for the flip flops in this process.
-- *****
```

```

Amag <= Amagh;
Bmag <= Bmagh;
Cmag <= Cmagh;
Apol <= '1';
Bpol <= '1';
Cpol <= '1';

--*****
-- Begin body of process arguments
--*****

        if (Amagh < Amagl) then
            Amag <= Amagl;
            Apol <= '0';
        end if;

        if (Bmagh < Bmagl) then
            Bmag <= Bmagl;
            Bpol <= '0';
        end if;

        if (Cmagh < Cmagl) then
            Cmag <= Cmagl;
            Cpol <= '0';
        end if;

end process;

--*****
-- End process
--*****

--*****
-- Declare the process sensitivity list (inputs to the process)
--*****

process(rfdata,bitbmp_q,bitseg_q,bitdith_q,dpnclk_q,spda_q,Amag,
        Bmag,Cmag,Apol,Bpol,Cpol)

--*****
--*****

-- Begin the process

```

```

-- This process is used to resolve which 5 bit window (A 0-4,
-- B 1-5,C 2-6) of data has the best ratio of matching bits.
-- This decision (bitdith) is used in a Tau-dither scheme to
-- determine where the sample window needs to be placed (RETARD,
-- DEFAULT, ADVANCE) for the next cycle. This decision is also
-- used to slide the trigger point (RETARDPT,DEFAULTPT,ADVANCEPT)
-- for the outbound derived (DPNCLK) PN clock.

```

```

--*****
--*****

```

```
begin
```

```

--*****
-- Declare the flip flops which require feedback
--*****

```

```

spda_d <= spda_q;
dpnclk_d <= dpnclk_q;
bitdith_d <= bitdith_q;

```

```

--*****
-- Declare the default states for the flip flops in this process.
--*****

```

```
dpnclk_d <= '0';
```

```
bitseg_d <= bitseg_q+1;
```

```

bitsmp_d(6 downto 1) <= bitsmp_q(5 downto 0);
bitsmp_d(0) <= rfdata;

```

```

--*****
-- Begin body of process arguments
--*****

```

```

    if (bitseg_q = MAXPT) OR
    ((bitdith_q = DEFAULT) AND (bitseg_q = DEFAULTPT)) OR
    ((bitdith_q = RETARD) AND (bitseg_q = RETARDPT)) OR
    ((bitdith_q = ADVANCE) AND (bitseg_q = ADVANCEPT)) then
    bitseg_d <= "000";
    dpnclk_d <= '1';

```

```

        if (Amag > Bmag) AND (Bmag >= Cmag) then
            spda_d <= Apol;
            bitdith_d <= ADVANCE;
        elsif (Cmag > Bmag) AND (Bmag >= Amag) then
            spda_d <= Cpol;
            bitdith_d <= RETARD;
        else
            spda_d <= Bpol;
            bitdith_d <= DEFAULT;
        end if;
    end if;

    if bitseg_q < "001" then
        dpnclk_d <= '1';

    end if;
end process;

--*****
-- End process
--*****

end behavior;

```

8.3 Despreader Correlator

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity desprd2_alt is
port
(

--*****
-- Declare I/O interface
--*****

signal spda:   in std_logic;

```

```

signal dpnclk: in std_logic;
signal ldvar:  in std_logic;
signal sysclrb: in std_logic;

signal dpack:  out std_logic;
signal dspclk: out std_logic;
signal pretrk: out std_logic;
signal trk:    out std_logic;
signal mbit:   out std_logic;
signal mdet:   out std_logic;
signal dsretard_o:out std_logic;
signal dsdefault_o:out std_logic;
signal dsadvance_o:out std_logic;
signal maxpol_o:out std_logic;

signal maxmag_o:out std_logic_vector(5 downto 0);
signal dsseg:   out std_logic_vector(6 downto 0);
signal dstkacc:out std_logic_vector(1 downto 0);
signal dsntacc:out std_logic_vector(3 downto 0)
);
end desprd2_alt;

```

```

--*****
-- Declare architecture behavior body
--*****

```

architecture behavior of desprd2_alt is

```

--*****
-- Declare the internal signals
--*****

```

```

signal dpack_d,dpack_q:std_logic;
signal dspclk_d,dspclk_q:std_logic;
signal pretrk_d,pretrk_q:std_logic;
signal trk_d,trk_q:std_logic;
signal mbit_d,mbit_q:std_logic;
signal mdet_d,mdet_q:std_logic;
signal dsCpol_d,dsCpol_q:std_logic;
signal dsBpol_d,dsBpol_q:std_logic;
signal dsApol_d,dsApol_q:std_logic;
signal dsretard:std_logic;
signal dsdefault:std_logic;
signal dsadvance:std_logic;
signal dsretard_d,dsretard_q: std_logic;

