University of Alberta

Simulation of Quantization Noise Effects on the Performance of a Wireless Preamble Detector and Demonstration of a Functional FPGA Prototype

by

Eric Tien Tze Son

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

Fall 2009
Edmonton, Alberta

Examining Committee

Dr. Vincent Gaudet, Electrical and Computer Engineering

Dr. Christian Schlegel, Computing Science

Dr. Bruce Cockburn, Electrical and Computer Engineering

Abstract

This thesis describes the implementation of the physical layer for an experimental low-power wireless communication device. The system utilizes differential coherent correlation and threshold-based detection to produce a robust random-access packet-based communications protocol. Prior to implementing the system in hardware, the detection algorithm was rigorously simulated with a software model in C. The simulations revealed the tradeoffs between the packet miss performance and different system parameters such as input bit precision and threshold value. Having determined a suitable configuration, the detection algorithm was implemented on an FPGA platform. The focus of the FPGA design was on throughput and resource utilization. The final system utilizes approximately 6% of the slices available on a Xilinx Virtex II XC2V8000 FPGA and has a throughput of about 5 MChips/Second.

Acknowledgement

I would like to express my sincerest gratitude to Dr. Vincent Gaudet and Dr. Christian Schlegel. This project would not have been possible without their guidance.

I would like to thank Dr. Bruce Cockburn for taking the time to review my thesis and serving on my final examination committee.

I would also like to thank my fellow lab mates Marcel Jar, Lukasz Kryzmien, Majid Ghanbarinejad, Malihe Ahmadi, Sumeeth Nagaraj, John Koob, Andrew Hakman, Brendan Crowley, Ji Sun, Saina Lajevardi and Mitchiko Maruyama for providing a supportive and wonderful working environment. A special thanks to Saeed Fard for sharing his knowledge on FPGA design. Furthermore, I would like to thank Russell Dodd for helping me with a wide variety of random tasks.

Finally, I must thank my family and Diana for their never-ending support and love.

Table of Contents

Chapter 1 Introduction xiii

1.1 Wireless Sensor Networks 1

1.2 Wireless Sensor Node 1

1.3 Thesis Overview 2

Chapter 2 Background 3

2.1 Differential Coding 3

2.2 Direct Sequence Spread Spectrum 5

2.3 Related Work 5

2.4 Packet Structure 8

2.5 Preamble Detection Algorithm 9

2.6 Pulse Shaping 11

Chapter 3 Simulations 13

3.1 System Parameters 13

3.2 Bit-true simulation 15

3.3 Chapter Summary 21

Chapter 4 FPGA Implementation 23

4.1 Platform and Design Overview 23

4.2 Transmitter 25

4.3 Receiver 29

4.4 Noise generator 36

4.5 Chapter Summary 40

Chapter 5 Results 41

5.1 C Simulations 41

5.2 FPGA Implementation 49

Chapter 6 Conclusions and Future Work 55

6.1 Future Work 55

Bibliography 59

List of Tables

Table 2.1: Summary of some wireless sensor nodes. 7
Table 2.2: Packet structure comparison 9
Table 4.1: Transmitter controller module state description 29
Table 4.2: Sign signal operation summary 31
Table 5.1: Simulation transmit signal window alignment 42
Table 5.2: Mutually exclusive events that can occur when detecting preambles 43
Table 5.3: Sequences used in the simulation and hardware implementation 44
Table 5.4: SNR and chip energy values 45
Table 5.5: FPGA utilization for preamble detection emulator 49
Table 5.6: FPGA utilization for the prototype of the preamble detection system 52

List of Figures

Figure 2.1: Constellation diagram for BPSK 4
Figure 2.2: Traditional packet structure 8
Figure 2.3: Presented packet structure 8
Figure 2.4: Block diagram of the preamble generator 9
Figure 2.5: Block diagram of the preamble detector 10
Figure 2.6: Block diagram of the preamble receiver 10
Figure 3.1: Simulator block diagram 15
Figure 3.2: Transmit signals (a) first 5 symbols of the preamble sequence (b) first 5 symbols of the differentially-encoded preamble sequence (c) length 16 spreading sequence (d) differentially-encoded preamble sequence multiplied by the spreading sequence (e) zoomed in view of 5 chips (f) the real component of the 5 chips after it has been sampled with amplitude of $\pm \boldsymbol{Es}$ (g) the imaginary component of the 5 chips after it has been sampled with amplitude of zero 18
Figure 4.1: Block diagram of the SignalMaster C67X FPGA [38] 24
Figure 4.2: Block diagram of the SM-ADAC Master II with programmable gain amplifiers (PGA) [39] 24
Figure 4.3: 16-bit shift register chain used in the preamble generator 26
Figure 4.4: Block diagram of the 16-bit data generator for the transmitter 27
Figure 4.5: Transmitter controller module state diagram 28
Figure 4.6: Block diagram of the chip correlator with modified adders/subtractors 30
Figure 4.7: Modified Adder/Subtractor 31
Figure 4.8: Chip correlator example with spreading sequence equal to 1100111101000110 32
Figure 4.9: Symbol storage module 33
Figure 4.10: Block diagram of the differential decoder 34

Figure 4.11: Block diagram of the preamble symbol correlator and comparator 35
Figure 4.12: Block diagram of payload extractor 36
Figure 4.13: Block diagram of the noise generator for the simulator 37
Figure 4.14: Plot of probability density function of a Gaussian noise generator that uses twelve 16-bit UDRV with linear y-axis 38
Figure 4.15: Plot of probability density function of a Gaussian noise generator that uses twelve 16-bit 39
Figure 4.16: Plot of probability density function of a Gaussian noise generator that uses 48 16-bit UDRV with linear y-axis 39
Figure 4.17: Plot of probability density function of a Gaussian noise generator that uses 48 16-bit UDRVs with log y-axis 40
Figure 5.1: Simulation test sequence structure 42
Figure 5.2: Synchronous System P_{miss} VS SNR_p – Comparison of P_{miss} performance of a system with B=4 and L=4 to the ideal analytical calculated P_{miss} for threshold values 29, 50 and 100. 46
Figure 5.3: Synchronous System P_{false} VS SNR_p – Illustrates that the measured P_{false} for a finite precision system are significantly different from the analytical P_{false} 47
Figure 5.4: Asynchronous System P_{miss} VS SNR_p – Compares the P_{miss} performance of different system 47
Figure 5.5: Asynchronous System P_{false} Vs. SNR_p– Compares the P_{false} performance of different system configurations for threshold values 29, 50 and 100..... 48
Figure 5.6: Synchronous System P_{miss} VS Frequency Offset – Illustrates the effects frequency offset on P_{miss} for different system configurations. SNR_p is set to 16dB..... 48
Figure 5.7: Synchronous System and Hardware comparison P_{miss} VS SNR_p – Compares the P_{miss} performance of the hardware with the simulation results..... 50
Figure 5.8: Synchronous System and Hardware comparison P_{false} VS SNR_p – Compares the P_{false} performance of the hardware with the simulation results 51
Figure 5.9: Packet detection FPGA board setup 52
Figure 5.10: The transmitter (top) and receiver (bottom) were implemented on separate Lyrtech SM-C67X FPGA platforms. A black SMA cable carries the baseband signal from the transmitter to the RF link. From the RF link a gray SMA cable carries the baseband

signal to the receiver and the oscilloscope. A screen shot of a received baseband signal is shown on the oscilloscope. 53
Figure 5.11: Picture of the radio link comprised of 2 MAX2837 RF transceivers evaluation kits. The baseband signal from the transmitter FPGA is sent to the RF transmitter board (right) through the black SMA cable. The receiver board (left) sends the baseband signal to the receiver FPGA through the gray SMA cable. 53

List of Symbols

c_i	Preamble Sequence
a_i	Differentially-Encoded Preamble Sequence
θ_k	Phase Angle
ϕ	Carrier Phase
E_s	Sample Energy
L_b	Length of the Spreading Sequence
W	Length of the Preamble Sequence
b	Spreading Sequence
d	Chip Sequence
η	Final Correlated Value
G	Threshold Value
$\hat{d}$	Estimated Chip Sequence
$\widehat{c_i}$	Estimated Preamble Sequence
$\widehat{a_i}$	Estimated Differentially-Encoded Preamble Sequence
T_c	Chip Timing Duration
I	In-Phase Component of the Signal
Q	Quadrature Phase Component of the Signal
P_{miss}	Probability of Missing a Preamble
P_{false}	Probability of False Detection of a Preamble
B	Number of Bits to Represent the Input Signal
L	Upper and Lower Limit of Values for the Signal
R_{full}	Number of Consecutive Samples Required to Perform a Full 2π Rotation
Δ	Frequency Offset
E_c	Chip Energy
S_c	Number of Samples Per Chip

List of Abbreviations

AWGN	Additive White Gaussian Noise
PSK	Phase Shift Keying
BPSK	Binary Phase Shift Keying
CDMA	Code Division Multiple Access
DSSS	Direct Sequence Spread Spectrum
MAC	Medium Access Control
SNR	Signal-to-Noise Ratio
LR-WPAN	Low-Rate Wireless Personal Area Networks
SYNC	Synchronization
DAC	Digital-to-Analog Converter
ADC	Analog-to-Digital Converter
ASIC	Application-Specific Integrated Circuits
PNG	Pseudorandom Number Generator
RF	Radio Frequency
UDRV	Uniformly-Distributed Random Variables
TP	True Positive
FP	False Positive
TN	True Negative
FN	False Negative
PLL	Phase Locked Loop
PGA	Programmable Gain Amplifier
LUT	Look-Up Table
ASK	Amplitude Shift Keying
ISI	Intersymbol Interference

Chapter 1
Introduction

Data is the foundation on which knowledge is built. Our modern society invests a lot of time and money to gather, process and securely store data. Gathering enough useful data can be a very painstaking task. With the help of some technologies the collection of data can be an automated process. One such technology is Wireless Sensor Networks (WSNs) [1].

1.1 Wireless Sensor Networks

Wireless sensor networks are spatially distributed sensing nodes that communicate with one another to accomplish a certain task. These networks have caught the attention of many computer science and telecommunication researchers. They offer unique problems in coordination, communication and integration.

The development of WSNs has enabled researchers in many different fields to gather data in a manner that was once impossible. Companies and industries also use WSNs to monitor and control their facilities. WSNs have found their way into a wide range of applications and systems, each having their own unique requirements and characteristics.

1.2 Wireless Sensor Node

Wireless sensor networks are composed of nodes or “motes”. A mote is a small device that contains a wireless transceiver, a processor and at least one sensor to measure anything from motion to temperature to electrochemical signals produced by neurons inside the brain.

Often WSNs operate without the presence of a centralized base station. In such a setting, synchronized communication between nodes becomes very difficult. It is much more effective to use asynchronous random-access communication techniques. A

random-access system allows nodes to easily transmit packets at arbitrary times. The problem arises at the receiver, where the packet must be detected. To enable the receiver to extract the random data, a known sequence called a preamble, is inserted in front of the packet. With a preamble the receiver only has to search for a known sequence.

Typically deployed wireless nodes have low data rates and only transmit or receive data intermittently with low duty cycles. Therefore these motes can usually enter an inactive or "sleep" state, where they only have to listen to the channel for packets. These wireless nodes are often battery-powered, and consequently have limited operational lifetime. With the packet detector always on, it is imperative to design a detection system with strong emphasis on low-power consumption. The thesis describes the first implementation of a low-power, asynchronous, random-access packet detection algorithm first introduced in [2] and [3].

1.3 Thesis Overview

This thesis is organized as follows. Chapter 2 starts by reviewing the communication techniques used in this thesis. The next section describes the related work in this field and compares them to the work in this thesis. This chapter also provides a description of the preamble detection algorithm. The last section reviews the effects of signal pulse shaping.

Chapter 3 describes the simulation performed on the preamble detection algorithm and the system's parameters.

Chapter 4 describes the implementation of the preamble detection system on an FPGA platform. An overview of the design is given and is followed by a detailed description of the transmitter, receiver and noise generator.

Chapter 5 is the last part of the main body. This chapter presents the results gathered from the simulations and FPGA implementation.

Chapter 6 provides a conclusion and suggestions for future work.

Chapter 2
Background

The packet detection system implemented in this thesis employs two communication techniques: differential coding [4] and direct sequence spread spectrum (DSSS) [5]. Sections 2.1 and 2.2 provide an overview of these two subjects. A review of relevant research is presented in Section 2.3. Sections 2.4 and Section 2.5 provide a description of the packet structure and the preamble detection algorithm used in this thesis. Section 2.6 describes the effects of signal pulse shaping when used in digital communications.

2.1 Differential Coding

Differential encoding is classified under a group of coding schemes that use memory. This means the output signal of the encoder depends not only on the current input symbol, but also on the previous output. The encoder scheme for differential coding is given in Equation (2.1). Here c_i is the input signal to the differential encoder and a_i is the differential output. To decode a differentially coded sequence Equation (2.2) is used. In Equation (2.2) a_i is the differentially encoded input to the decoder and $\boldsymbol{c_i}$ is the decoded output [4].

$$\boldsymbol{a_i = a_{i-1} \oplus c_i} \tag{2.1}$$

$$\mathbf{c}_{\boldsymbol{i}} \boldsymbol{= a_{i-1} \oplus a_i} \tag{2.2}$$

Differential coding is a technique used to offset the effects of phase ambiguities introduced by the communication channel and oscillator mismatch. Rather than mapping information directly onto individual symbols, differential encoding conveys information in the phase difference between two successive symbols. For example, in

differential binary-phase modulation, a 1 is transmitted as a 180° phase shift relative to the previous signal and a 0 is transmitted as a zero phase shift relative to the previous signal. When implemented as a BPSK system, there is a 180° of separation between the two constellation points in the complex plane as seen in Figure 2.1. This separation allows for ± 90° phase ambiguity in the demodulated signal [5].

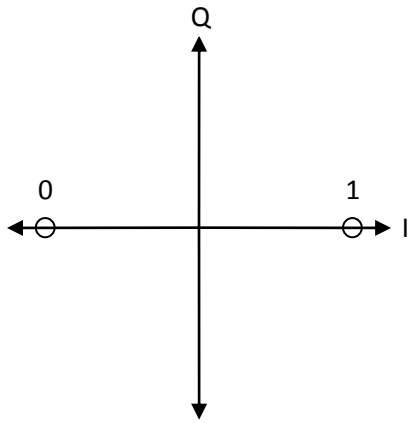

Figure 2.1: Constellation diagram for BPSK

Another feature of differential coding is that the demodulator does not require knowledge of the carrier phase because the phase of each received signal is compared relative to the phase of preceding signal. To illustrate this, take any two sequential demodulated outputs as shown in the following complex numbers.

$$\boldsymbol{y_k} = \sqrt{\boldsymbol{E_s}}\, \boldsymbol{e^{j(\theta_k - \phi)}} + \boldsymbol{n_k} \qquad (2.3)$$

$$\boldsymbol{y_{k-1}} = \sqrt{\boldsymbol{E_s}}\, \boldsymbol{e^{j(\theta_{k-1} - \phi)}} + \boldsymbol{n_{k-1}} \qquad (2.4)$$

In Equations (2.3) and (2.4), $\boldsymbol{\theta_k}$ is the phase angle of the received signal and $\boldsymbol{\phi}$ is carrier phase. After performing differential decoding by multiplying these two signals together, the resulting complex number is

$$\boldsymbol{y_k y_{k-1}^*} = \boldsymbol{E_S}\, \boldsymbol{e^{j(\theta_k - \theta_{k-1})}} + \sqrt{\boldsymbol{E_s}}\, \boldsymbol{e^{j(\theta_k - \phi)}} \boldsymbol{n_{k-1}^*} = \sqrt{\boldsymbol{E_s}}\, \boldsymbol{e^{j(\theta_{k-1} - \phi)}} \boldsymbol{n_k} + \boldsymbol{n_k n_{k-1}^*}. \qquad (2.5)$$

From Equation (2.5), in the absence of noise, the resulting signal would have a phase offset of $\boldsymbol{\theta_k - \theta_{k-1}}$ [5].

Differential coding eliminates the need for a phase lock loop (PLL) circuit to recover the carrier frequency, thereby reducing circuit complexity and lowering the power consumption. The low power capabilities of a differential coded system has been shown in [6], where a differential phase shift keying system was implemented that operates below 1mW at rates up to 100 kb/s.

2.2 Direct Sequence Spread Spectrum

Spread spectrum is a technique used in telecommunications to distribute the energy of a signal over a much wider signal band than the original signal. DSSS is a technique to perform spreading, where the information signal is multiplied by a pseudorandom sequence of 1s and -1s called chips. This pseudorandom sequence of chips is called the spreading sequence and must be known at both the transmitter and receiver. The resulting sequence has a higher rate than the original information signal. The longer the spreading sequence, the wider the band that is occupied by the resulting signal. At the receiver the signal is multiplied by the known spreading sequence and the original signal is recovered. This process is called despreading.

Signal spreading has many advantages over a non spread signal. Having the signal energy spread over a wider band is beneficial because the signal becomes more tolerant to unintentional or intentional jamming [5]. Also, with the use orthogonal spreading sequences, multiple users are able to share the same band. When decoding a particular signal, the signals from other users are observed as noise [5]. Furthermore, DSSS signals also have the advantage of being stealthy, since the resulting chip signal can be below the noise floor [5]. Without knowledge of the spreading sequence, these signals are difficult to decode or even to detect. In this thesis the noise tolerance and the channel sharing capabilities of DSSS are exploited [5].

2.3 Related Work

Low power communication between wireless nodes is a fundamental issue that is continually being researched and developed. The current dominant communication interface that has been adopted by the wireless sensor network research community is the 802.15.4-2006 Zigbee Standard [7]. Prior to the standardization of this low rate

wireless personal area network LR-WPAN, a wireless embedded system group at the University of California, Berkeley made significant contributions to the development of wireless sensor nodes [8] [9].

This wireless embedded systems research center under the guidance of David Culler has produced several generations of wireless motes, most of which have been developed using off-the-shelf components such as microprocessors and radio frequency (RF) transceivers. The mote project involved an interdisciplinary group, with people specializing in areas such as communications, sensors and signal processing. One of their main contributions was developing TinyOS, an open-source operating system designed for wireless sensor nodes to handle intensive concurrent operations with minimal hardware requirements [10] [11]. TinyOS enabled the motes to adopt a generalized wireless sensor node architecture. This architecture does not rely on specific radio or processor technology; instead it arbitrates the relay of data between computation and communication. The earlier generation mote, Mica, used TinyOS and a hardware accelerator to offload wireless communication. The hardware accelerator is a flash memory based microcontroller and is connected to the main processor through a serial peripheral interface (SPI). Its role is to search for the start symbols to extract packets [9] [8]. The Mica used a short-range amplitude shift keyed based radio transceiver, RFM TR1000 [12]. Several generations of this mote were developed and they incorporated faster microprocessors and better RF modules [13] [14]. Eventually Spec was developed, which was a complete wireless sensor node system all integrated on a single integrated circuit [9].