```

```

signal dsdefault_d,dsdefault_q: std_logic;
signal dsadvance_d,dsadvance_q: std_logic;
signal maxpol: std_logic;

signal dsseg_d,dsseg_q:std_logic_vector (6 downto 0);
signal dsmbth_d,dsmbth_q:std_logic_vector (5 downto 0);
signal dsmdth_d,dsmdth_q:std_logic_vector (5 downto 0);
signal dstkth_d,dstkth_q:std_logic_vector (1 downto 0);
signal dsnth_d,dsnth_q:std_logic_vector (3 downto 0);
signal dsmdacc_d,dsmdacc_q:std_logic_vector (5 downto 0);
signal dstkacc_d,dstkacc_q:std_logic_vector (1 downto 0);
signal dsntacc_d,dsntacc_q:std_logic_vector (3 downto 0);
signal xor_array: std_logic_vector (62 downto 0);
signal sum1,sum2,sum3,sum4: std_logic_vector (5 downto 0);
signal dsshf_d,dsshf_q:std_logic_vector (62 downto 0);
signal dspn_d: std_logic_vector (62 downto 0);
signal dspn_q: std_logic_vector (62 downto 0);
signal dsCmag_d,dsCmag_q:std_logic_vector (5 downto 0);
signal dsBmag_d,dsBmag_q:std_logic_vector (5 downto 0);
signal dsAmag_d,dsAmag_q:std_logic_vector (5 downto 0);
signal compmag:std_logic_vector (5 downto 0);
signal maxmag:std_logic_vector (5 downto 0);

--*****
-- Declare the required constants
--*****

constant DSRETARDPT: std_logic_vector(6 downto 0) := "0111101";
constant DSDEFAULTPT: std_logic_vector(6 downto 0) := "0111110";
constant DSADVANCEPT: std_logic_vector(6 downto 0) := "0111111";
constant DSMAXPT: std_logic_vector(6 downto 0) := "1000000";

constant PNCODE: std_logic_vector(62 downto 0):=
"101010110011011101101001001110001011110010100011000010000011111";
constant GND6:std_logic_vector(5 downto 0):= "000000";

--set programmable thresholds as constants to free up logic cells.
--dsmbth=50, dsmdth=62, dsnth=15, dstkth=2
constant dsmbth:std_logic_vector(5 downto 0) := "110010";
constant dsmdth:std_logic_vector(5 downto 0) := "111110";
constant dstkth:std_logic_vector(1 downto 0) := "10";
constant dsnth:std_logic_vector(3 downto 0) := "1111";

--*****
-- Declare the operational sequence definitions (text only)

```



```

dspclk <= dspclk_q;
pretrk <= pretrk_q;
trk <= trk_q;
mbit <= mbit_q;
mdet <= mdet_q;
dsseg <= dsseg_q;
dstkacc <= dstkacc_q;
dsntacc <= dsntacc_q;

```

```

dsretard_o <= dsretard_q;
dsdefault_o <= dsdefault_q;
dsadvance_o <= dsadvance_q;
maxpol_o <= maxpol;
maxmag_o <= maxmag;

```

```

end process flops;

```

```

--*****
-- Declare the process sensitivity list (inputs to the process)
--*****

```

```

process(spda,dsshf_q,dspn_q)

```

```

--*****
--*****

```

```

-- Begin the P1 process

```

```

-- This process is used to accumulate the input spread data stream.
-- The input spread data is right shifted into a 63 (PN spread ratio)
-- bit shift register. The chip bits (spda) are clocked into this
-- shift register with the derived PN clock (dpnclk). The spda data
-- and the ddspnclk are sent to the desreader module from the polarity
-- decoder (POLDEC) module. The results from this process are the
-- expected PN (PNCODE) code and the accumulated (dsshf_q) inbound
-- PN value.

```

```

--*****
--*****

```

```

begin

```

```

--*****
-- Declare the flip flops which require feedback
--*****

```

```

--*****
-- Declare the default states for the flip flops in this process.
--*****

dsshf_d(62 downto 1) <= dsshf_q(61 downto 0);
dsshf_d(0) <= spda;

dspn_d <= PNCODE;

xor_array <= dsshf_q XOR dspn_q;

end process;

--*****
-- End process
--*****

--*****
-- Declare the process sensitivity list (inputs to the process)
--*****

process(xor_array)
begin

sum1<=(GND6&xor_array(0))+(GND6&xor_array(1))+(GND6&xor_array(2))+(GND6
&xor_array(3))+(GND6&xor_array(4))

+(GND6&xor_array(5))+(GND6&xor_array(6))+(GND6&xor_array(7))+(GND6&xor_a
rray(8))+(GND6&xor_array(9))

+(GND6&xor_array(10))+(GND6&xor_array(11))+(GND6&xor_array(12))+(GND6&xor
rray(13))+(GND6&xor_array(14));

sum2<=(GND6&xor_array(15))+(GND6&xor_array(16))+(GND6&xor_array(17))+(GN
D6&xor_array(18))+(GND6&xor_array(19))

+(GND6&xor_array(20))+(GND6&xor_array(21))+(GND6&xor_array(22))+(GND6&x
or_array(23))+(GND6&xor_array(24))

```

```
+(GND6&xor_array(25))+ (GND6&xor_array(26))+ (GND6&xor_array(27))+ (GND6&xor_array(28))+ (GND6&xor_array(29))
+(GND6&xor_array(30));
```

```
sum3<=(GND6&xor_array(31))+ (GND6&xor_array(32))+ (GND6&xor_array(33))+ (GND6&xor_array(34))+ (GND6&xor_array(35))
```

```
+(GND6&xor_array(36))+ (GND6&xor_array(37))+ (GND6&xor_array(38))+ (GND6&xor_array(39))+ (GND6&xor_array(40))
```

```
+(GND6&xor_array(41))+ (GND6&xor_array(42))+ (GND6&xor_array(43))+ (GND6&xor_array(44))+ (GND6&xor_array(45))
+(GND6&xor_array(46));
```

```
sum4<=(GND6&xor_array(47))+ (GND6&xor_array(48))+ (GND6&xor_array(49))+ (GND6&xor_array(50))+ (GND6&xor_array(51))
```

```
+(GND6&xor_array(52))+ (GND6&xor_array(53))+ (GND6&xor_array(54))+ (GND6&xor_array(55))+ (GND6&xor_array(56))
```

```
+(GND6&xor_array(57))+ (GND6&xor_array(58))+ (GND6&xor_array(59))+ (GND6&xor_array(60))+ (GND6&xor_array(61))
+(GND6&xor_array(62));
```

```
end process;
```

```
P2: process(sum1,sum2,sum3,sum4)
```

```
--*****
--*****
```

```
-- Begin the P2 process
```

```
-- This process is used to perform an XOR and comparison of the
-- expected PN code and the inbound PN code. The result is a zero
-- polarity magnitude, based on the 63 bit total magnitude field.
```

```
--*****
--*****
```

```
begin
```

```
compmag <= sum1 + sum2 + sum3 + sum4;
```

end process P2;

```
--*****  
-- End process  
--*****
```

```
--*****  
-- Declare the process sensitivity list (inputs to the process)  
--*****
```

P3:

```
process(compmag,dsAmag_q,dsBmag_q,dsCmag_q,dsApol_q,dsBpol_q,dsCpol_q,  
dsmbth_q,dsmdth_q,dsnth_q,dstkth_q,dsmdacc_q,dsntacc_q,dstkacc_q)
```

```
--*****  
--*****
```

-- Begin the P3 process

```
-- This process is used to perform four essential and sequential  
-- operations. The first operation is to subtract a zero magnitude  
-- from the comparator (from the P2 process) magnitude. The second  
-- operation is to derive a bit polarity from the PN magnitude.  
-- The third operation stores and rotates three PN magnitudes and  
-- their associated bit polarities. The fourth operation is used  
-- to compare the three magnitude values and decide if the dither  
-- controller needs to be in default or retard or advance.
```

```
--  
-- retard mode: dsretard=1,dsseq roll=61,maxmag=dsAmag.maxpol=dsApol  
-- default mode: dsdefault=1,dsseq roll=62,maxmag=dsBmag.maxpol=dsBpol  
-- advance mode: dsadvance=1,dsseq roll=63,maxmag=dsCmag.maxpol=dsCpol
```

```
--*****  
--*****
```