The Spec integrated RISC CPU, SRAM, RF transmitter, communication hardware accelerator and ADC all in one package. This mote had a fairly simple packet detection system. The communication hardware accelerator searches for a specific start sequence, which can be configured up to a length of 24 bits. In order to detect the start sequence, the communication hardware accelerator samples the channel at twice the rate of the start symbols. When a start sequence is detected, a signal is sent to the microprocessor and the packet data is extracted. During the early stages of development of these motes and TinyOS, a standard communication protocol to address the needs of wireless sensor nodes had not yet been created. However, soon

after the completion of the Spec project at Berkeley, the Zigbee data link layer, which operates over top of the IEEE 802.15.4, was standardized [15].

By 2003, Zigbee had filled a void in the communication protocol standards. It provided a communication platform with a low data rate and low power, with the addition of providing secure networking. The standard was promoted by an alliance of 25 firms and many research groups began developing Zigbee-compliant transceivers as it quickly gained popularity [16] [17] [18] [19].

Zigbee is now the most widely used communication protocol for wireless sensor networks [20]. Many commercially available motes use Zigbee as their primary communication interface [21] [22] [23] [24]. There is a wide selection of Zigbee compliant RF-ICs [25] [26] [27] [28] [29] [30] [31]. These RF-ICs have a large range of features; some are just simple transceivers, while others have integrated microprocessors, memory and ADCs. The following section introduces the packet structure used in this thesis and compares it to the one used in the IEEE 802.15.4-2003 standard. A summary of some of the motes developed at Berkeley and some commercially available nodes are given in Table 2.1.

Table 2.1: Summary of some wireless sensor nodes.

Module Name	Communication Interface	Frequency Band	Data Rate	Note
Mica [15] [8]	ASK and TinyOS	916.5 MHz.	50 Kbps	Uses RFM TR1000 [12] and TinyOS
Mica2 [15] [14]	ASK and TinyOS	868/916 MHz	76 Kbps	Uses ChipCon CC1000 [13] and TinyOS
Spec [15] [9]	ASK and TinyOS	900 MHz	100Kbps	ASIC
Miniaturized Mote [32]	IEEE 802.15.4 Zigbee	2.4-2.4835 GHz		ChipCon CC2430
MicaZ [23]	IEEE 802.15.4 Zigbee	2.4-2.4835 GHz	250Kbps	Mica with ZigBee
TelosB [24]	IEEE 802.15.4 Zigbee	2.4-2.4835 GHz	250Kbps	Open Source Platform
Imote2 [22]	IEEE 802.15.4 Zigbee	2.4-2.4835 GHz	250Kbps	Advanced wireless sensor node platform
IRIS [21]	IEEE 802.15.4 Zigbee	2.4-2.4835 GHz	250Kbps	Large Scale Sensor networks (1000+ nodes)

2.4 Packet Structure

A traditional packet structure such, as the one used in IEEE 802.15.4 standard LR-WPANs, is shown in Figure 2.2. In this packet structure the preamble facilitates the detection of the packet. It is followed by a synchronization header (SHR), which is used to synchronize and lock onto the bit stream after the packet has been detected.

Figure 2.2: Traditional packet structure

Figure 2.3: Presented packet structure

Figure 2.3 shows the packet structure used in this thesis conceived by a research group in the High Capacity Digital Communications Lab at the University of Alberta [2] [33]. Note how it is similar to the one used in the IEEE 802.15.4 standard. The packet detector in the system is able to perform detection and synchronization concurrently, and therefore an SHR is not required in the packet structure.

The physical layer header (PLH) contains information to facilitate the extraction of the payload. This information usually consists of frame length and packet ID. The PLH is followed by the payload, which contains the medium access control (MAC) information and the data. The PLH and payload can take on a large variety of data formats and they do not necessarily have to follow the same format of the preamble section. Table 2.2 summarizes and compares the packet format presented in this thesis with the packet format used in [7]. This thesis only deals with the detection of the preamble and the synchronization to the bit stream.

Table 2.2: Packet structure comparison

	802.15.4 BPSK Packet Format	Presented Packet Format	Description
Preamble	32 symbols	40 symbols	Allows the device to detect an incoming packet
Synchronization Header	8 symbols	None	Allows the device to synchronize and lock onto the data stream
Physical Layer Header	8 symbols	Undetermined	Contains frame information
Payload	Variable	Variable	Carries the data

2.5 Preamble Detection Algorithm

Each symbol c_m in the preamble sequence of length W is given a value of 1 or -1 according to a given maximum length sequence [4]. The preamble sequence c is then differentially encoded to form sequence a. Next DSSS is performed on each symbol in a by up sampling L_b times and multiplying by a pseudorandom spreading sequence b. The result is a chip sequence d of length $W \times L_b$. A block diagram of how the preamble sequence is generated is given in Figure 2.4. In this thesis the length of the preamble sequence W is 40 and the length of the spreading sequence L_b is 16. These two values are chosen so that the preamble length is similar to that of the packet structure in the IEEE 802.15.4 standard [7].

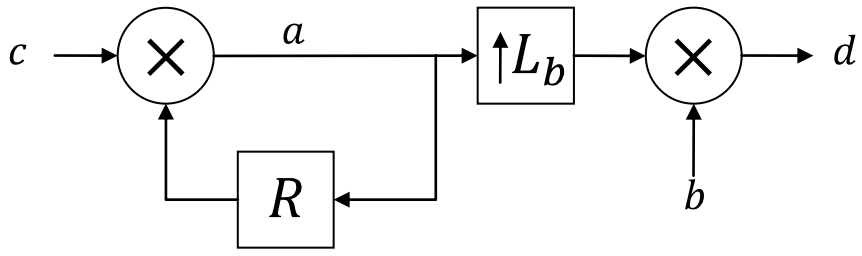

Figure 2.4: Block diagram of the preamble generator

In the receiver a threshold-based correlator is used to detect preambles. The received baseband signal $\hat{d}$ contains both in-phase I and quadrature Q components. These two signals are correlated and despread with the known spreading sequence b. The resulting estimated symbol sequence $\hat{a}$ is differentially decoded to give the

estimated preamble sequence $\hat{c}$. The resulting estimate of preamble sequence $\hat{c}$ is then correlated with the known preamble sequence c. The final correlated value η is compared to a threshold G to determine if a preamble has been detected. When a preamble has been detected the resulting correlation peak is used to synchronize the bit stream. Figure 2.5 shows a block diagram of the preamble detector.

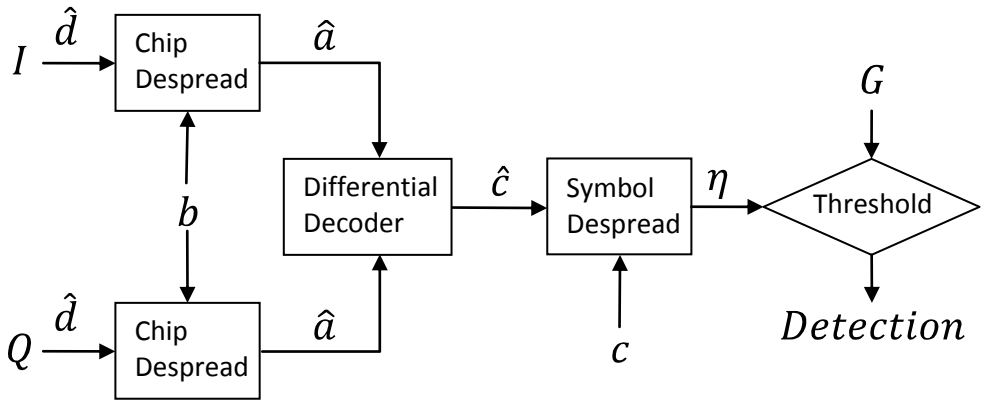

Figure 2.5: Block diagram of the preamble detector

To deal with sampling timing issues, two sample points are taken for each chip. The two points are separated by $T_c/2$, that is half the duration of a chip. This ensures that even in the worst case a sufficient sample point is taken [33]. Since it is not known which sample point is better, both points are sent to two individual streams for correlation. A preamble detection occurs when either one of the streams detects a preamble. Figure 2.6 shows the two streams to correlate the two samples coming from the sampler and analog to digital converter. To provide samples for both detection streams, the sampler and ADC operates at 2 times $1/T_c$.

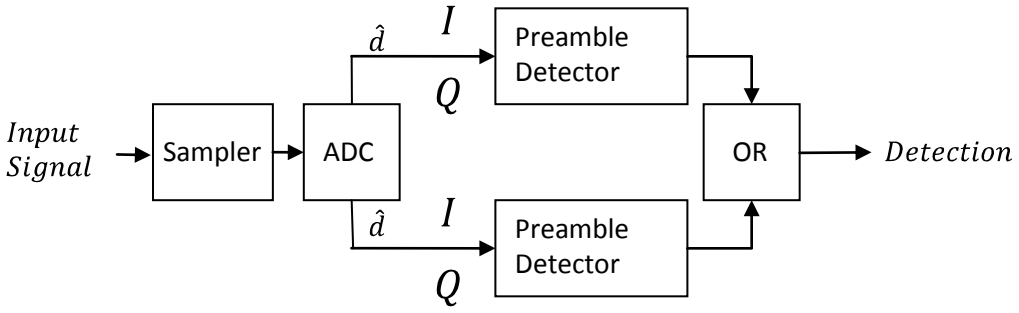

Figure 2.6: Block diagram of the preamble receiver

2.6 Pulse Shaping

In digital communications, bits or symbols are represented using pulses. By manipulating the pulse's shape we can adjust the bandwidth required to transmit a digital signal. Additionally, pulse shaping can reduce intersymbol interference (ISI) [34].

The Nyquist criterion addresses ISI. It stipulates that a pulse shape must pass through zero at time $t = kT$, where $k = \pm 1, \pm 2, \pm 3, \ldots$ and T is the bit duration. A pulse shape with this property is referred to as a Nyquist pulse. Using a Nyquist pulse reduces ISI since at the sampling interval there is zero contribution from neighboring pulses [34].

There are many different pulse shapes that satisfy the Nyquist criterion for ISI, each with their own unique characteristics. The most common pulse shapes are provided in Table 2.3 and Figure 2.7 [34]. The rectangular pulse is simple and easy to generate. However, its spectrum has several high sidelobes which can interfere with signals in adjacent frequency bands. The sinc pulse has a frequency response in which all the energy is confined in a very narrow region of the spectrum. However, the sinc pulse is susceptible to ISI when sampling instants are not perfectly aligned to T, since there are significantly large sidelobes in the time domain signal. The raised cosine pulse has a roll-off factor that allows users to suppress the sidelobes in the time domain. Furthermore, the raised cosine pulse has a narrow frequency response.

At this current development stage we are not concerned with spectral efficiency. Therefore, we have decided to implement a system using a simple rectangular pulse.

Table 2.3: Common pulse shapes

Rectangular Pulse	$\frac{1}{2T}$ $0 \le t \le T$
Sinc Pulse	$\frac{\sin\left(\frac{\pi}{T}t\right)}{\frac{\pi}{T}t}$
Raised Cosine Pulse	$\frac{\sin\left(\frac{\pi}{T}t\right)}{\frac{\pi}{T}t}\left(\frac{\cos\left(\frac{\pi}{T}\alpha t\right)}{1-\left[\frac{4\alpha}{2T}t\right]^2}\right)$ $0 < \alpha < 1$

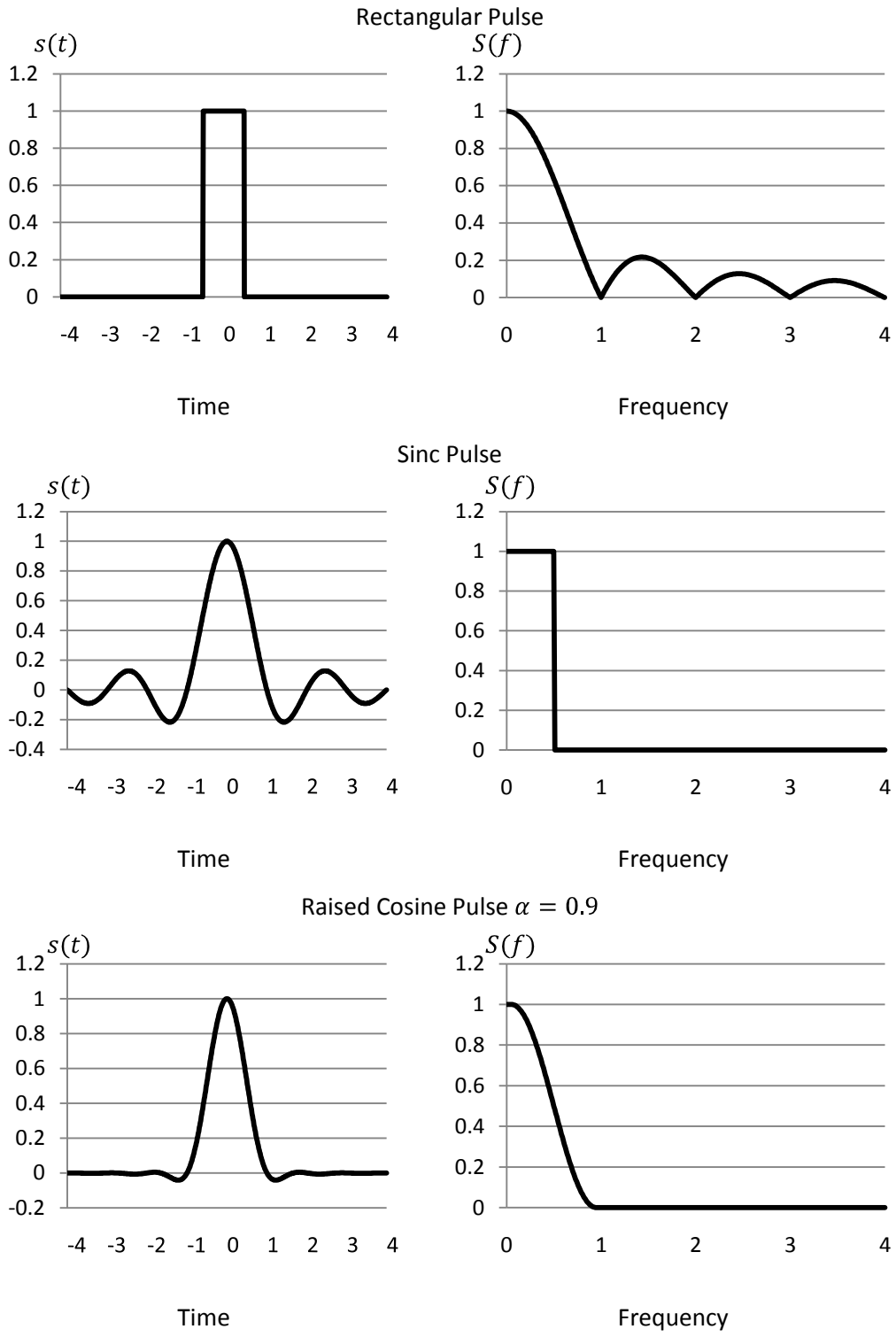

Figure 2.7: Common pulse shapes and their spectra. The x-axis is normalized to bit duration and bit rate

Chapter 3
Simulations

Performance of the system was measured with analytical studies and simulations in [33]. The results formulated in [33] were derived using high-precision floating-point numbers. To understand the effects of having bit-limited fixed-point operations, such as those available on a hardware implementation, further research is required.

Since the targeted platform for this packet detection system is a wireless mote, it is very desirable to build a system which consumes as little energy as possible. To build such a system requires limiting the complexity and size of logic circuits. One effective way to reduce size of logic circuits is to use fewer bits to represent the signals. Signals with a low bit precision require smaller arithmetic circuits, such as adders and multipliers. However, using fewer bits results in data loss, which in turn reduces the performance of the system. Simulations are used to determine the tradeoffs between hardware complexity and system performance. The following sections provide a detailed description of the system parameters and the simulations used to study the effects of quantization on the preamble detection algorithm.

3.1 System Parameters

There are many ways the preamble detector system can be configured. This section describes the different parameters that influence the performance of the system. Before explaining the parameters of the system, this section first defines how performance is measured.

There are two metrics that measure the performance of the preamble detector system: the probability of missing a packet P_{miss} and the probability of a false detection P_{false}. Reducing P_{miss} is more crucial, since allowing a packet to pass undetected can lead to the loss of data.

Occasionally, when the system falsely detects a packet when one does not actually exist, it would result in an extracted payload that does not make any sense. This mistaken payload can be expected to be easily recognized by a high control layer of the communication system and simply be discarded. False detections are a nuisance, but do not usually degrade the overall reliability of the system. The only circumstance where P_{false} poses a problem is when it becomes high enough such that the system starts randomly guessing if a packet is present or not. For this to happen, the P_{false} rate needs to be approximately 50%. These two performance measures, P_{false} and P_{miss}, are directly related to correlation peak threshold parameter G.

A correlated peak for a certain sequence of chips and symbols is compared to the threshold parameter G. If a correlated peak is greater than G, the detector responds with a signal to indicate that a preamble has been detected and a payload is ready to be extracted. The P_{false} can be reduced by increasing the value of G, but in doing so, the likelihood of missing a preamble increases. On the other hand, if we lower the value of G, the P_{false} increases and P_{miss} decreases. This strong relationship between the threshold and system performance, allows the P_{false} and P_{miss} to be easily regulated by adjusting the single parameter G.

The hardware implementation is strongly influenced by the number of bits B used to represent the incoming signal. Using fewer bits can decrease the size of the arithmetic units required in the hardware implementation of the preamble detector. This, however, comes at the cost of decreasing the number of different values the preamble detector can receive. Having fewer values to represent the signal decreases the precision of the data entering into the detector, which can lead to degraded system performance. Another parameter that is closely related to B is L, the upper and lower limit of values that the incoming signal can take. Together these two parameters determine the bin width of the ADC using Equation (3.1). For a fixed B, a larger L allows the system to represent a wider range of numbers, but with less precision. The converse is also true, that a smaller L causes the system to represent a narrower range of numbers, but with increased precision. The combination of these two parameters B and L, significantly influences the performance of the system as well as the hardware implementation.

$$Bin\ Width = \frac{2L}{B^2} \tag{3.1}$$

The final parameters for the preamble detection system simulator are related to the distortion caused by the emulated channel. The channel emulator can be adjusted using the following parameters: signal-to-noise ratio (SNR), frequency offset and frequency drift. The noise generator used in the simulation produces additive white Gaussian noise (AWGN). The noise power is constant, so to change the SNR value the energy of the samples is adjusted. The simulator is also able to add a random frequency shift to the transmitted signal. There are two parameters to introduce frequency distortion to the signal. The first is a phase offset, which shifts each sample by a given phase increment between 0 and 2π. The second parameter introduces a frequency offset which causes a continuous frequency phase rotation in the signal.