begin

```
--*****  
-- Declare the flip flops which require feedback  
--*****
```

```
--*****  
-- Declare the default states for the flip flops in this process.
```

```

--*****

        dsBmag_d <= dsCmag_q;
        dsBpol_d <= dsCpol_q;
        dsAmag_d <= dsBmag_q;
        dsApol_d <= dsBpol_q;

--*****
-- Begin body of process arguments
--*****

        if (63 - compmag) > compmag then
            dsCmag_d <= (63 - compmag);
            dsCpol_d <= '0';
        else dsCmag_d <= compmag;
            dsCpol_d <= '1';
        end if;

        if (dsAmag_q > dsBmag_q) AND (dsBmag_q >= dsCmag_q) then
            maxmag <= dsAmag_q;
            maxpol <= dsApol_q;
            dsadvance <= '0';
            dsdefault <= '0';
            dsretard <= '1';
        elsif (dsCmag_q > dsBmag_q) AND (dsBmag_q >= dsAmag_q) then
            maxmag <= dsCmag_q;
            maxpol <= dsCpol_q;
            dsadvance <= '1';
            dsdefault <= '0';
            dsretard <= '0';
        else
            maxmag <= dsBmag_q;
            maxpol <= dsBpol_q;
            dsadvance <= '0';
            dsdefault <= '1';
            dsretard <= '0';
        end if;

end process P3;

--*****
-- End process
--*****

```

```

--*****
-- Declare the process sensitivity list (inputs to the process)
--*****
P4:
process(maxmag,maxpol,dsretard,dsdefault,dsadvance,dpack_q,dspclk_q,pretrk_q,
trk_q,dsmbth_q,dsmdth_q,dsnth_q,dstkth_q,dsmdacc_q,dstkacc_q,dsntacc_q,dsseg_q,
dsdefault_q,dsretard_q,dsadvance_q)

--*****
--*****

-- Begin the P4 process

-- This process is used to derive the spread (dspclk) clock. This
-- clock is generated by applying the DSRETARDPT or DSDEFAULTPT or
-- DSADVANCEPT to slide the trigger point for the outbound derived
-- spread clock. This process also resolves the the outbound despread
-- (dpack) data. This process also cotrols whether the despreader
-- correlator module is in search or pretrack or track mode. This
-- process also measures missed bit and missed detect thresholds.

--*****
--*****

begin

--*****
-- Declare the flip flops which require feedback
--*****

dpack_d <= dpack_q;
dspclk_d <= dspclk_q;
pretrk_d <= pretrk_q;
trk_d <= trk_q;
dstkacc_d <= dstkacc_q;
dsntacc_d <= dsntacc_q;
dsseg_d <= dsseg_q+1;

--*****
-- Declare the default states for the flip flops in this process.
--*****

dspclk_d <= '0';
mbit_d <= '0';

```

```

mdet_d <= '0';
dsmbth_d <= dsmbth;
dsmnth_d <= dsmdth;
dstkth_d <= dstkth;
dsnth_d <= dsnth;
dsdefault_d <= dsdefault_q;
dsretard_d <= dsretard_q;
dsadvance_d <= dsadvance_q;

```

```

--*****
-- Begin body of process arguments
--*****

```

```

if (dsseg_q = DSMAXPT) OR
  ((dsdefault_q = '1') AND (dsseg_q = DSDEFAULTPT)) OR
  ((dsretard_q = '1') AND (dsseg_q = DSRETARDPT)) OR
  ((dsadvance_q = '1') AND (dsseg_q = DSADVANCEPT)) then
  dsdefault_d <= dsdefault;
  dsretard_d <= dsretard;
  dsadvance_d <= dsadvance;
  dsseg_d <= "0000000";
  dspclk_d <= '1';
  dpack_d <= maxpol;

```

```

if trk_q = '0' then
  if maxmag > dsmdth_q then
    dstkacc_d <= dstkacc_q+1;
    if dstkacc_q > dstkth_q then
      trk_d <= '1';
    end if;
  else
    pretrk_d <= '0';
  end if;
else
  if maxmag < dsmdth_q then
    dsntacc_d <= dsntacc_q+1;
    if dsntacc_q > dsnth_q then
      trk_d <= '0';
    end if;
  else
    dsntacc_d <= "0000";
  end if;
end if;

```

```

        if (trk_q = '1') AND (maxmag < dsmbth_q) then
            mbit_d <= '1';
        end if;

        if (trk_q = '1') AND (maxmag < dsmdth_q) then
            mdet_d <= '1';
        end if;

    end if;

    if (maxmag > dsmbth_q) AND (pretrk_q = '0') then
        pretrk_d <= '1';
        dsseg_d <= "0000000";
        dstkacc_d <= "00";
        dsntacc_d <= "0000";
    end if;

    if pretrk_q = '0' then
        dpack_d <= '0';
    end if;

    if dsseg_q < "0100000" then
        dspclk_d <= '1';
    end if;

end process P4;

--*****
-- End process
--*****

end behavior;

```

8.4 Protocol Remover

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

```

```

use ieee.std_logic_unsigned.all;

--
*****
*
-- I/O Interface Declaration
--
*****
*

entity protocol_alt is
port
(
-- inputs

    signal dpack:    in std_logic;
    signal dspclk:   in std_logic;
    signal sysclrb:  in std_logic;
    signal trk:      in std_logic;
    signal mdet:     in std_logic;
    signal mbit:     in std_logic;

-- outputs

    signal strk:     out std_logic;
    signal smdet:    out std_logic;
    signal smbiter:  out std_logic;
    signal sdata:    out std_logic

);
end protocol_alt;

--
*****
*
-- Architecture body
--
*****
*

architecture behavior of protocol_alt is

    signal in1, in2:std_logic;
    signal sdata_sig: std_logic;
    signal trk_q:    std_logic;

```

```

signal mdet_q: std_logic;
signal mbit_q: std_logic;

begin

process(sysclrb, dspclk)

begin
if sysclrb = '0' then
    trk_q <= '0';
    mdet_q <= '0';
    mbit_q <= '0';
    in1 <= '1';
    in2 <= '0';
elsif rising_edge(dspclk) then
    trk_q <= trk;
    mdet_q <= mdet;
    mbit_q <= mbit;
    in2 <= in1;
    in1 <= dpack;
end if;

sdata_sig <= in1 xor in2;
strk <= trk_q;
smdet <= mdet_q;
smbit <= mbit_q;
sdata <= sdata_sig;

end process;

end behavior;

```

8.5 Packet Detector

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

```

```

--*****
-- I/O Interface Declaration
--*****

entity pack_det is
port(
--inputs
  sdata:      in std_logic;
  dspclk:     in std_logic;
  -- uid:      in std_logic_vector(2 downto 0);
  clacq:      in std_logic;
  rstb:       in std_logic;
  ldvar:      in std_logic;
  strk:       in std_logic;

--outputs
  wsclk:      buffer std_logic;
  pkda_ec:    bufferstd_logic_vector(9 downto 0);
  match:      buffer std_logic;
  pkdwr:      out std_logic;
  pkwc:       buffer integer range 0 to 63;
  pkwvr:      buffer std_logic;
  acq:        out std_logic;
  pkda:       out std_logic_vector(9 downto 0)
);
end pack_det;

--*****
-- Architecture Body
--*****
architecture behavior of pack_det is

--Constants for Graviton demo
--Packet words are transmitted from the ACQ LSB first
constant pklen:integer:= 6;
constant pkwid:integer := 10;
constant frame_sync:std_logic_vector(9 downto 0):="1111100000";
constant GNDA:std_logic_vector(9 downto 0):="0000000000";
constant GND7:std_logic_vector(6 downto 0):="0000000";

--set programmable parameters as constants to free up logic cells.
constant uid: std_logic_vector(2 downto 0) := "101";

signal id_cnt: integer range 0 to 20;
signal uid_sig: std_logic_vector(2 downto 0);

```

```

signal pkwvr_sig:std_logic;
signal pkdwr_sig:std_logic;
signal pkdwr_sig_d1:std_logic;
signal pkdwr_sig_d2:std_logic;
signal pkdwr_sig_d3:std_logic;
signal pkdwr_sig_d4:std_logic;
signal pkdwr_sig_d5:std_logic;
signal acq_sig: std_logic;
signal ref_uid: std_logic_vector(pkwid-1 downto 0);
signal pkwc_sig:integer range 0 to 63;
signal ec_cnt: integer range 0 to 10;
signal pkda_sig:std_logic_vector(9 downto 0);
signal pkda_sig1:std_logic_vector(pkwid-1 downto 0);
signal pkda_sig2:std_logic_vector(pkwid-1 downto 0);
signal pkda_int:std_logic_vector(9 downto 0);

begin

acquisition: process(rstb, clacq, dspclk)
begin
if ldvar = '1' then
uid_sig(0) <= uid(2);
uid_sig(1) <= uid(1);
uid_sig(2) <= uid(0);
ref_uid <= GND7 & uid_sig;
else
ref_uid <= ref_uid;
end if;
if rstb = '0' or clacq = '1' then
pkda_int <= GNDA;
match <= '0';
acq_sig <= '0';
ec_cnt <= 0;
wsclk <= '1';
id_cnt <= 0;
elsif falling_edge(dspclk) then
pkda_int(8 downto 0) <= pkda_int(9 downto 1);
pkda_int(9) <= sdata;
if (match or acq_sig) = '1' then
if ec_cnt = 9 then
pkda_ec <= pkda_int;
ec_cnt <= 0;
wsclk <= '0';
else
pkda_ec <= GNDA;