Each of the parameters described in this section is used to perform a variety of simulations to determine a good configuration for the hardware implementation. The following section describes the simulator in detail.

3.2 Bit-true simulation

A bit-true simulation is a program that emulates the same bit operations that occur in hardware. The analytical and simulation results from [33] provide ideal performance curves using high-precision floating-point numbers. The effects of converting from floating-point numbers to bit-limited fixed-point quantized numbers are investigated using bit-true simulations. These simulations provide a better picture of how a system performs when implemented in hardware. The simulation is built in ANSI C because C programs can be easily compiled and executed quickly on different platforms. There are four main components in the preamble detector simulator: channel emulator, transmitter, receiver and statistics monitor. These components are shown in the block diagram in Figure 3.1 and are explained in detail in the following sections.

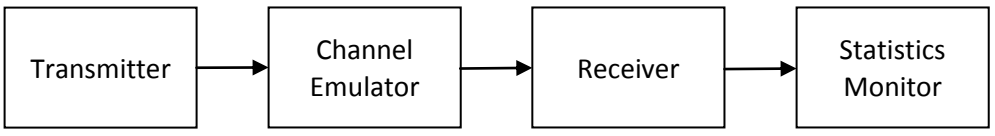

Figure 3.1: Simulator block diagram

3.2.1 **Channel Emulator**

The channel emulator consists of two main components: the AWGN generator and the frequency shifter. The AWGN generator is implemented using the Box-Muller transform [35]. The Box-Muller transform creates two independent standard normally-distributed random numbers using uniform random numbers. To generate the uniform random numbers we use the Mersenne Twister uniform pseudorandom number generator [36].

The frequency shifter is implemented with Equations (3.2), (3.3) and (3.4). Equations (3.2) and (3.3) produces the real and imaginary components of the frequency shifting function $\boldsymbol{F(n)}$, where $\boldsymbol{n}$ is an index value from 1 to the number of samples in the transmitted sequence. The two parameters that are used to perform the frequency shifting are Δ and $\boldsymbol{R_{full}}$. The frequency offset Δ is a constant parameter that can be set to any value ranging from 0 to 2π. To simulate frequency drift the simulator uses $\boldsymbol{R_{full}}$, which is the number of consecutive samples required to perform a full 2π rotation in the transmit signal. The frequency shift is applied by multiplying the transmit signal $\boldsymbol{T(n)}$ by the frequency shift function $\boldsymbol{F(n)}$, as shown in Equation (3.4).

$$\boldsymbol{Re\{F(n)\}} = \boldsymbol{cos}\left(\Delta + \frac{\boldsymbol{2\pi(n\ mod\ R_{full})}}{\boldsymbol{R_{full}}}\right) \tag{3.2}$$

$$\boldsymbol{Im\{F(n)\}} = \boldsymbol{sin}\left(\Delta + \frac{\boldsymbol{2\pi(n\ mod\ R_{full})}}{\boldsymbol{R_{full}}}\right) \tag{3.3}$$

$$\boldsymbol{S(n) = F(n) \times T(n)} \tag{3.4}$$

3.2.1 **Preamble Generator**

The preamble generator is implemented in three components: differential encoder, spreader and up sampler. The preamble sequence c and the spreading sequence b are filled with a random set of 1s and -1s.

The differential encoder takes the preamble sequence c and creates the differential-coded sequence a using Equation (2.1). To get the sequence started the first value in a is set to the first value c. The first five bits of c and a are shown in Figure 3.2a and Figure 3.2b, respectively.

The next step is to spread the sequence a by b. At this point all values are either 1 or -1. An example of a length-16 spreading sequence is shown in Figure 3.2c. In Figure 3.2d the first five symbols of the differentially-coded spreading sequence in Figure 3.2b is spread with the spreading sequence in Figure 3.2c. Next we simulate the signal as it moves into the analog domain. To achieve this we up sample each chip by a factor of S_c. In this thesis we have chosen S_c to be 8. Figure 3.2e shows a subset of the chips from Figure 3.2d. Each sample is now represented by both a real and an imaginary component. The real component of each sample is multiplied by the sample energy value E_s, which is given in Equation (3.5) and the imaginary component of each sample is set to zero. Figure 3.2f and Figure 3.2g shows the real and imaginary components of the sampled chip sequence in Figure 3.2e. The result is our transmitted signal vector with 640 chips each with 8 samples and with value at $\pm E_s$.

$$E_s = \sqrt{\left(\frac{2 * E_c}{8}\right)} \tag{3.5}$$

The transmit signal then enters into the channel emulator where noise and frequency distortion can be added. After the signal leaves the channel emulator, the preamble sequence is completely buried in the noise. The next step is for the signal to be decoded by the preamble detector.

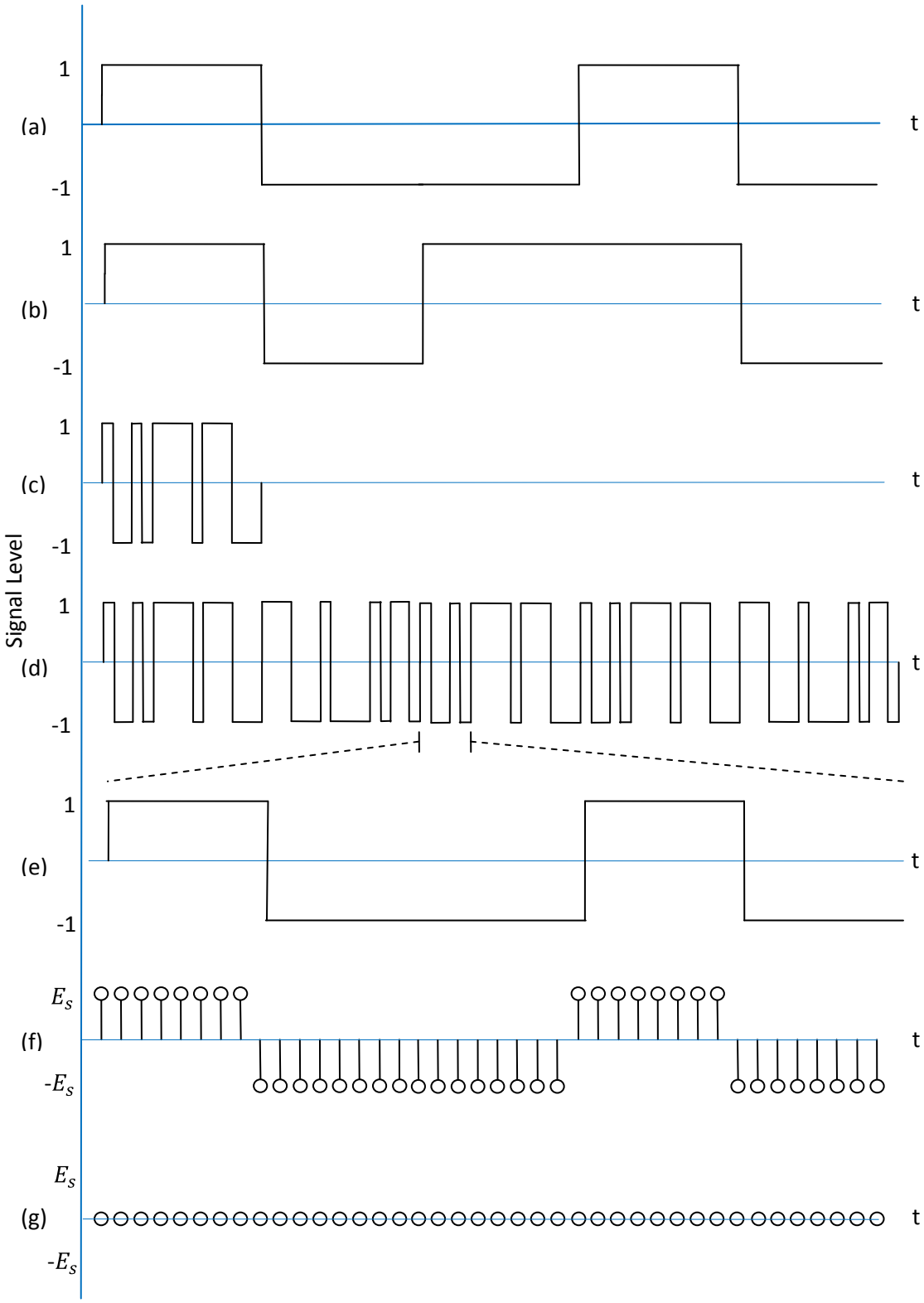

Figure 3.2: Transmit signals (a) first 5 symbols of the preamble sequence (b) first 5 symbols of the differentially-encoded preamble sequence (c) length 16 spreading sequence (d) differentially-encoded preamble sequence multiplied by the spreading sequence (e) zoomed in view of 5 chips (f) the real component of the 5 chips after it has been sampled with amplitude of $\pm E_s$ (g) the imaginary component of the 5 chips after it has been sampled with amplitude of zero

3.2.3 **Preamble Detector**

The preamble detector's role is to correlate a given window of the transmitted signal with the preamble sequence c and the spreading sequence b. The correlated values are compared to the threshold G to determine if a preamble was present or not.

The first step is to perform a matched filter on the transmitted signal. The matched filter is given in Equation (3.6) . The chip rate is known by the receiver, but the detector has no information that can assist it in picking the sample timing with the optimal signal strength. However, it has been shown in [33] that it is sufficient to perform two concurrent detections on two sample points separated by half the chip interval. Using two sample points yields similar performance to that of a system which samples at the optimal signal strength.

$$\boldsymbol{MF = \sqrt{1/S_c}} \tag{3.6}$$

Before the signal can be correlated, we need to move back to the digital domain by binning the selected received samples. Each sample, when it first enters the binning stage, is compared to the sample limit $\pm L$ which is the maximum/minimum allowed value. Positive sample values that are greater than $+L$, are reassigned to $+L$ and similarly negative sample values that are less than $-L$, are reassigned to $-L$. Before binning can performed, the bin width needs to be determined, which is done using Equation (3.1). After the samples are binned they can be represented digitally by B bits. To represent the digital binary values we assign the sample values to integers between -2^{B-1} to $2^{B-1}-1$ which is the range of a two's complement binary number with B bits. Equation (3.7) is used to perform this mapping from sampled values to integers. After the floating-point sample values are converted to integers, the system can begin to perform correlation.

$$\boldsymbol{Binned\ Value = floor\left\lfloor \frac{Sample\ Value}{Bin\ Width} \right\rfloor} \tag{3.7}$$

The sample values are first correlated with b. To perform this task the simulator takes the first L_b chip samples and multiplies it by b. The results are then summed together to form an estimate of the first symbol estimate. Next the simulator moves on to the next L_b chips to obtain the next symbol estimate. This continues until the system has enough estimated symbols to represent a preamble of length W. The system is now ready to perform differential decoding on the estimated differentially encoded preamble sequence $\hat{a}$. Equation (2.2) is used to perform differential decoding. Since these are complex signals, it is normally required to perform four multiplication operations. However, the algorithm only requires the real component of the signal after the differential decode. Thus the simulation only performs the multiplication required to obtain the real component of the signal; this is shown in Equation (3.8). After differential decoding, the system obtains an estimate of the preamble sequence $\hat{c}$. When performing differential decoding, the first symbol in $\hat{a}$ is skipped since it does not have a preceding symbol. Therefore the length of $\hat{c}$ is one symbol shorter than the length of $\hat{a}$. The next step is to correlate $\hat{c}$ with the known preamble sequence c. This is done by multiplying each value of $\hat{c}$ by c. Since the preamble sequence is of length W and we only have $W-1$ estimated preamble sequences, the system ignores the first value and starts correlating at the second value c. After the multiplications, the products are all added together to obtain a final correlated value η. This value is compared to the threshold value G to determine if a preamble was detected or not.

$$\hat{\boldsymbol{c}} = \boldsymbol{Re}\{\hat{\boldsymbol{a}}_1 * \hat{\boldsymbol{a}}_2\} = \boldsymbol{Re}\{\hat{\boldsymbol{a}}_1\} \times \boldsymbol{Re}\{\hat{\boldsymbol{a}}_2\} + \boldsymbol{Im}\{\hat{\boldsymbol{a}}_1\} \times \boldsymbol{Im}\{\hat{\boldsymbol{a}}_2\} \quad (3.8)$$

Each of the two sample values per chip are correlated in two separate processing streams. Both processing streams perform the same computations on their own sample values. A preamble is declared to be detected if either stream produces a correlation value greater than the threshold G.

3.2.4 **Statistics Monitor**

The statistics monitor module has two main roles. Its first role is to set up and test signals for preambles and the second is to keep track of the statistics. Details about the statistics monitor are given in Section 5.1.

3.3 Chapter Summary

Simulating a design in software is a necessary step when implementing a new communication algorithm in hardware. These bit-true simulations provide a good understanding of how the system performs in hardware. More specifically, simulation results enable us to analyze the tradeoffs between different system parameters. The following chapter describes the hardware implementation of the preamble detection system.

Chapter 4

FPGA Implementation

A hardware implementation brings us one step closer to realizing the detection algorithm in a real system. Field-programmable gate arrays are excellent platforms to develop initial hardware prototypes. Their easy reconfigurability and versatility make them very effective for developing and verifying hardware designs. Although their clock speeds are lower than those of an ASIC design, they are more than sufficient for our system. These reasons have led to the decision of implementing the preamble detection system on FPGAs.

The following section provides an overview of the FPGA platforms and tools used to design the preamble detection system. The main body of this chapter contains a detailed description of each component implemented on the FPGAs.

4.1 Platform and Design Overview

The preamble detection system was implemented on two SignalMaster-C67X FPGA platforms by Lyrtech [37]. Each of these platforms contains a Xilinx XC2V8000 Virtex II FPGA [37]. An onboard microcontroller is used to communicate to the FPGA and other peripherals including an SM-ADAC Master II, which has dual high performance ADCs and DACs with programmable gain amplifiers. Block diagrams of the SignalMaster C67X FPGA platforms [38] and the ADAC Master II [39] are provided in is shown in Figure 4.1 and Figure 4.2, respectively. Also included is an onboard Ethernet controller which allows access to the platform through a local area network.

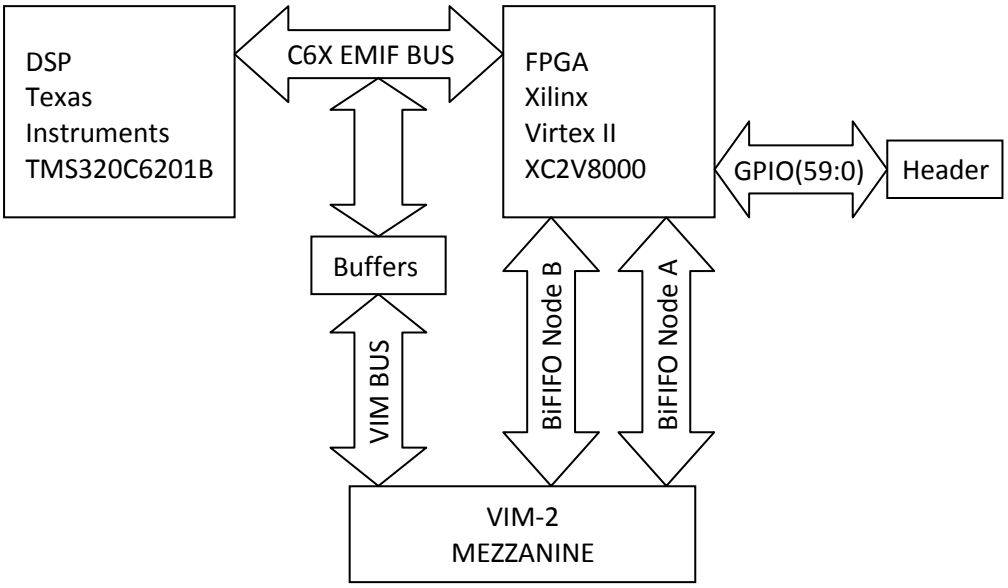

Figure 4.1: Block diagram of the SignalMaster C67X FPGA [38]

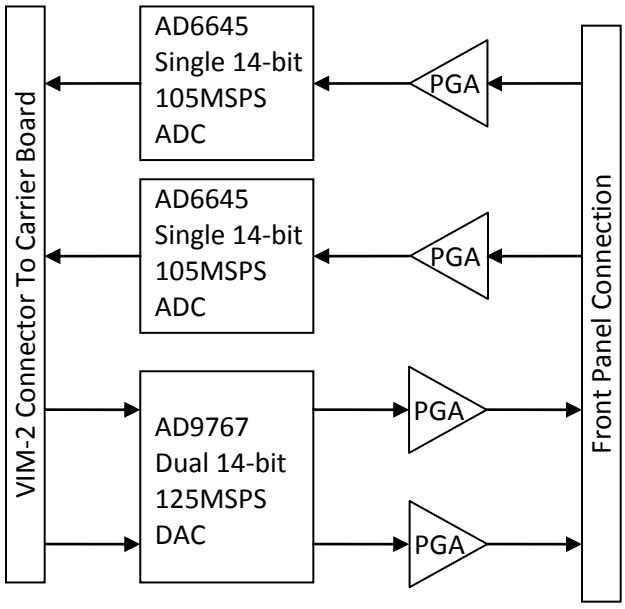

Figure 4.2: Block diagram of the SM-ADAC Master II with programmable gain amplifiers (PGA) [39]

Our implementation consists of 3 main components: transmitter, noise generator, and receiver. The complete system was originally developed on one SignalMaster-C67X platform. It was later split onto two FPGA platforms, one containing the transmitter and the other the receiver and noise generator. With the receiver and transmitter on two separate boards we are able emulate a real asynchronous system with independent clocks. To connect the two systems together we use the ADAC Master II. The ADAC Master II is used to send the baseband signal from the transmitter to the receiver board. This link is configured so that a RF front-end can be added to create a wireless link.

Both the transmitter and receiver board are controlled using a host computer. The host is used to initialize the system, which includes enabling the system and setting the chip energy on the transmitter and the threshold value G on the receiver. Other than the communication line for the I and Q signals, there is no coordination between the transmitter and receiver platform. Therefore in order to collect statistics for the system, we imbedded real data in the payload section of the transmitted signal. The data consists of a simple 16-bit counter which is incremented for every transmitted preamble. The payload is extracted by the receiver and sent to the host computer to be analyzed. Section 5.2 describes how the data is processed.

From the simulation results we have determined that it would be sufficient to implement a system which uses only 2 bits to represent the I and Q signal. The following sections will describe the implementation of a preamble system.

4.2 Transmitter

Our transmitter contains three main components: preamble generator, data generator and control module. The control module produces the transmission signal using the preamble generator and the data generator. These components are discussed in more detail in the following sections.