```

```

ec_cnt <= ec_cnt + 1;
wsclk <= '1';
end if;
else
ec_cnt <= ec_cnt;
wsclk <= '1';
end if;
if pkda_int = frame_sync then
match <= '1';
else
match <= match;
end if;
if match = '1' and acq_sig = '0' then
if pkda_int = ref_uid and id_cnt = 19 then
acq_sig <= '1';
id_cnt <= 0;
else
if id_cnt = 20 then
acq_sig <= '0';
id_cnt <= 0;
match <= '0';
else
acq_sig <= acq_sig;
id_cnt <= id_cnt + 1;
end if;
end if;
else
acq_sig <= acq_sig;
id_cnt <= id_cnt;
end if;

end if;
acq <= acq_sig;
end process acquisition;

```

```

data_strobe: process(dspclk, rstb, acq_sig)
begin
if rstb = '0' or acq_sig = '0' then
pkwvr_sig <= '1';
pkdwr_sig <= '1';
pkdwr_sig_d1 <= '1';
pkdwr_sig_d2 <= '1';
pkdwr_sig_d3 <= '1';
pkdwr_sig_d4 <= '1';
elsif falling_edge(dspclk) then

```

```

if acq_sig = '1' then
pkdwr_sig_d4 <= wsclk;
else
pkdwr_sig_d4 <= pkdwr_sig_d4;
end if;
pkdwr_sig <= pkdwr_sig_d1;
pkdwr_sig_d1 <= pkdwr_sig_d2;
pkdwr_sig_d2 <= pkdwr_sig_d3;
pkdwr_sig_d3 <= pkdwr_sig_d4;
if pkwc_sig = pklen then
pkwvr_sig <= pkdwr_sig;
else
pkwvr_sig <= pkwvr_sig;
end if;
if strk = '0' then
pkwvr_sig <= '0';
end if;
end if;
pkdwr <= pkdwr_sig;
pkwvr <= pkwvr_sig;
end process data_strobe;

```

```

data_out: process(dspclk, rstb)
begin
if rstb = '0' then
pkda_sig <= GNDA;
pkda_sig1 <= GNDA;
elsif rising_edge(dspclk) then
if wsclk = '0' then
pkda_sig1 <= pkda_sig;
pkda_sig <= pkda_ec;
else
pkda_sig1 <= pkda_sig1;
pkda_sig <= pkda_sig;
end if;
end if;
pkda <= pkda_sig1;
end process data_out;

```

```

word_count: process(wsclk, rstb, acq_sig)
begin
if rstb = '0' or acq_sig = '0' then
pkwc_sig <= 0;
elsif rising_edge(wsclk) then
if pkwc_sig = pklen or strk = '0' then

```

```

        pkwc_sig <= pkwc_sig;
    else
        pkwc_sig <= pkwc_sig + 1;
    end if;
end if;
pkwc <= pkwc_sig;
end process word_count;

end behavior;

```

8.6 FIFO Controller

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--*****
-- I/O Interface Declaration
--*****

entity fctrl is
port(
--inputs
    signal dspclk:in std_logic;
    signal rstb:   in std_logic;
    signal strk:   in std_logic;
    signal rd:     in std_logic;
    signal acq:    in std_logic;
    signal pkst:   in std_logic;
    signal pkdwr:in std_logic;
    signal pkwwr:in std_logic;
    signal full:   in std_logic;
                    --Data FIFO full
    signal mty:    in std_logic;
                    --Data FIFO empty
    signal temp_full:in std_logic;

--outputs
    signal ovr:    buffer std_logic;
    signal temp_clrb:buffer std_logic;
                    --temp_clrb is the asynchronous active low data FIFO clear.
    signal temp_pwr:buffer std_logic;

```

```

--tempwr is the active high write to the data FIFO signal.
signal tpclk: buffer std_logic;
signal drdy: buffer std_logic;
signal xfer: buffer std_logic;
--xfer is the active high data FIFO write signal
signal dfclk: buffer std_logic;
signal dfrd: buffer std_logic;
signal clacq: buffer std_logic
);
end fctrl;

```

```

--*****
-- Architecture Body
--*****
architecture behavior of fctrl is

```

```

--The data FIFO is 96 by 10
--Expected word count is the expected packet length minus 2.
--For the Graviton demo, pklen = 6.

```

```

constant pklen:integer := 6;
constant pkwid:integer := 10;

```

```

signal temp_clrb_d, temp_clrb_q:std_logic;
signal tempwr_d, tempwr_q:std_logic;
signal dfrd_d, dfrd_q:std_logic;
signal clacq_d, clacq_q:std_logic;
signal drdy_d, drdy_q:std_logic;
signal xfer_d, xfer_q:std_logic;
signal xfer_d2, xfer_q2:std_logic;
signal ovr_d, ovr_q:std_logic;
signal dfwrclk:std_logic;
signal tprdelk: std_logic;
signal clk_cnt: integer range 0 to pkwid;
signal ovr_cnt:integer range 0 to 7;
signal xfer_cnt:integer range 0 to pklen;
signal dfclk_d:std_logic;
signal dfclk_q:std_logic;

```

```

begin

```

```

--*****
-- Synchronicity
--*****
flops: process(rstb, dspclk)

```

```

begin
if rstb = '0' then
    temp_clrb_q <= '1';
    tempwr_q <= '0';
    clacq_q <= '0';
    xfer_q <= '0';
    xfer_q2 <= '0';
    drdy_q <= '0';
    ovr_q <= '0';
    dfclk_q <= '1';
elsif falling_edge(dspclk) then
    temp_clrb_q <= temp_clrb_d;
    tempwr_q <= tempwr_d;
    clacq_q <= clacq_d;
    xfer_q <= xfer_d;
    xfer_q2 <= xfer_d2;
    drdy_q <= drdy_d;
    ovr_q <= ovr_d;
    dfclk_q <= dfclk_d;
end if;
temp_clrb <= temp_clrb_q;
tempwr <= tempwr_q;
dfrd <= drdy_q;
clacq <= clacq_q;
xfer_d2 <= xfer_q;
xfer <= xfer_q2;
drdy <= drdy_q;
dfclk <= dfclk_q;
ovr <= ovr_q;
end process flops;

--*****
-- Clock Generation
--*****

--Generate clocks for the single-clock FIFOs

clk_gen: process(rstb, dspclk)
begin
if rstb = '0' then
    clk_cnt <= 0;
    tprdclk <= '1';
    dfwrcclk <= '1';
elsif falling_edge(dspclk) then
    case clk_cnt is

```

```

        when 0 =>
            tprdclk <= '0';
            dfwrclk <= '1';
            clk_cnt <= clk_cnt + 1;
        when 1 =>
            tprdclk <= '1';
            dfwrclk <= '0';
            clk_cnt <= clk_cnt + 1;
        when pkwid - 1 =>
            clk_cnt <= 0;
            tprdclk <= '1';
            dfwrclk <= '1';
        when others =>
            tprdclk <= '1';
            dfwrclk <= '1';
            clk_cnt <= clk_cnt + 1;
        end case;
    end if;
end process clk_gen;

--*****
-- Clock Selection
--*****

--Generate the data ready signal (drdy) and switch clocks

clk_sel: process(rstb, dspclk)
begin
    if rstb = '0' or mty = '1' then
        drdy_d <= '0';
    elsif rising_edge(full) then
        drdy_d <= '1';
    end if;
    if drdy = '1' then
        dfclk_d <= rd;
    elsif xfer = '1' then
        dfclk_d <= dfwrclk;
    else
        dfclk_d <= '1';
    end if;
    if xfer = '0' then
        tpcclk <= pkdwr;
    else
        tpcclk <= tprdclk;
    end if;
end process;

```

```
end process clk_sel;
```

```
--*****  
-- Data FIFO and Temporary Packet Buffer Control  
--*****
```

```
--Write to the temporary packet buffer is disabled if tracking is  
--lost (strk=0), the packet is bad (pkst=1), or the data FIFO has  
--overflowed (temp_full=1). If the packet is good then it is written to  
--the temporary packet buffer.
```

```
temp_ctrl: process(rstb, dspclk)  
begin  
if rstb = '0' or acq = '0' or drdy = '1' or temp_full = '1' then  
    tempwr_d <= '0';  
elsif falling_edge(dspclk) then  
    tempwr_d <= '1';  
end if;  
clacq_d <= pkst or not (pkwvr) or drdy;  
temp_clrb_d <= rstb and not(pkst) and not(full);  
end process temp_ctrl;
```

```
--If the data in the temporary packet buffer is ready to  
--be sent to the data FIFO, then writing to the data FIFO and reading  
--from the temporary packet buffer (xfer) is enabled.
```