4.2.1 **Preamble Generator**

The preamble generator is implemented using the shift registers available on the FPGA. Each slice on the FPGA can be configured as a 16-bit shift register with clock enables. Conveniently, our system uses a 16-bit spreading sequence, which allows each symbol to fit perfectly into a slice. This allows for an efficient implementation of the

preamble generator using a chain of 40 16-bit shift registers. The preamble sequence is hard coded into the shift registers when the FPGA is programmed. The output of these chained shift registers is connected to the output of the preamble generator as well as back into the input the shift register chain. This output feedback loop allows the preamble generator to be ready again after 640 bits of the preamble have been shifted out. Figure 4.3 illustrates the 40 16-bit chained shift registers.

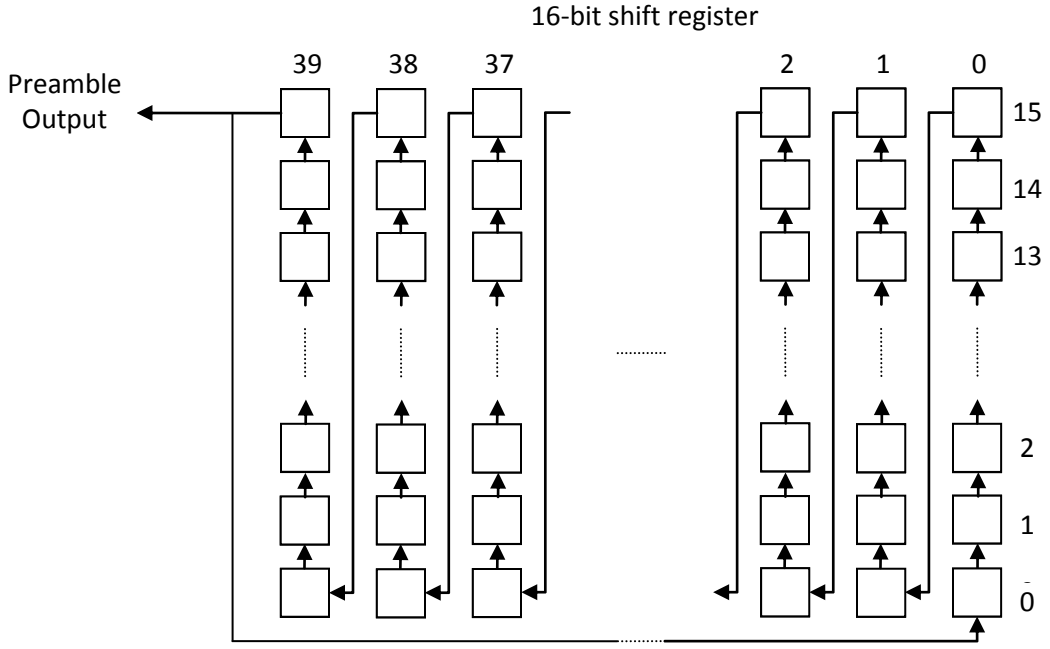

Figure 4.3: 16-bit shift register chain used in the preamble generator

4.2.2 **Data Generator**

The implemented data generator is specifically designed to create a payload that can be extracted by the receiver. Each preamble is attached with a 16-bit number that is embedded into the payload. The number comes from a simple counter, which increments every time a new preamble has been transmitted. Once the 16-bit counter reaches 65535, which is the maximum value, it wraps around back to 0. More detail on how this payload is used to gather statistics for the system is given in Section 5.2 .

The chip energies for the preamble and the data are the same, which makes it very difficult to extract the payload without any error. So to increase the SNR of the payload data, the system uses a simple repetition code. Each bit in the counter is

repeated 4 times, which produces an approximate gain of 6 dB. The resulting 64 symbols are then differentially encoded to compensate for phase offset in the payload.

The data generator module outputs a new chip every clock cycle. Since each symbol is repeated 4 times and there are 16 chips per symbol, the complete 16-bit data payload requires 1024 chips. The spreading sequence b and its complement b' are stored in two different 16-bit shift registers. As the spreading sequence is shifted out, the output of the shift register is fed back to the input to keep the spreading sequence for the next symbol. At every positive clock edge the output chip value is set to either b or b', depending on the input data. A 16-bit multiplexor is used to sequentially select each bit of the input data. The output of the 16-bit multiplexor is sent to a differential encoder. To keep track of which bit to select, a 10-bit counter is used. Since each bit is repeated four times and each symbol has 16 chips, every new data bit requires 64 clock cycles. Thus the select lines for the multiplexor require a 4-bit counter that counts up every 64 cycles. To generate this counter we connect the select lines of the multiplexor to the 10^{th}, 9^{th}, 8^{th} and 7^{th} bit line of the 10-bit counter. After the data generator outputs all 1024 chips for the data section, the counter wraps around back to zero and is ready for the next 16-bit input data. All the components of the data generator are designed to return back to the initial state after all 1024 chips of the data have created. Figure 4.4 is a block diagram of the data generator.

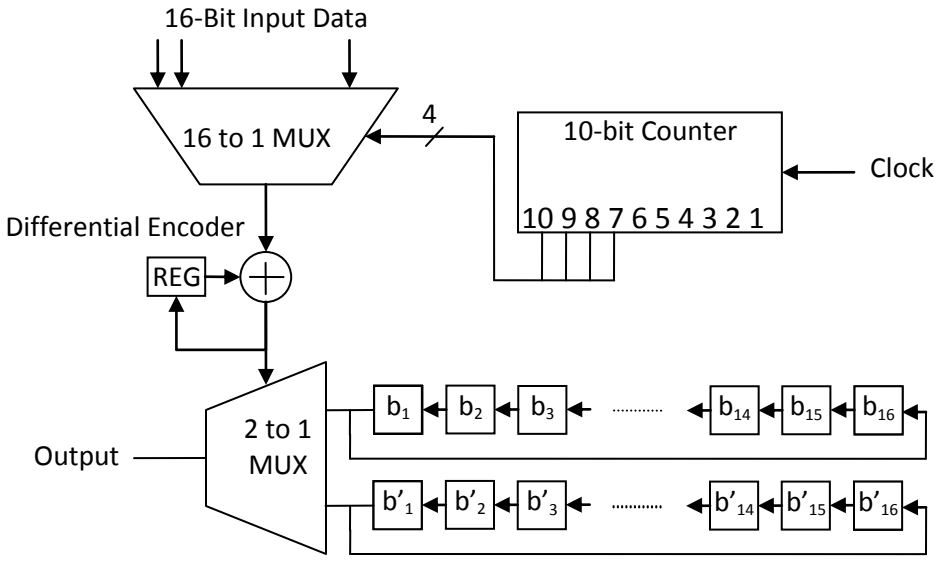

Figure 4.4: Block diagram of the 16-bit data generator for the transmitter

4.2.3 **Control Module**

The role of the control module is to coordinate the generation of a packet. To perform this task the control module uses a finite state machine and several counters. The finite state machine works together with the counters to keep track of the number of the different chips that are transmitted. There are three different types of chips that the counters need to keep track of: preamble, payload and delay. The packet consists of 640 preamble chips and 1024 data chips. Delay chips are added onto the end of the packet to create space between preambles so that the host and the receiver have more time to process statistical data. There is a two-cycle latency to enable and disable the preamble and data generators. For this reason we require delay states in the control module finite state machine to ensure that all of the chips are correctly transmitted. The state transitions are depicted in Figure 4.5 and the description of each state is in Table 4.1. The 16384-chip delay in Figure 4.5 was chosen arbitrarily and can be modified to meet timing constraints.

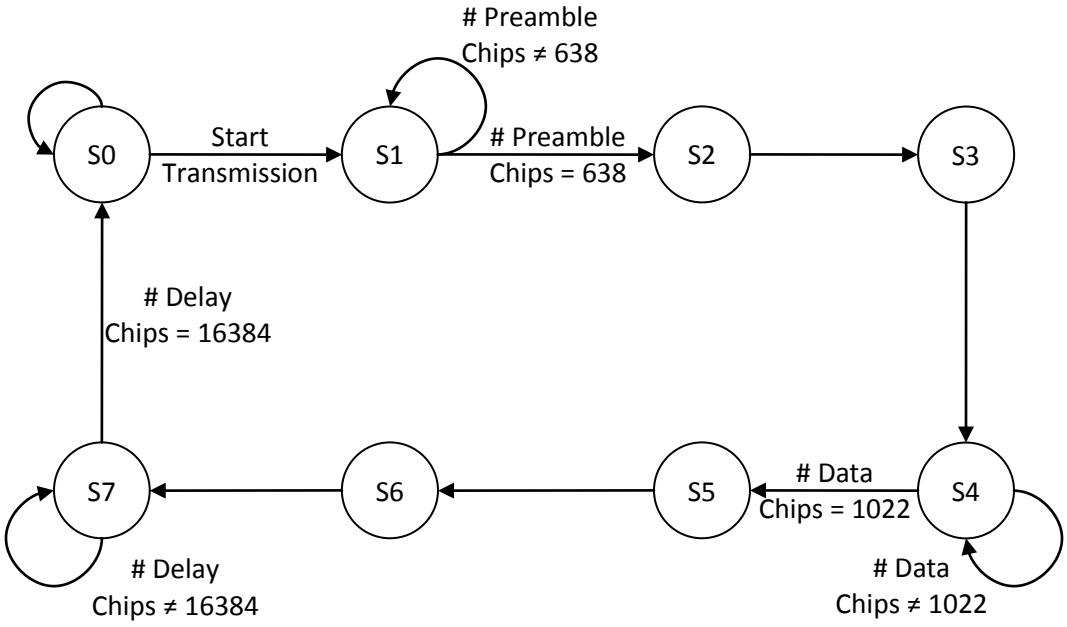

Figure 4.5: Transmitter controller module state diagram

Table 4.1: Transmitter controller module state description

State	Description
S0	Wait in this state until the start transmission signal is received. This is also the default and starting state.
S1	Enable the preamble generator and transmit the preamble chips. Stay in this state until 638 chips are transmitted.
S2	Transmit the second last chip of the preamble. Enable the data generator and disable the preamble generator.
S3	Send out last preamble chip.
S4	Transmit the data chips. Stay in this state until 1022 payload chips are transmitted.
S5	Disable the data generator and transmit the second last payload chip.
S6	Transmit the last payload chip.
S7	Stay in this state until the system transmits 16384 delay chips. Then go back to the wait state.

The generated transmit sequence is sent to the onboard DAC on the ADAC Master II. Each binary chip in the packet is converted to either one of two voltages and sent out on a wire to the receiver platform. The voltage levels can be adjusted using the integrated programmable gain outputs on the ADAC Master II boards. Currently the system only transmits data on the in-phase signal. However the system is designed to utilize both the in-phase and the quadrature component of the signal. The decoding of the packet is described in the Section 4.3

4.3 Receiver

Our packet receiver contains two main components: the preamble detector and the payload extractor. The preamble detector correlates the incoming signal. When a packet has been detected, the preamble detector determines a synchronization point at the start of the data for the payload extractor. With knowledge of where the data begins the payload extractor is able to retrieve the packet data. After the data has been extracted it is sent to the host computer for analysis. The following two sections describe in detail how the preamble detector and payload extractor are implemented.

4.3.1 **Preamble Detector**

The preamble detector can be split into four components: chip correlator, differential decoder, symbol correlator, and threshold comparator. The chip correlator and differential decoder have two different data paths for both the in-phase and

quadrature components of the signal. The differential decoder combines the two signal paths and the resulting values are correlated and compared with a threshold G. A block diagram of the preamble detector is shown in Figure 2.5. With a fully pipelined architecture the preamble detector is able to produce a new correlated value every time a new chip sample is clocked in.

Figure 4.6 shows the flow of data in the chip correlator. At every positive clock edge a new 2-bit chip sample is driven into the 16 stage 2-bit shift registers. As each new data sample is shifted into the registers, a new sequence is created and has to be correlated. Every time a new sequence appears in the shift registers, new values are calculated at each level and stored in the register layer above. This pipelined tree structured design produces a new correlated symbol at every clock cycle. Since there are 4 layers in this tree structure, there is a latency of 4 cycles. Normally before summing the values of the shift registers you need to multiply each sample by the spreading sequence.

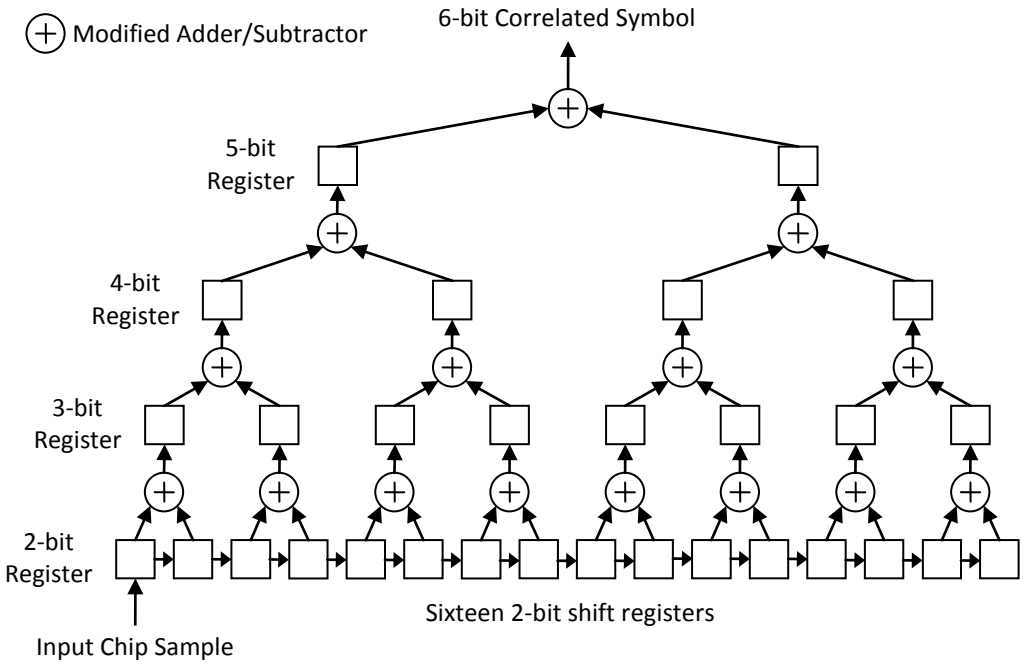

Figure 4.6: Block diagram of the chip correlator with modified adders/subtractors

The process of multiplying the samples by the spreading sequence can be simplified to just changing the sign of the samples whenever the spreading sequence

values is equal to -1, since the system uses BPSK. Changing the sign for a 2's complement number is accomplished by inverting the bits and adding one to the resulting number. To avoid this step we use a modified adder/subtractor, instead of a simple adder, as shown in Figure 4.7. Changing an adder to an adder/subtractor on a FGPA requires very little additional logic. Along with the normal input and output signals associated with an adder/subtractor, this modified version also accepts a sign change signal for each of the two input values. If the sign change signal is 1, then the associated value is what it should be. However, if the sign change signal is 0, then the corresponding value is inverted. Table 4.2 provides a summary of how the modified adder/subtractor operates. There are 4 different cases, as shown in Table 4.2. If the sign change signal of input values A and B are the same then we add the two values. On the other hand if the sign change signal of the two input values are different then we subtract B from A. The resulting sign change is always the same as the sign change of A. The sign change signal for each value is initially assigned so that it corresponds to the spreading sequence. The sign change signal is carried through the adder tree structure. Figure 4.8 shows an example of what operations would occur at each modified adder/subtractor when the spreading sequence for the system is 1100 1111 0100 0110.

Table 4.2: Sign signal operation summary

A's Sign Change	B's Sign Change	Operation	Resulting Sign Change
0	0	A+B	0
0	1	A-B	0
1	0	A-B	1
1	1	A+B	1

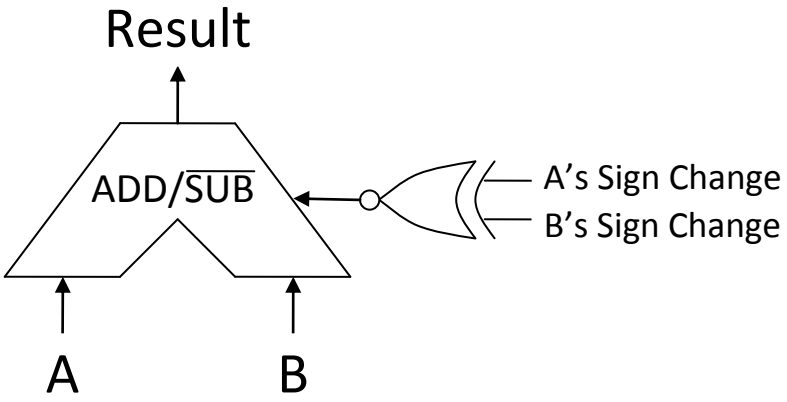

Figure 4.7: Modified Adder/Subtractor

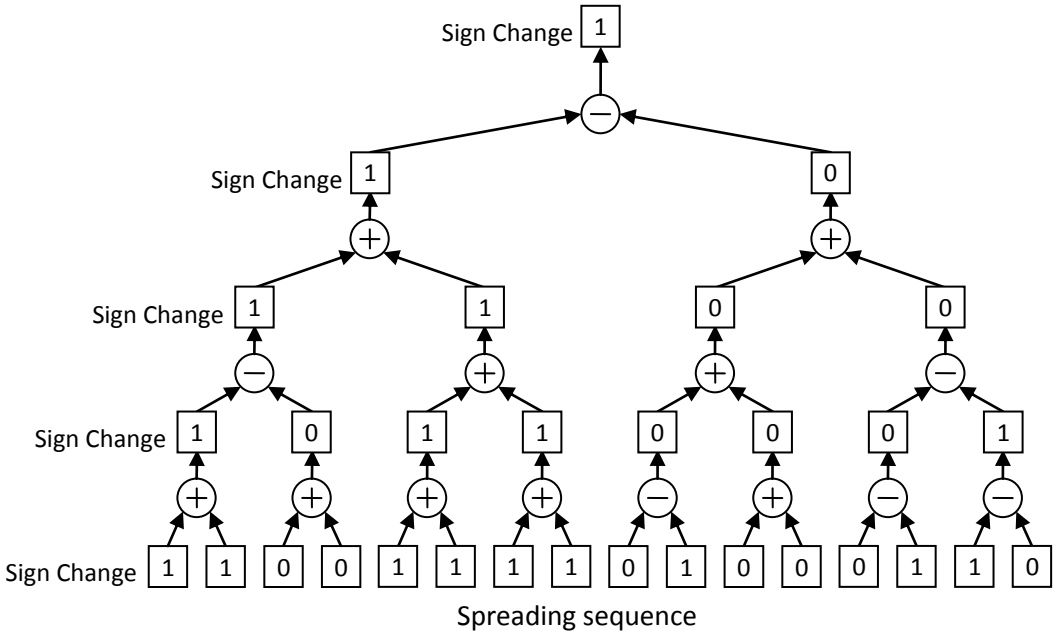

Figure 4.8: Chip correlator example with spreading sequence equal to 1100111101000110

The output of the chip correlator goes to a storage module which comprises 16-bit shift registers. We chose to use 16-bit shift registers because they are readily available on the Xilinx XC2V8000 Virtex II FPGA. Utilizing these 16-bit shift registers we are able to produce a fast and resource efficient storage module for the symbols. The shift registers are configured as shown in Figure 4.9. A set of 6 parallel 16-bit shift registers is required to store the 6-bit symbols coming from the chip correlator. There are 39 sets of 16-bit shift registers, which are all connected in a long chain to form the symbol storage module.