```
data_ctrl: process(rstb, dfclk)  
begin  
if rstb = '0' then  
    xfer_cnt <= 0;  
    xfer_d <= '0';  
elsif rising_edge(dfwrclk) then  
    case xfer_cnt is  
    when 0 =>  
        if temp_full = '1' then  
            xfer_cnt <= 1;  
        end if;  
    when 6 =>  
        xfer_cnt <= 0;  
    when others =>  
        xfer_cnt <= xfer_cnt + 1;  
    end case;  
    if xfer_cnt = 0 then  
        xfer_d <= '0';  
    else
```

```

                xfer_d <= '1';
            end if;
        end if;
    end process data_ctrl;

    --*****
    -- Overflow
    --*****

    oc: process(rstb, mty, dspclk)
    begin
        if rstb = '0' or mty = '1' then
            ovr_cnt <= 0;
        elsif falling_edge(dspclk) then
            if full = '1' then
                ovr_cnt <= ovr_cnt + 1;
            else
                ovr_cnt <= ovr_cnt;
            end if;
            if ovr_cnt = 100 then
                ovr_d <= '1';
            else
                ovr_d <= '0';
            end if;
        end if;
    end process oc;

    end behavior;

```

8.7 Packet Error Logic

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

--*****
-- I/O Interface Declaration
--*****

entity pack_err is

```

```

port
(
-- inputs
  signal clk:    in std_logic;
  signal pkwvr:in std_logic;
  signal pkwc:  in integer range 0 to 63;
-- signal pkwth:in integer range 0 to 63;
  signal mbit:  in std_logic;
-- signal pkmbth:in integer range 0 to 63;
  signal mdet:  in std_logic;
-- signal pkmdth:in integer range 0 to 63;
  signal acq:   in std_logic;
  signal sysclrb:in std_logic;
  signal ldvar: in std_logic;

-- outputs
  signal ambit: buffer integer range 0 to 63;
  signal amdets: buffer integer range 0 to 63;
  signal pkst:  buffer std_logic
);
end pack_err;

--*****
-- Architecture body
--*****

architecture behavior of pack_err is

  signal mbabove:std_logic;
  signal mdabove:std_logic;
  signal wcbelow:std_logic;
  signal mbth:   integer range 0 to 63;
  signal mdth:   integer range 0 to 63;
  signal wcth:   integer range 0 to 63;

--set programmable thresholds as constants to free up logic cells.
constant pkwth:integer := 6;
constant pkmdth:integer := 3;
constant pkmbth:integer := 3;

begin

--*****
-- Load Programmable Settings
--*****

```

```

load: process (ldvar)
begin
if rising_edge(ldvar) then
    --Load externally programmable settings
    mbth <= pkmbth;
    mdth <= pkmdth;
    wcth <= pkwcth;
else
    mbth <= mbth;
    mdth <= mdth;
    wcth <= wcth;
end if;
end process load;

--*****
-- Missed Bit and Missed Detect Accumulators
--*****
acc: process (mbit, mdet, sysclrb, acq)
begin
if sysclrb = '0' or acq = '0' then
    ambit <= 0;
elsif falling_edge(mbit) then
    ambit <= ambit + 1;
end if;
if sysclrb = '0' or acq = '0' then
    amdet <= 0;
elsif falling_edge(mdet) then
    amdet <= amdet + 1;
end if;
end process acc;

--*****
-- Packet Status Check
--*****
psc: process (clk, pkwvr, sysclrb, acq)
begin
if sysclrb = '0' then
    mbabove <= '0';
    mdabove <= '0';
    wcbelow <= '0';
elsif falling_edge(clk) then
    --Compare missed detect and missed bit counts to thresholds
    if (ambit > mbth) then
        mbabove <= '1';
    else

```

```

        mbabove <= '0';
    end if;
    if (amdet > mdth) then
        mdabove <= '1';
    else
        mdabove <= '0';
    end if;
    -- Check incoming packet word count to see if it is below the threshold
    if (pkwc < wcth) then
        wcbelow <= '1';
    else
        wcbelow <= '0';
    end if;
end if;
if acq = '0' then
    pkst <= '0';
elseif falling_edge(pkwwr) then
    --Check for packet errors and set pkst flag high if packet is bad
    pkst <= mdabove OR mbabove OR wcbelow;
end if;
end process psc;
end behavior;

```

Vita

Brian Parker Chesney

Received Bachelor's of Science in Electrical Engineering from Rice University in 1998.

The BSEE had a systems emphasis with course work concentrated in VLSI design, digital communications and digital signal processing. Have been working at the Oak Ridge National Laboratory under Dr. Charles Britton in the Monolithic Systems Group in the Instrumentation and Control Division. Work has focused on telesensing using digital communication. Interests include VLSI design, information theory, digital signal processing and digital communications.