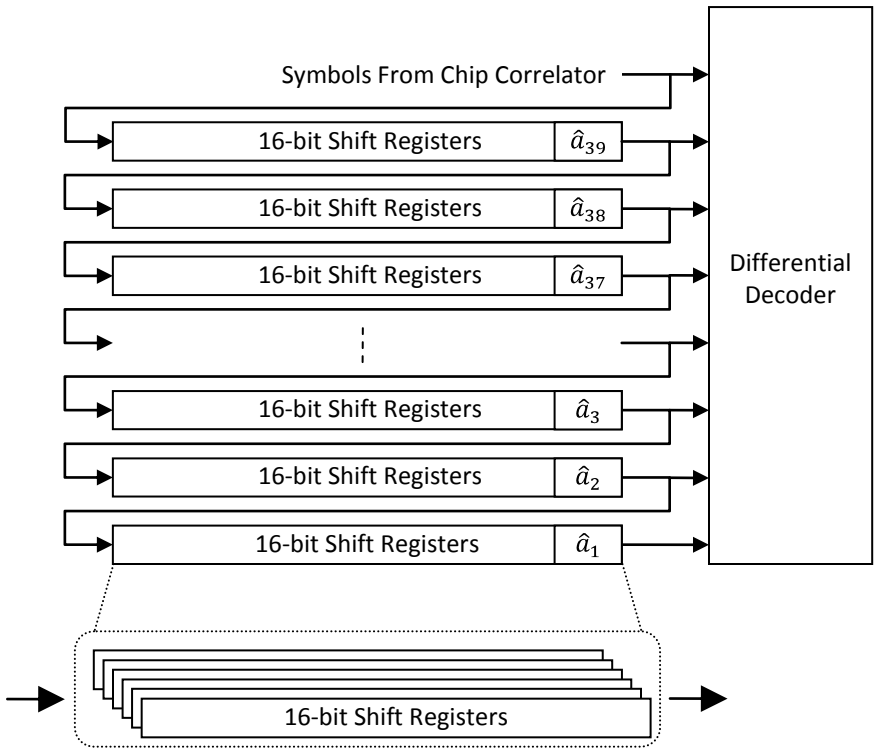

Figure 4.9: Symbol storage module

The chip correlator produces consecutive symbols every 16 clock cycles. Therefore in order to get sequential symbols that make up a preamble we take every other 16th value in the symbol storage module. To do this we take the output of each shift register set along with the output from the chip correlator and feed it into the differential decoder.

Both the chip correlator and symbol storage module operate independently on the in-phase and quadrature paths. The differential decoder combines both these paths by performing complex variable multiplication on every consecutive pair of symbols as, shown in Figure 4.10. Performing complex variable multiplication requires four multiplications and two additions. However, the algorithm is only concerned with the real component of the product, therefore the system only needs to perform two multiplications and one addition, as shown in Equation (3.8) and Figure 4.10. After performing differential decoding the system is left with 39 estimated symbols of the preamble.

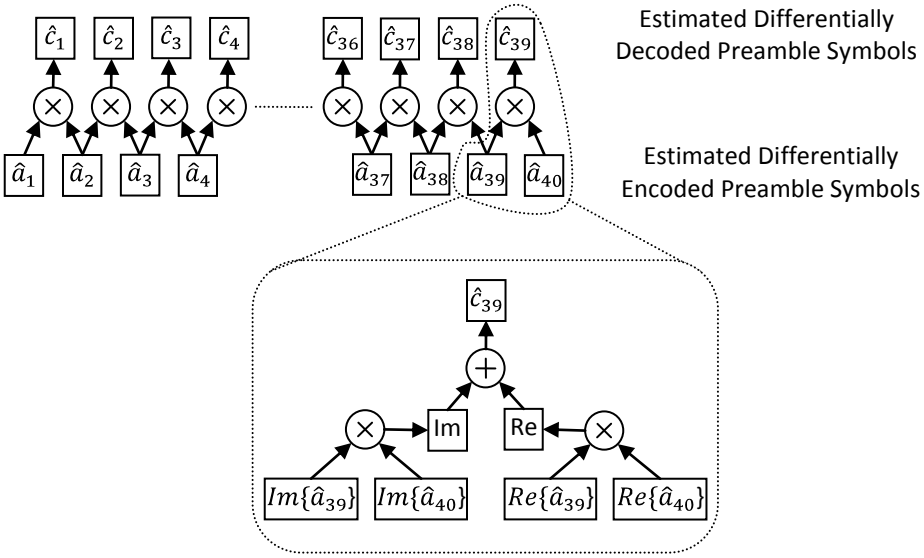

Figure 4.10: Block diagram of the differential decoder

The next step is to correlate the estimated preamble symbols $\hat{c}$ with the known preamble sequence c. The module to perform this correlation is very similar to that of the chip correlator. The difference is that we have 39 13-bit symbols to correlate. With 39 values to add together we cannot form a perfect binary tree such as the one found in the chip correlator module. In this case we have to settle for a slightly less efficient structure. The correlation tree structure is show in Figure 4.11. A perfect binary tree structure is built for the first 32 values. The remaining 7 values are summed with a partial tree structure. Delays are added to the structure so that all values for a particular preamble are added together. With the symbols lining up in each layer we are able to pipeline the symbol correlator. The resulting sum at the top of the tree structure is our estimated correlation peak η.

The final module on the preamble detector is a comparator. It compares the correlated value with a given threshold G. If η is greater than G, a preamble is declared detected and the signal is sent to the payload extractor to decode the subsequent payload data.

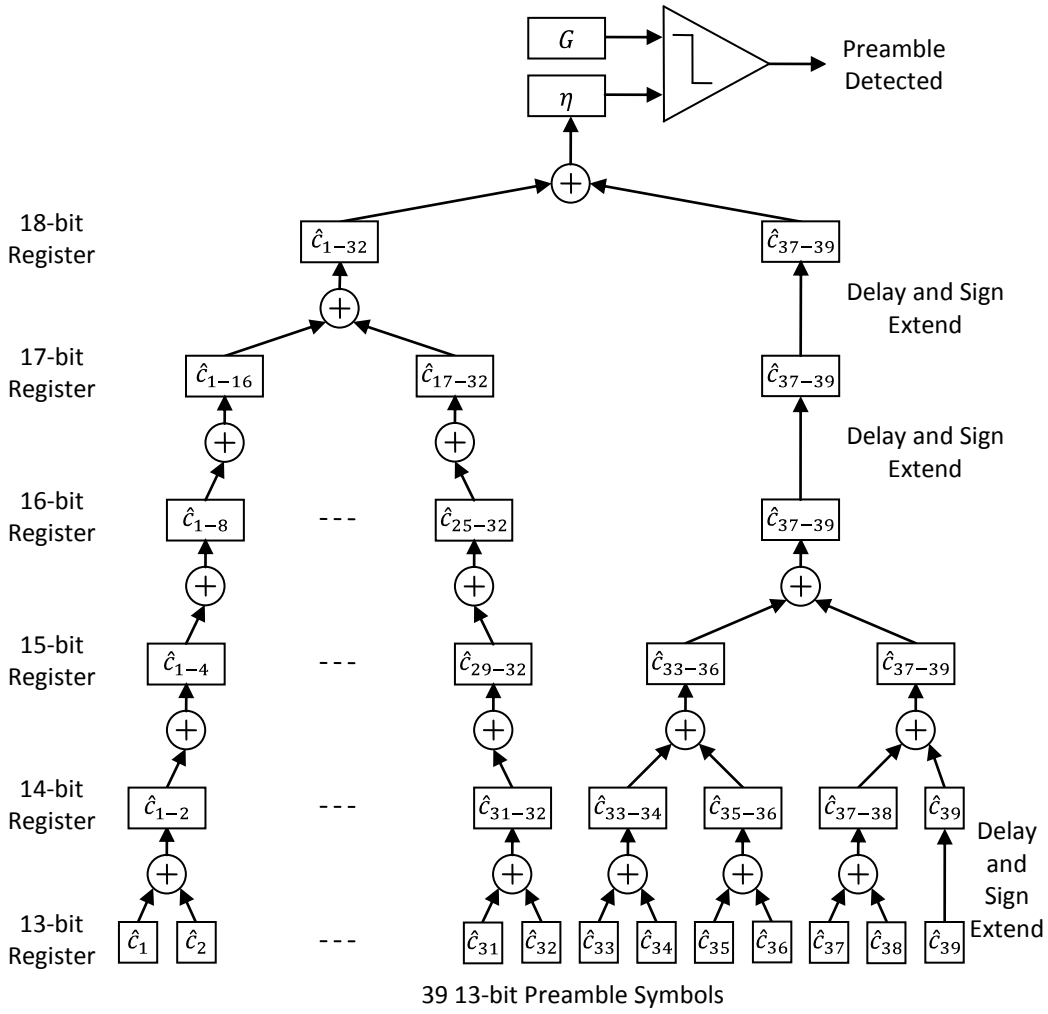

Figure 4.11: Block diagram of the preamble symbol correlator and comparator

4.3.2 **Payload Extractor**

When a preamble has been detected it sets a flag bit to indicate that the payload is ready for extraction. The same chip correlator module used in the preamble detector is used to correlate the symbols for the payload. Since the payload consists of 16-bits, with each bit repeated 4 times, we need to add the 4 symbols together. The symbols come serially from the differential decoder and are shifted into shift registers so that they can be summed in parallel. Summation is done with a small adder tree structure, as shown in Figure 4.12. The sign of the each sum represents one bit of the payload and is shifted into a 16-bit shift register for storage until the complete payload is extracted. To control the flow of data a counter is used. Every 16 chip clock periods,

a symbol from the chip correlator is driven into the shift registers. After 4 new symbols have been shifted into the registers, another clock signal triggers data out of the adders and into the payload shift registers. The value of the counter is monitored to determine the end of the payload.

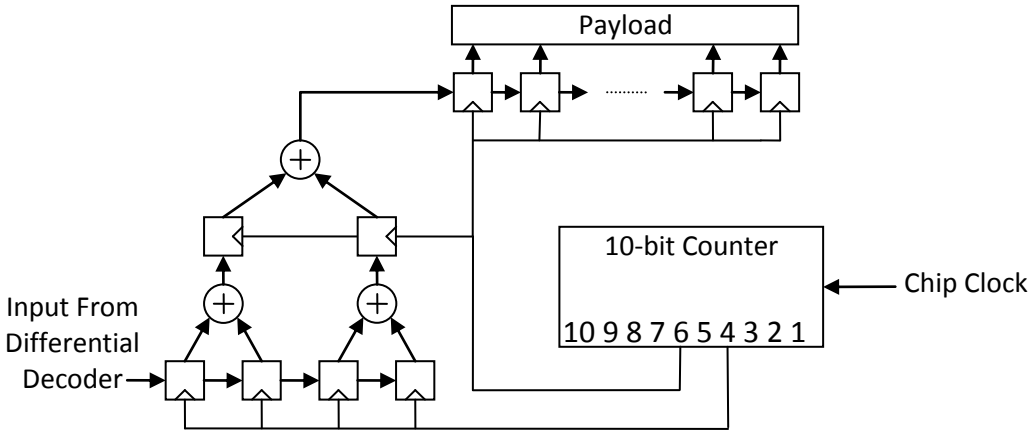

Figure 4.12: Block diagram of payload extractor

4.4 Noise generator

The noise generator creates Gaussian noise samples by adding 48 uniformly distributed random variables (UDRV) together. To generate the UDRV we use the Tausworthe algorithm [40].

The Tausworthe pseudorandom number generator (PNG) provides the system with a new 32-bit UDRV every clock cycle. The noise generator instantiates two PNGs with different seed values. Each 32-bit UDRV is split in half to provide the noise generator with four 16-bit random vectors every clock cycle. The four 16-bit UDRV, are summed using a two stage adder tree and the result is fed into an accumulator. Using a pipelined architecture, the PNG is able to create a new sample every 12 cycles by combining 48 16-bit UDRV. A block diagram of this noise generator is given in Figure 4.13. The Gaussian noise samples have a mean of 0 and a variance of 1. A more detailed mathematical description of how the noise generator works is given in the Section 4.4.1.

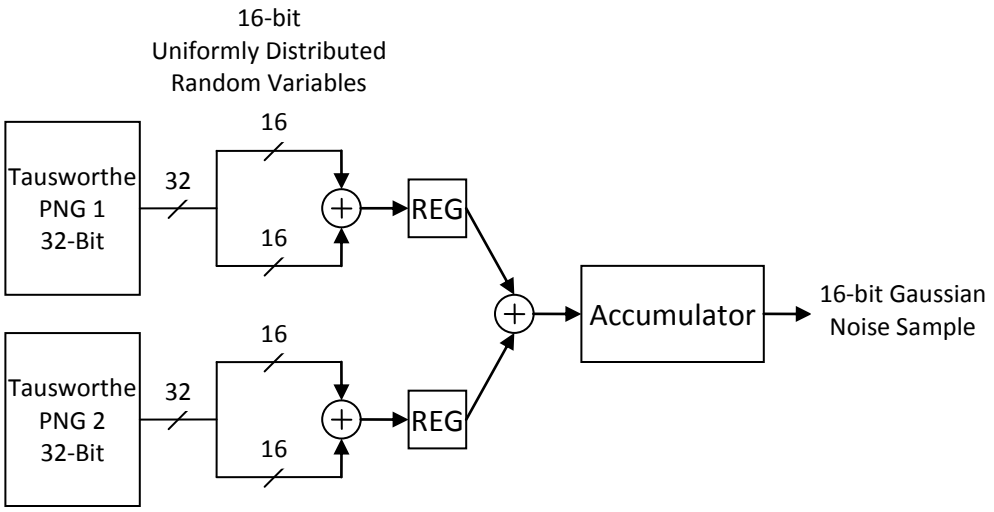

Figure 4.13: Block diagram of the noise generator for the simulator

4.4.1 **Tausworthe Pseudorandom Number Generator**

Given a 16-bit uniformly distributed random vector we can interpret it as a two's complement number representing a value between -0.5 and 0.5. By adding 12 of this 16-bit UDRV we can obtain a Gaussian random sample with a mean of 0 and variance of 1.

$$\sigma_u^2 = E\{u^2\} = \int_{-\frac{1}{2}}^{\frac{1}{2}} u^2 du = \frac{u^3}{3}\Big]_{-\frac{1}{2}}^{\frac{1}{2}} = \frac{1}{3}\left(\frac{1}{8}+\frac{1}{8}\right) = \frac{1}{12} \tag{4.1}$$

$$\sigma_s^2 = E\{SS^*\} = \sum_{i=1}^{12}\sum_{j=1}^{12} E\{u_i u_j\} = 12\sigma_u^2 = 1 \tag{4.2}$$

$$\mu_s = E\{S\} = \sum_{i=1}^{12} E\{u_i\} = 0 \tag{4.3}$$

The above model was simulated in Matlab and plots were generated to compare our model with a reference Gaussian curve [41]. In one test case 10^7 sample points were generated, where twelve 16-bit UDRVs were added together. The sample points had a mean of -1.4600×10^{-4} and variance of 1.0000. The plots of the

probability density function of the sample points are shown Figure 4.14 and Figure 4.15. From the plots we observe that this model is able to produce a decent Gaussian distribution going out to approximately 3 standard deviations. Beyond that the PDF of the samples generated from our model does not align to the reference Gaussian curve. To improve our model we needed to add more UDRVs together. But in doing so we would no longer have a unit variance Gaussian noise generator. To obtain a Gaussian noise generator with unit variance we would have to perform division on our samples. An implementation of a divider in hardware can be a complex task, unless the divisor is a power of 2, in which case the operation can be achieved with a simple logical shift. This simplistic way of performing division has led to our decision of using 48 UDRV to generate our Gaussian noise samples. Adding 48 UDRV results in a $\sigma_s^2 = 48\sigma_u^2 = 4$, if we divide each sample by 2 the σ_s^2 will be 1.

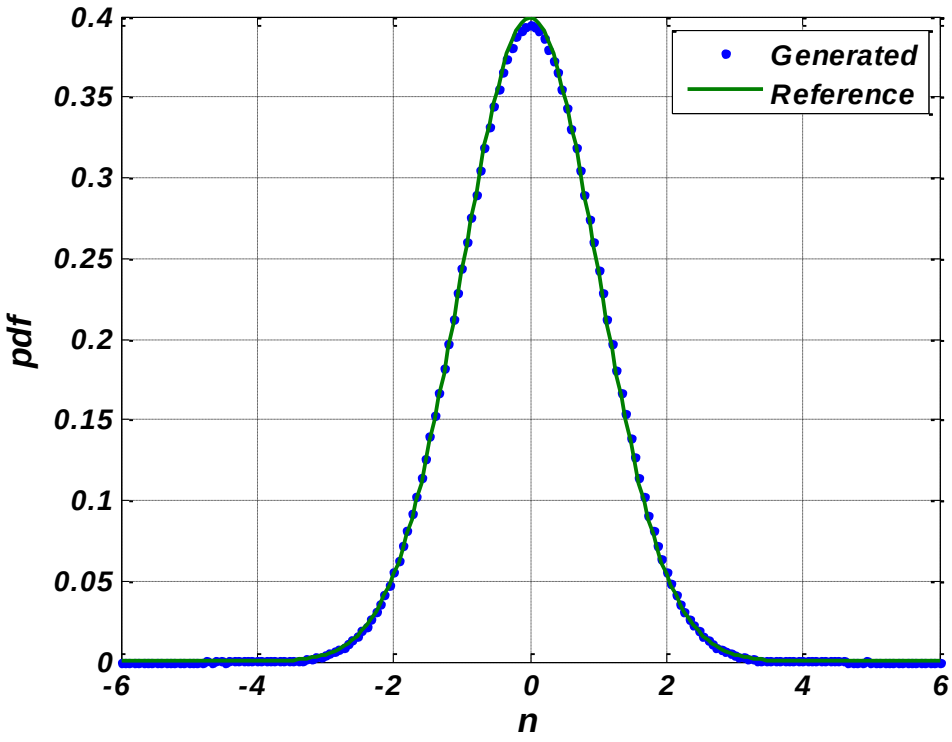

Figure 4.14: Plot of probability density function of a Gaussian noise generator that uses twelve 16-bit UDRV with linear y-axis

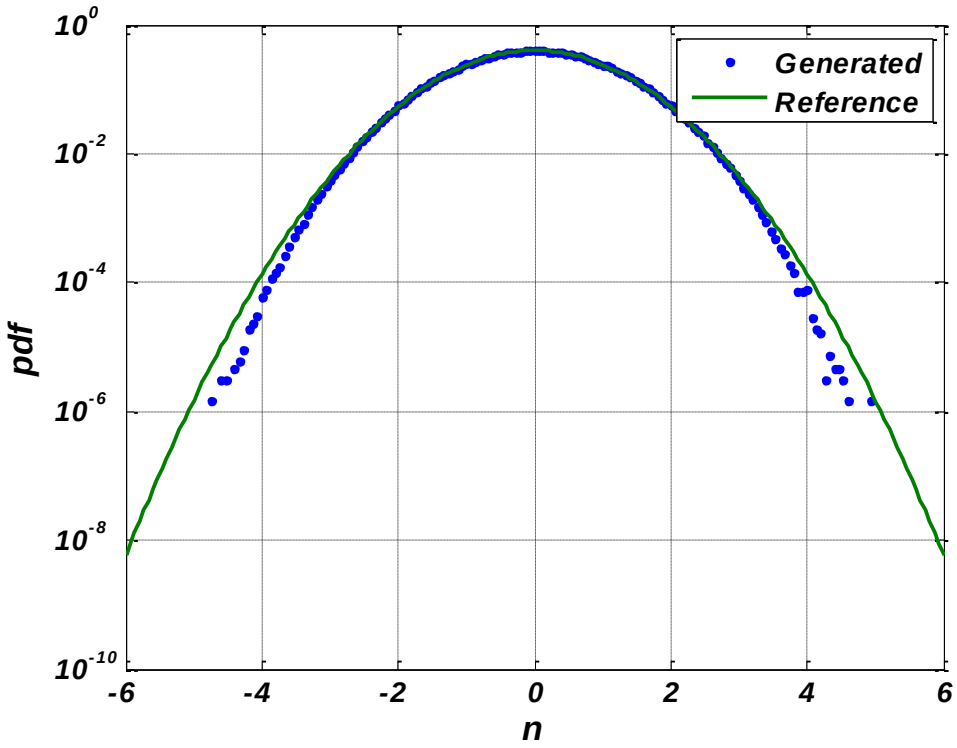

Figure 4.15: Plot of probability density function of a Gaussian noise generator that uses twelve 16-bit UDRVs with log y-axis

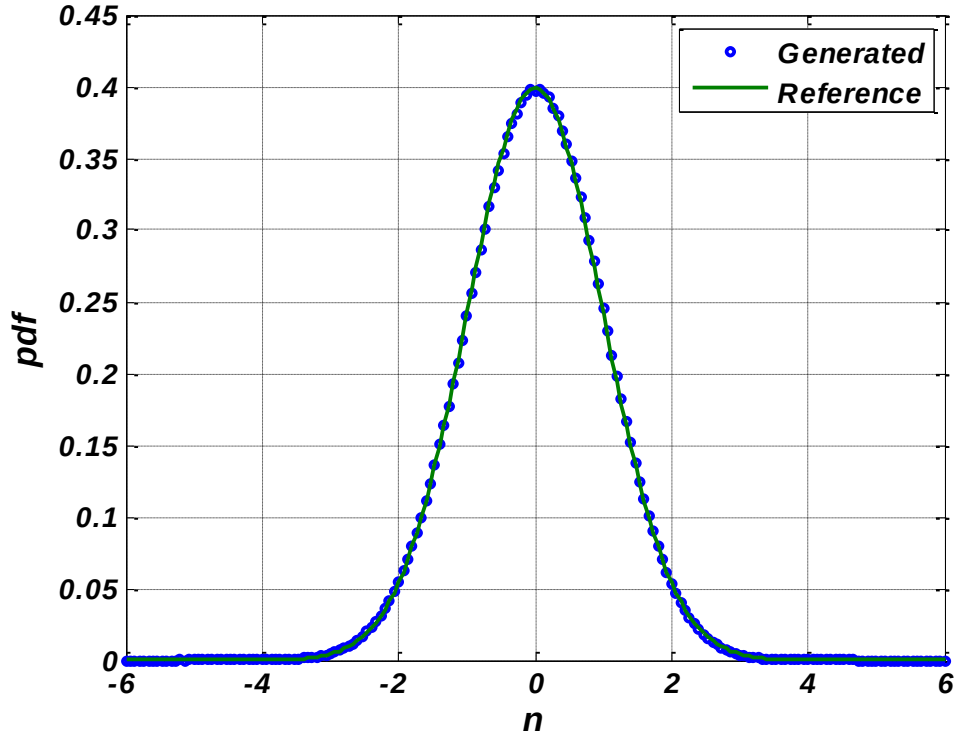

Figure 4.16: Plot of probability density function of a Gaussian noise generator that uses 48 16-bit UDRV with linear y-axis

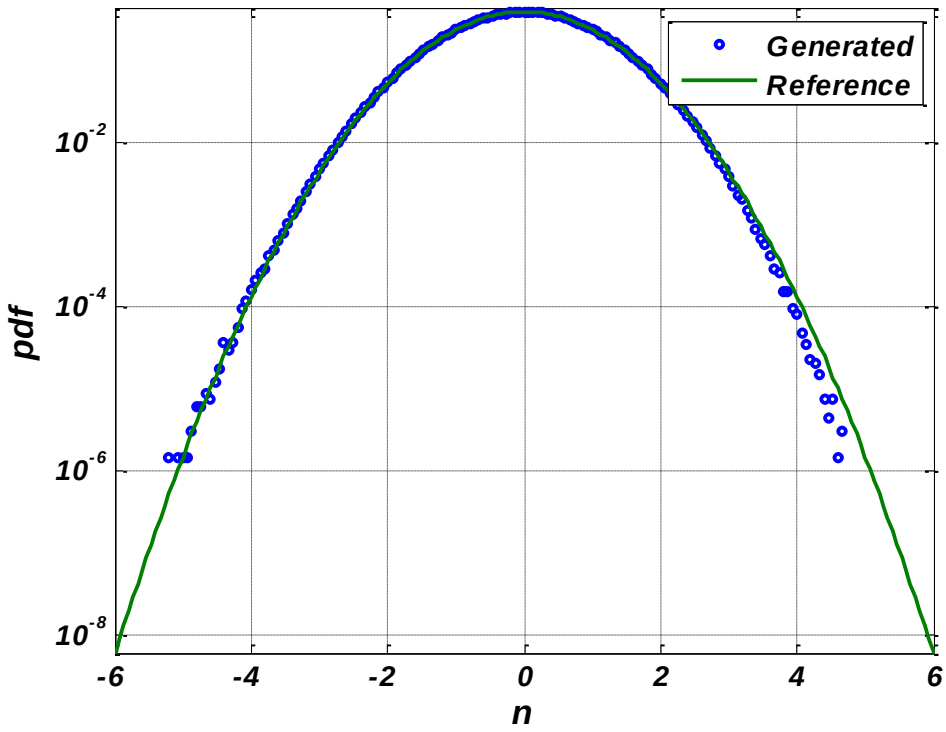

Figure 4.17: Plot of probability density function of a Gaussian noise generator that uses 48 16-bit UDRVs with log y-axis

A simulated test run was performed again with our model, but this time we used 48 UDRVs. In one test run with 10^7 sample points the mean was 4.9406×10^{-4} and variance of 0.9995. The PDF of the samples points generated with this model as shown in Figure 4.16 and Figure 4.17 are a closer match to the reference Gaussian curve. This Gaussian noise generator model is apparently a fairly reliable in producing samples points up to about 4 standard deviations.

4.5 Chapter Summary

A description of the hardware implementation was presented in this chapter. Each component of the transmitter, receiver and noise generator was explained in detail. Similar to the low-power preamble detection system in [42], our system operates on a 2-bit preamble correlator. The following chapter shows, that this 2-bit implementation provides performance curves similar to that of the analytical results in [33].

Chapter 5
Results

The chapter is divided into two main sections. Section 5.1 presents the results gathered from the C simulations and Section 5.2 presents results from the hardware implementation of the preamble detection emulator.

5.1 C Simulations

The C simulations allowed us to determine tradeoffs between different system configurations. The section is divided into 2 parts: Experiment Setup and Results/Observations.

5.1.1 Experiment Setup

Two types of simulations were performed: synchronous and asynchronous. The synchronous simulation assumes perfect sampling, where each chip is sampled at the maximum chip energy. In the asynchronous simulation, each chip is split into 8 samples and discrete rectangular pulse matched filtering is performed. The output of the matched filter is sampled and sent to the decoder.

The simulator prepares a sequence with two preambles and random data in between, as shown in Figure 5.1. The preamble sequence a and spreading sequence b are randomly determined. The symbols in both the preambles and data are spread by the same chip spreading sequence. The energy of each chip is set according to a given SNR value. Once the test sequence is prepared, a particular window of $W \times L_b$ consecutive chips is sent to the preamble detector. For every test sequence all possible windows alignments are tested. This test sequence structure was designed to emulate all the possible alignments for a preamble. The different cases for all possible offsets are described in Table 5.1.

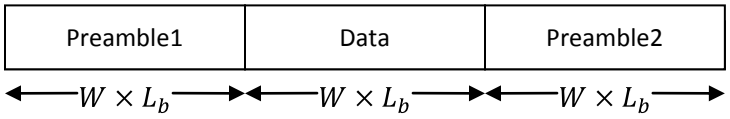

Figure 5.1: Simulation test sequence structure

Table 5.1: Simulation transmit signal window alignment

Offset (x)	Window Content
$x = 0$	Contains all of preamble1
$0 < x < W \times L_b$	Contains only the end of preamble1 and some random data
$x = W \times L_b$	Contains only random data
$W \times L_b < x < 2 \times W \times L_b$	Contains only the start of preamble2 and some random data
$x = 2 \times W \times L_b$	Contains all of preamble2

A monitor module counts four mutually exclusive events: true positive (TP), false positive (FP), true negative (TN) and false negative (FN). A TP event occurs when a preamble is sent and is subsequently successfully detected. Conversely, if the detector is unsuccessful, a FN event occurs. When the transmitter has not sent a preamble and the detector erroneously indicates a detection, this is a FP event. Lastly, the most common case for the system is a TN event. This occurs when the system is simply listening to the empty channel and correctly reports that a preamble has not been sent. The conditions that trigger these events are summarized Table 5.2. The statistics gathered from counting these events can be aggregated into more useful information. In particular, it is advantageous to determine the probability of missing a preamble (P_{miss}) and the probability of false alarms (P_{false}).

Table 5.2: Mutually exclusive events that can occur when detecting preambles

		Preamble Detected	
		YES	NO
Preamble Sent	YES	True Positive	False Negative
	NO	False Positive	True Negative

P_{miss} is how often the system does not detect a preamble that was sent. P_{false} is how often the system falsely detects a preamble that does not exist. The equations P_{miss} and P_{false} are given in Equation (5.1) and Equation (5.2).

$$\boldsymbol{P_{miss} = \frac{miss\ count}{total\ number\ of\ sent\ preambles} = \frac{FN}{TP + FN}} \quad (5.1)$$

$$\boldsymbol{P_{false} = \frac{false\ detection\ count}{total\ number\ of\ detection\ with\ no\ preamble} = \frac{FP}{TN + FP}} \quad (5.2)$$

5.1.1 **Preamble and Spreading Sequence Effects on Performances**

The choice of preamble and spreading sequence affects the performance of the preamble detection system. As long as the preamble is pseudo-random, it has no effect on the P_{miss} performance. The spreading and preamble sequence does, however, greatly influence the performance of P_{false}. A preamble which has an autocorrelation with many large peaks, produces poor P_{false} performance. This occurs because the packet detection system performs threshold based decisions. If a certain preamble has an autocorrelation with several high peaks, a system with a low threshold value will indicate several detections for the single preamble, thus increasing the number of false detections.

The ideal preamble would have an autocorrelation with only one large peak at the instance when the sequences are perfectly aligned and all other positions correlate to zero. This is impossible to achieve with any practical preamble length. Therefore, we have to settle for a preamble which correlates to a high value when the sequences are aligned and every other position correlates to low values. More specifically, we want to find a sequence with a significantly smaller second highest correlated value when compared with the top two peaks in its autocorrelation. Since our preamble has an inner and outer sequence, the highest correlations values only occur when the inner sequences are align. Therefore, we only need to search for a low correlated outer sequence.

Given a certain preamble length, it is difficult to determine an optimal sequence analytically. Researchers have exhaustively searched and found the best low correlated binary sequences of length 3 to 60 [43]. The sequence of length 39 is provided in Table 5.3 ; its second highest autocorrelation peak is 0.077. This sequence was found in the literature after the results were gathered and presented in this thesis; we used a sequence of length 39 with a second highest autocorrelation peak of 0.183.

Table 5.3: Sequences used in the simulation and hardware implementation

Spreading Sequence	1101100111000010
Preamble Sequence	001001001000000111110001010100111001110

5.1.2 **Results**

This section presents results from the system with different configurations and parameters. Since we have two values to measure the performance of the system, the simulation sets the threshold G at certain values and reports back the P_{miss} and P_{false}.

All results are plotted against different values of preamble SNR_p. Calculations for the symbol SNR_s, chip SNR_c and chip energy E_c are given in Equations (5.3), (5.4) and (5.5). Typical SNR and chip energy values are given in Table 5.4.

$$\boldsymbol{SNR_s = SNR_p - 10log(W)} \tag{5.3}$$

$$\boldsymbol{SNR_c = SNR_s - 10log(L_b)} \tag{5.4}$$

$$E_c = 10^{\frac{SNR_c}{10}} \quad (5.5)$$

Table 5.4: SNR and chip energy values

SNR_p	SNR_s	SNR_c	E_c
10	-6.02	-18.06	0.0156
11	-5.02	-17.06	0.0197
12	-4.02	-16.06	0.0248
13	-3.02	-15.06	0.0312
14	-2.02	-14.06	0.0392
15	-1.02	-13.06	0.0494
16	-0.02	-12.06	0.0622
17	0.98	-11.06	0.0783
18	1.98	-10.06	0.0986
19	2.98	-9.06	0.1241
20	3.98	-8.06	0.1563
21	4.98	-7.06	0.1967

The first set of results is shown in Figure 5.2. This figure shows the performance curves for a system configured with B=4 (4-bit inputs) and L=4 (limit of ±4). We observe that the system's performance curve is within 0.5dB to the performance curve for the ideal system, which was determined by the analytical studies in [33]. There are three sets of curves for the 3 different values of $\boldsymbol{G}$: 29, 50 and 100. For an ideal synchronous system that uses high precision floating-point numbers, threshold values of 29, 50 and 100 is suppose to produce a P_{false} of 10^{-3} , 10^{-6} and 10^{-15}, respectively [33]. However, the measured P_{false} for a finite precision system with B=4 and L=4 is significantly different, as shown in the experiment results plotted in Figure 5.3.

Figure 5.4 and Figure 5.5 show the simulation results for an asynchronous system. These simulations use the same threshold values as the synchronous system simulation. Three different systems are simulated to determine the best configuration to implement in hardware. We have determined in the first set of experiments, that a synchronous system with B=4 and L=4 is close enough to the ideal system. Simulating a system using more inputs bits could only yield slightly improved performance, and would not be justifiable for the extra hardware that would be required to implement

the extra bits. However, we are interested in how much performance decreases with lower input bit precision, thus simulations were carried out on systems configured with B=2. Two different limit values, L=2 and L=4, were simulated on the system configured with B=2.

Figure 5.6 illustrates the frequency offsets that the system is able to tolerate. The simulation sets the preamble SNR value to 16dB, while measurements of P_{miss} are taken for frequency offsets of 10 to 1000Hz. An analytical curve is also provided for reference [33]. This chart shows that finite bit precision systems offer similar tolerance to frequency offsets as an ideal system.

From the simulation results we have decided that a system configured using B=2 and L=2 produces results that are good enough to proceed with the hardware implementation. This configuration has a good balance between hardware complexity and performance. Although we can achieve better performance with B=4 and L=4 system, the extra hardware required to implement this system is not worth the 1dB gain.

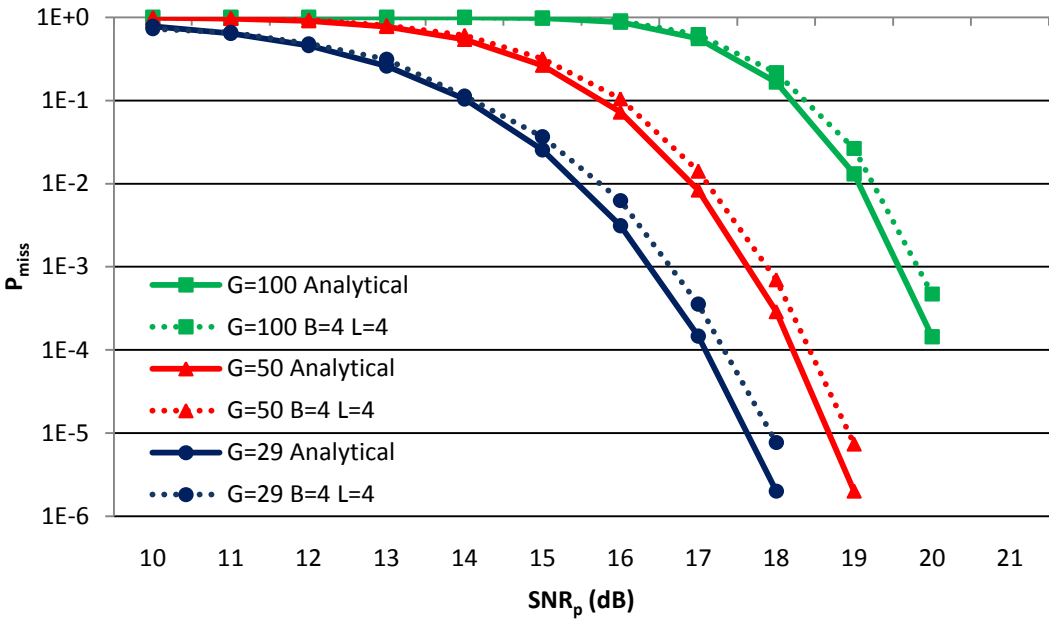

Figure 5.2: Synchronous System P_{miss} VS SNR_p – Comparison of P_{miss} performance of a system with B=4 and L=4 to the ideal analytical calculated P_{miss} for threshold values 29, 50 and 100.

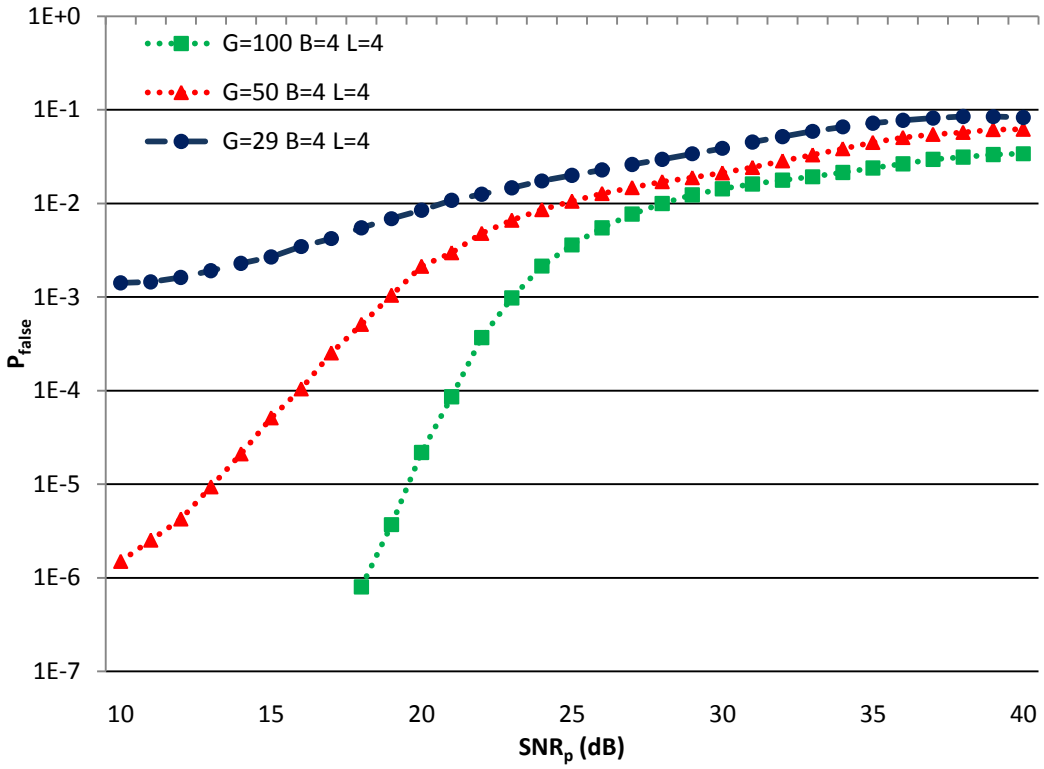

Figure 5.3: Synchronous System P_{false} VS SNR_p – Illustrates that the measured P_{false} for a finite precision system are significantly different from the analytical P_{false}

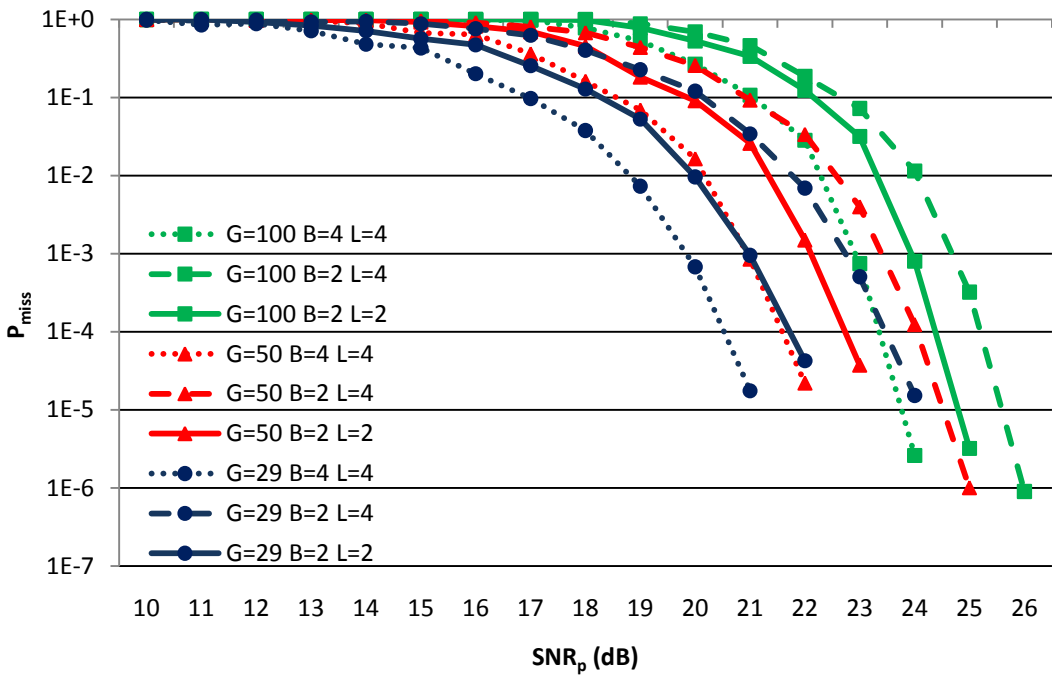

Figure 5.4: Asynchronous System P_{miss} VS SNR_p – Compares the P_{miss} performance of different system configurations for threshold values 29, 50 and 100.

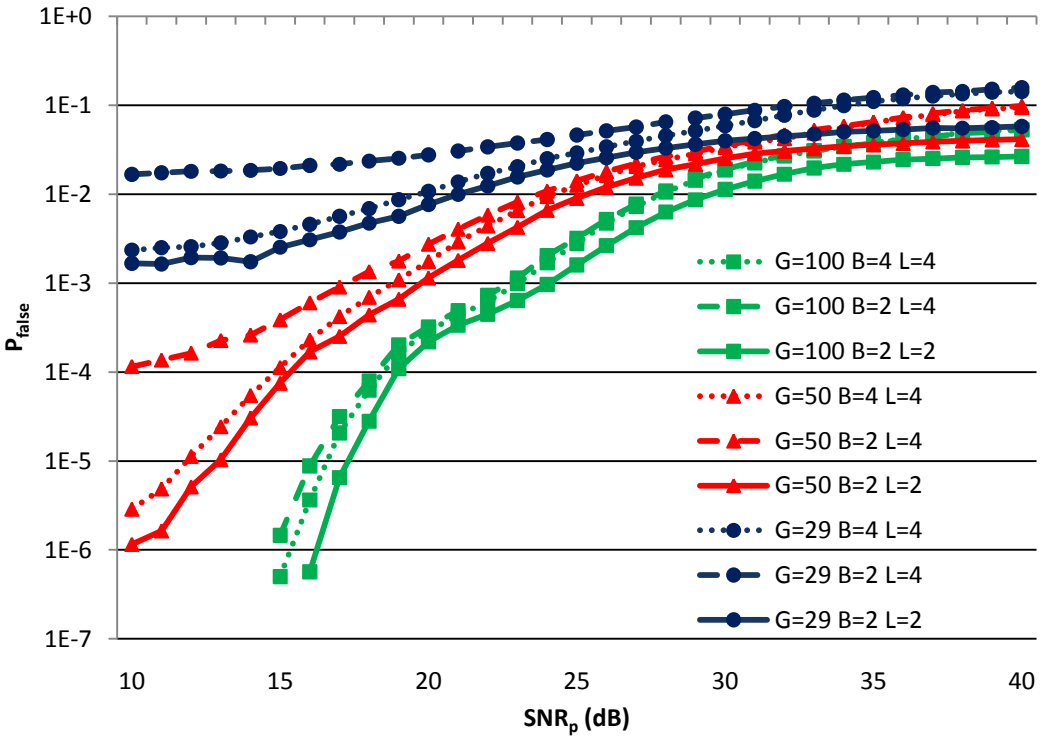

Figure 5.5: Asynchronous System P_{false} Vs. SNR_p– Compares the P_{false} performance of different system configurations for threshold values 29, 50 and 100.

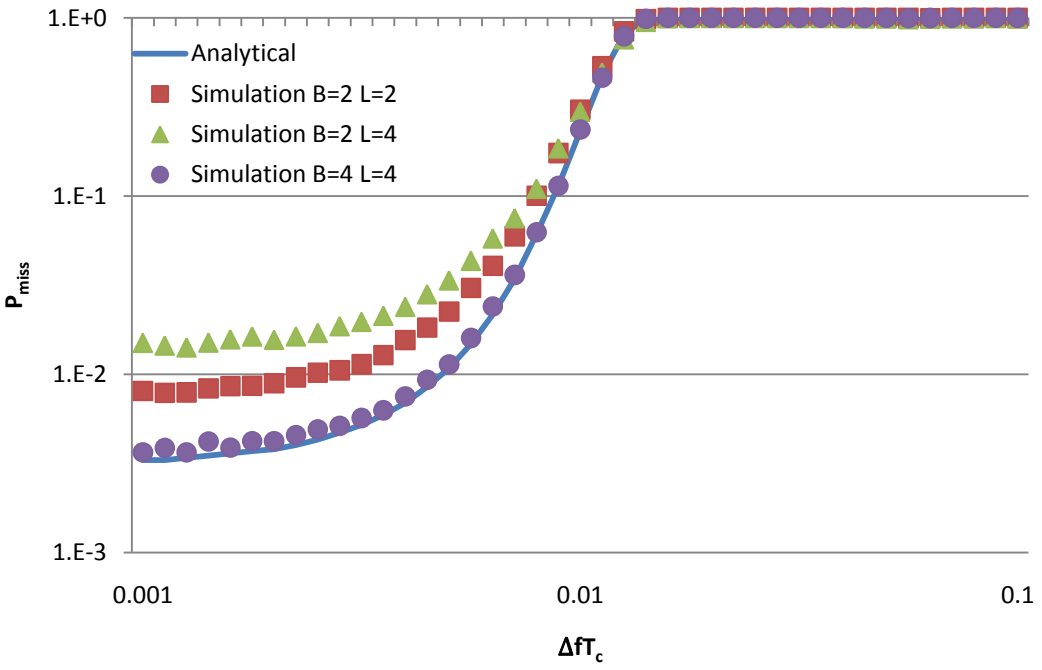

Figure 5.6: Synchronous System P_{miss} VS Frequency Offset – Illustrates the effects frequency offset on P_{miss} for different system configurations. SNR_p is set to 16dB

5.2 FPGA Implementation

Two different systems were implemented on the FPGA platforms. Section 5.2.1 describes an implementation of the preamble detection system in hardware. In this setup the whole system, including transmitter, noise generator and receiver, were implemented on one FPGA platform. A second implementation is described in Section 5.2.3. This implementation was developed as a proof of concept, to show that the preamble detection system could reliably detect packets asynchronously over two wireless platforms.

5.2.1 Hardware Emulation

An emulator was built to simulate the performance of the preamble detection algorithm in hardware. This hardware implementation consists of a transmitter, AWGN generator and receiver. Running at maximum clock speed the system produces approximately 75 MChips per second. At this rate the emulator simulates about 15000 packets per second. The emulator has two configurable parameters: SNR and threshold. To gather statistics the emulator uses 3 counters to keep track of missed packets, false detections and total packets generated. A software interface, running on a host computer, monitors these values and manipulates the parameters. Table 5.5 provides a summary of the FPGA utilization for the packet detection hardware emulator. Values are shown for a two different implementations, one that makes use of available onboard multipliers and another that only uses look-up tables (LUTs). Both sets of utilization values are given to provide an estimate of the additional resources that would be required if the system was implemented on a FPGA without on board multipliers.

Table 5.5: FPGA utilization for preamble detection emulator

	Preamble Detection System Using onboard multipliers	Preamble Detection System Using LUTs
Flip Flops	10685 (13%)	10763 (11%)
Look Up Tables	10745 (11%)	13943 (14%)
Multipliers	78 (46%)	0 (0%)

5.2.2 **Hardware Emulation Results**

In Figure 5.7 and Figure 5.8 the emulation results are plotted against the software simulation curves for a synchronous system with B=2 and L=2. Also included in the P_{miss} plots are analytical curves for reference. To maintain consistency, the same threshold values used in Section 5.1 are used here. Figure 5.7 and Figure 5.8 shows that software simulation and the hardware emulation produce similar P_{miss} and P_{false} performance curves. In Figure 5.8 all the curves follow a similar trend, which is as SNR increases so does the probability of false detections. Overall the results from the hardware emulations and software simulations agree.

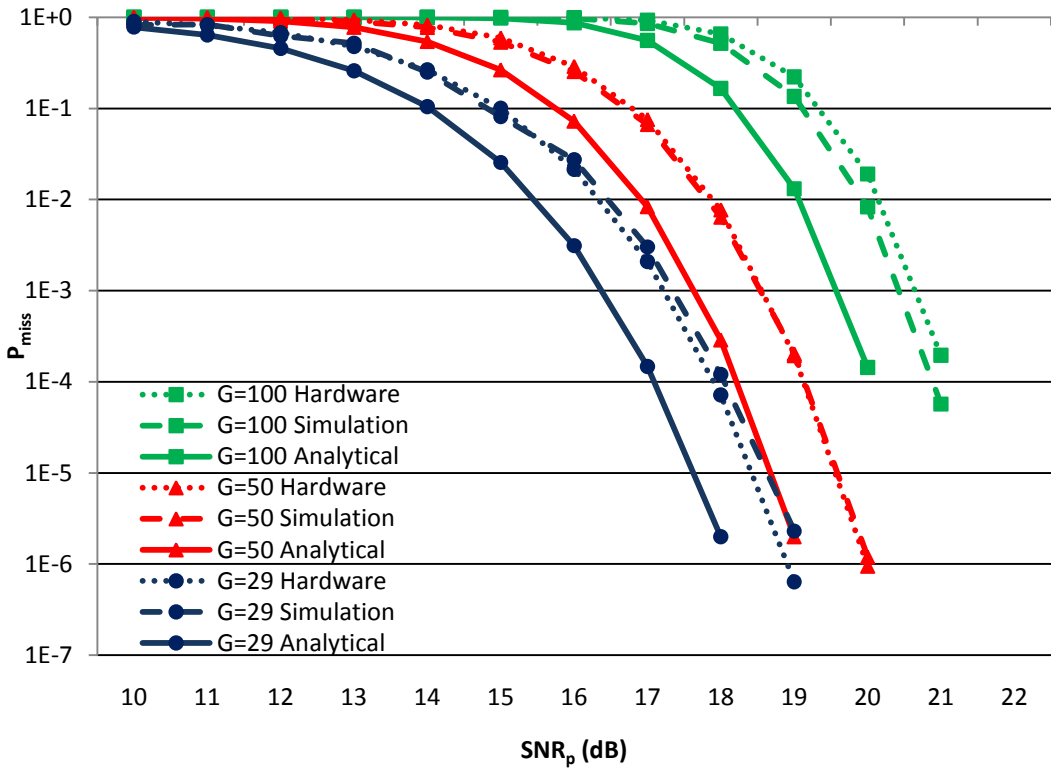

Figure 5.7: Synchronous System and Hardware comparison P_{miss} VS SNR_p – Compares the P_{miss} performance of the hardware with the simulation results.

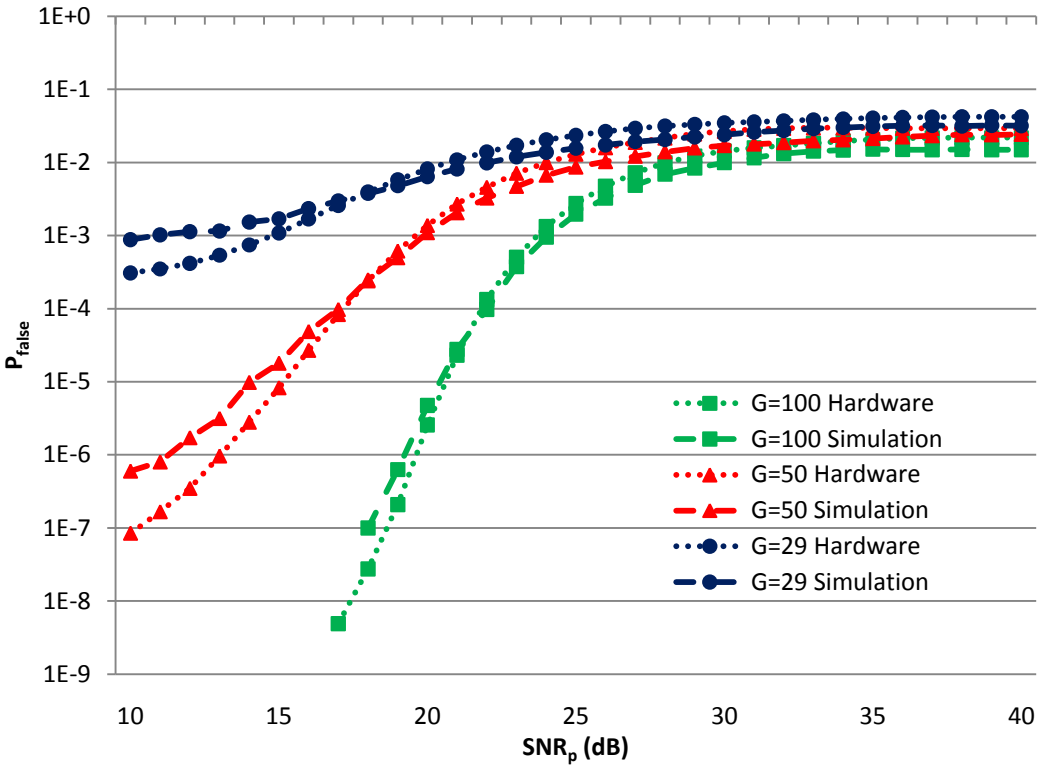

Figure 5.8: Synchronous System and Hardware comparison P_{false} VS SNR_p – Compares the P_{false} performance of the hardware with the simulation results

5.2.3 **Prototype**

The system was implemented on two Lyrtech SignalMaster-C67X FPGA platforms, one for the transmitter and the other for the receiver. Figure 5.9 shows the setup of the system. Packets are generated by the transmitter on the FPGA platform. The baseband signal of these packets is sent to the onboard DAC, where the digital sequence is converted to voltage levels and is sent to the radio link. The RF link is created using MAX2837 transceivers and is described in more detail in the next paragraph [44]. On the receiver platform, the voltage levels are converted back to a digital sequence with an ADC. The baseband signal is then passed to the packet receiver, where the preamble is detected and the payload is extracted. The two boards are connected to a host computer through a DSP to monitor and configure the system. A picture of the Pictures of the SM-C67X FPGA platform is shown in Figure 5.10. Furthermore the estimated utilization values are provided in Table 5.6.

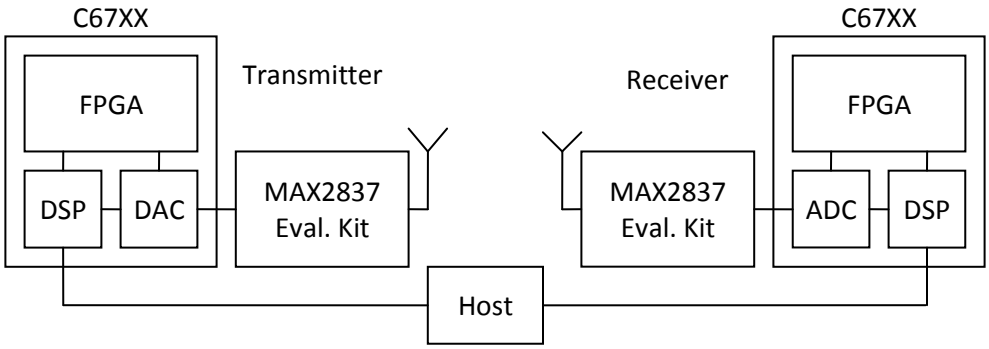

Figure 5.9: Packet detection FPGA board setup

Table 5.6: FPGA utilization for the prototype of the preamble detection system

	Preamble Receiver Using MULTs	Preamble Receiver Using LUTs	Preamble Transmitter
Flip Flops	5672 (6%)	5828 (6%)	73 (< 1%)
Look Up Tables	3730 (4%)	10126 (10%)	140 (< 1%)
Multipliers	156 (92%)	0 (0%)	0 (0%)

The RF front end consists of 2 MAX2837 RF transceivers. One transceiver is configured as a transmitter while the other is configured as a receiver. The baseband from the transmitter FPGA platform is sent through a SMA cable to the RF transmitter. The RF transmitter converts the baseband signal and sends it wirelessly over the 2.4GHz ISM band to the RF receiver. At the RF receiver module the signal is converted back to baseband and sent to the receiver FPGA platform. The process of transmitting the baseband sign from one FPGA platform to the other introduces an unknown amount of noise into the system. This makes it difficult to produce any meaningful performance curves. However this implementation does show that the preamble detection algorithm works. The Max 2837 evaluation kit setup is shown in Figure 5.11.

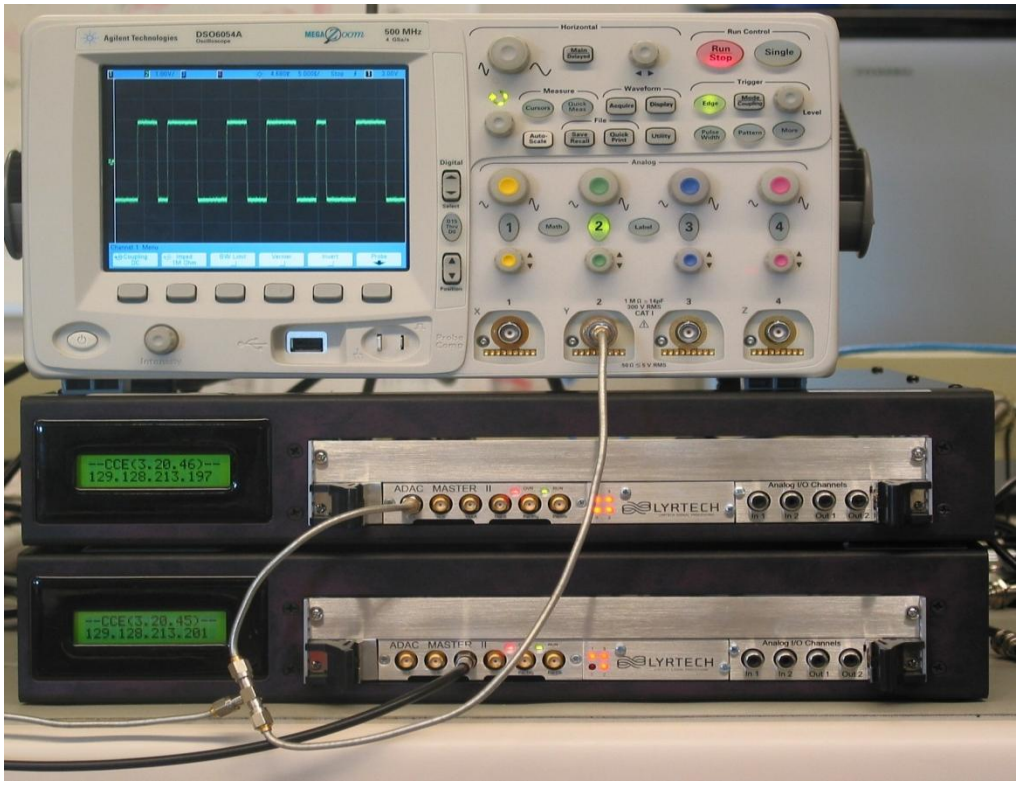

Figure 5.10: The transmitter (top) and receiver (bottom) were implemented on separate Lyrtech SM-C67X FPGA platforms. A black SMA cable carries the baseband signal from the transmitter to the RF link. From the RF link a gray SMA cable carries the baseband signal to the receiver and the oscilloscope. A screen shot of a received baseband signal is shown on the oscilloscope.

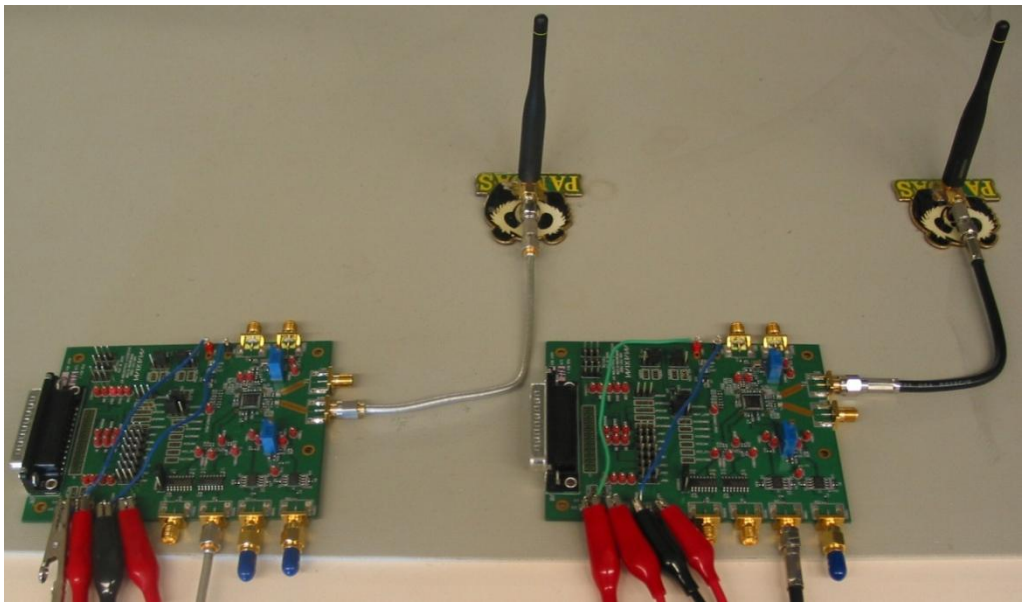

Figure 5.11: Picture of the radio link comprised of 2 MAX2837 RF transceivers evaluation kits. The baseband signal from the transmitter FPGA is sent to the RF transmitter board (right) through the black SMA cable. The receiver board (left) sends the baseband signal to the receiver FPGA through the gray SMA cable.

5.2.4 **Chapter Summary**

In this chapter we presented results from both the C simulation and the hardware implementation. It was shown that both systems had similar performance curves to that of the analytical results found in [33]. The P_{false} has been shown to steadily increase with SNR. This undesirable phenomenon can be resolved using an adaptive threshold base detection system, as presented in [45] [46] [47] and [48]. Also presented is this chapter was a prototype of the preamble detection system. This prototype demonstrated the functionality of the detection algorithm over a wireless RF link. The next chapter summarizes this thesis and presents some possible future directions.

Chapter 6
Conclusions and Future Work

In this thesis we present an implementation of a low-power, asynchronous, random-access packet detector. The system was first developed in a series of bit-true C simulations. From these simulations, it was determined that a system configured with 2-bit inputs and a limit of ±2, had sufficient P_{miss} and P_{false} performance to continue on to a hardware implementation.

The hardware implementations are on FPGA platforms. Two different systems were developed. The first system is a hardware emulation of the preamble detection algorithm. In this implementation the whole packet detection system, consisting of the transmitter, noise generator and receiver, is implemented on a single FPGA. Measurements taken from this system closely track the fixed-point simulations. The second system was designed as a proof of concept. The packet detection system was implemented on two FPGA platforms. One platform consisted of the transmitter while the other platform consisted of the receiver. Packets are sent from the transmitter to the receiver through a wireless radio link. This system demonstrates the functionality of the packet detection algorithm.

This thesis has described the design of an operational FPGA implementation of the preamble detection algorithm present in [33]. With some minor adjustments, the register transfer level code, for the core components of this system are ready to be added to an ASIC design.

6.1 Future Work

The work of thesis is only a small part of a much larger project. Since this work concentrates mainly on the detection of packets, not much effort was put in to the design of the payload section. Additional research will be required to determine the most appropriate coding scheme for the payload. Furthermore this physical layer

design will eventually require some higher level control. There is still much more work to be done before the preamble detection system can be truly appreciated. The following sections describe specific future work for the simulations and FPGA implementation.

6.1.1 Simulation

The simulation described in Chapter 3 is the preliminary study of how the preamble detection system performs in the real world. There are more studies that can be conducted to further our understanding of the system. This section describes some of the future work that can be done for the simulator.

The channel model used in this simulation deals with the effects of AWGN and some frequency distortion. A better model can be implemented to study other effects of a real channel, such as fading. Multi-path fading is a physical phenomenon that occurs as a result of multiple copies of the transmitted signal traversing different paths in the environment before arriving at the receiver. Fading is a random process and can change with time, geographical position and carrier frequency. The effects of fading can sometimes be helpful, but more often than not the results are destructive [4].

Another issue that can be further studied is the effect of interference. Interference occurs as different transmitters send packets concurrently and the signals overlap at the receiver. A more specific case in interference to study is the near-far problem. Picture a scenario where two devices are transmitting a signal with equal power. One of the transmitters is very far away from the receiver, while the other is closer. Due to the inverse square law the signal strength of the closer transmitter is considerably stronger, in some cases the signal might be orders of magnitude greater. Since the signal from one transmitter contributes noise to another, the closer transmitter is essentially jamming the channel for further away transmitters. The near-far effect needs to be simulated to understand its significance on our system performance [49].

6.1.2 FPGA Implementation

The FPGA implementation in Chapter 4 was designed as a proof of concept and contains many modules that can be redesigned to become more compact and power efficient.

The symbol storage module currently implemented on the system uses shift registers to hold the data. At every positive clock edge all the data shifts over one slot, which causes a lot of switching activity in the registers. High switching activity consumes more power and thus is an undesirable feature. One way to reduce the switching activity on the registers is to set up a structure similar to random-access memory. In such a structure we would be able to update a single individual symbol periodically. With only one symbol being updated each cycle the switching activity on the registers would be significantly reduced. Implementing such a structure would require a memory access control module which would increase the complexity of the storage module.

To ensure that no data loss occurs inside the preamble detector, additional bits are added to the intermediate results at each stage by the arithmetic processing units. The end result is a correlated value represented by 19 bits, which can represent values between -262 145 to 262 144. This range is much larger than the typical correlated values that have been observed in simulation. Further research is required to determine at which stages bits can be truncated without significant performance lost.

Another module that can be redesigned is the preamble generator. The current design uses forty 16-bit shift registers to store the preamble sequence. The preamble sequence is hardcoded and initialized into the shift registers when the FPGA is programmed. Without reprogramming the FPGA we cannot change the preamble sequence or structure. This module in the future may require designing to allow for the runtime configuration of the preamble sequence.

Bibliography

[1] K. Romer and F. Mattern, "The Design Space of Wireless Sensor Networks," *IEEE Wireless Communications*, vol. 11, no. 6, pp. 54-61, December 2004.

[2] S. Nagaraj, S. Khan, C. Schlegel, and M. Burnashev, "Differential Preamble Detection in Packet-Based Wireless Networks," *IEEE Transactions on Wireless Communications*, vol. 8, no. 2, pp. 599-607, February 2009.

[3] E. Son, B. Crowley, C. Schlegel, and V. Gaudet, "Architecture and FPGA Implementation of a Packet Detector for RF Motes," in *IEEE Military Communications Conference*, 2009.

[4] J. G. Proakis, *Digital Communications*, Fourth Edition ed.: McGraw Hill, 2001.

[5] J. G. Proakis and M. Salehi, *Fundamentals of Communication Systems*, 1st ed.: Pearson Prentice Hall, 2005.

[6] M. Yuce, W. Liu, J. Damiano, B. Bharath, P. Franzon, and N. Dogan, "SOI CMOS Implementation of a Multirate PSK Demodulator for Space Communications," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 54, no. 2, pp. 420-433, February 2007.

[7] IEEE std. 802.15.4-2006, "Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs)," 2006.

[8] J. Hill and D. Culler, "MICA: A Wireless Platform For Deeply Embedded Networks," *IEEE MICRO*, vol. 22, no. 6, pp. 12-24, Nov/Dec 2002.

[9] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy, "The Platforms Enabling Wireless Sensor Networks," *Communications of the ACM*, vol. 47, no. 6, pp. 41-46, June 2004.

[10] TinyOS. (2009, August) TinyOS Community Forum. [Online]. http://webs.cs.berkeley.edu/tos/

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 93-104, November 2000.

[12] RFM. (April, 2008) TR1000 916.50 MHz Hybrid Transceiver Datasheet. http://www.wirelessis.com.

[13] Chipcon. (2009, Feb) CC1000 Datasheet Single Chip Very Low Power RF Transceiver. www.ti.com/lprf.

[14] Crossbow. (2009, August) MICA2 Datasheet. http://www.xbow.com.

[15] J. Hill, *System Architecture for Wireless Sensor Networks,PhD Thesis Dissertation*.: University of California, Berkeley, 2003.

[16] C. Evan-Pughe, "Bzzzz zzz [ZigBee wireless standard]," *IEE Review*, vol. 49, pp. 28-31, March 2003.

[17] W. Kluge, F. Poegel, H. Roller, M. Lange, T. Ferchland, L. Dathe, and D. Eggert, "A Fully Integrated 2.4-GHz IEEE 802.15.4-Compliant Transceiver for ZigBee™ Applications," *IEEE Journal of Soild-State Circuits*, vol. 41, no. 12, pp. 2767-2775, December 2006.

[18] T. Nguyen, N. Oh, V. Le, and S. Lee, "A Low-Power CMOS Direct Conversion Receiver With 3-dB NF and 30-kHz Flicker-Noise Corner for 915-MHz Band IEEE 802.15.4 ZigBee Standard," *IEEE Transactions on Mircowave Theory And Techniques*, vol. 54, no. 2, pp. 735-741, February 2006.

[19] B. Guthrie, J. Hughes, T. Sayers, and A. Spencer, "A CMOS Gyrator Low-IF Filter for a Dual-Mode Bluetooth/ZigBee Transceiver," *IEEE Journal of Solid-State Circuits*, vol. 40, no. 9, pp. 1872-1879, September 2005.

[20] IEEE Computer Society, *IEEE Std 802.15.4™-2003: Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*., 2003.

[21] Crossbow. (2009, August) IRIS Datasheet. http://www.xbow.com.

[22] Crossbow. (2009, August) Imote2 Datasheet. http://www.xbow.com.

[23] Crossbow. (2009, August) MICAZ Datasheet. http://www.xbow.com.

[24] Crossbow. (2009, August) TELOSB Datasheet. http://www.xbow.com.

[25] Chipcon. (2008) CC2420 Datasheet: 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. http://focus.ti.com.

[26] Texas Instruments. (2009) CC2480 Datasheet: Z-Accel 2.4 GHz ZigBee® Processor. http://focus.ti.com.

[27] Chipcon. (2008) CC2431 Datasheet: System-on-Chip for 2.4 GHz ZigBee®/ IEEE 802.15.4 with Location Engine. http://focus.ti.com/.

[28] Texas Instruments. (2009, April) CC2530 Datasheet: A True System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee Applications. http://focus.ti.com.

[29] Freescale Semiconductor. (2009, August) MC13211/212/213 Datasheet: ZigBee™-Compliant Platform -2.4 GHz Low Power Transceiver for the IEEE® 802.15.4 Standard plus Microcontroller. http://www.freescale.com.

[30] ember. (2006, July) EM260 Datasheet: ZigBee/802.15.4 Network Processor. http://www.ember.com.

[31] ember. (2004) EM2420 Datasheet: 2.4 GHz IEEE 802.15.4 / ZigBee RF Transceiver. http://www.ember.com.

[32] K. J. Choi and S. Jong-In, "A Miniaturized Mote for Wireless Sensor Networks," in *International Conference of Advanced Communication Technology*, 2008, pp. 514-516.

[33] S. Khan, *Joint Packet Detection and Frame Synchronization for Asynchronous Wireless Networks, MSc Thesis Dissertation*.: University of Alberta, 2007.

[34] P. M. Shankar, *Introduction to Wireless Systems*., 2002.

[35] G. Box and M. Muller, "A Note on the Generation of Random Normal Deviates," *The Annals of Mathematical Statistics*, vol. 29, no. 2, pp. 610-611, 1958.

[36] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator," *ACM Transactions*

on Modeling and Computer Simulation (TOMACS), vol. 8, no. 1, pp. 3-30, 1998.

[37] Lyrtech Signal Processing, "SM-C67X-CPCI Technical Reference Manual v2.0," Lyrtech Inc., 2002.

[38] Lyrtech Signal Processing, "SignalMaster Hardware Technical Reference Guide V2.0," Lyrtech Inc., 2001.

[39] Lyrtech Signal Processing, "SM-ADAC Master II," Lyrtech Inc., 2005.

[40] P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Mathematics of Computation*, vol. 65, no. 213, pp. 203-213, January 1996.

[41] A. L. Garcia, *Probability and Random Processes for Electrical Engineering*, Second Edition ed., 1994.

[42] K. Gunnam, G. Choi, M. Yeary, and Y. Zhai, "A Low-Power Preamble Detection Methodology for Packet Based RF Modems on All-Digital Sensor Front-Ends," in *IEEE Instrumentation and Measurement Technology Conference*, 2007, pp. 1-4.

[43] H. Schotten and H. Luke, "On the search for low correlated binary sequence," *International Journal of Electronics and Communications*, vol. 59, no. 2, pp. 67-78, May 2005.

[44] MAXIM. (2007, August) MAX2837 Evaluation Kit Datasheet. http://datasheets.maxim-ic.com.

[45] A. Swaminathan and D. Noneaker, "The Effect of Automatic Gain Control on Serial, Matched-Filter Acquisition in Direct-Sequence acket Radio Communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 5, pp. 909-919, May 2001.

[46] S. G. Glisic, "Automatic Decision Threshold Level Control in Direct-Sequence Spread-Spectrum Systems," *IEEE Transactions on Communications*, vol. 39, no. 2, pp. 187-192, February 1991.

[47] S. G. Glisic, "Automatic Decision Threshold Level Control (ADTLC) in Direct-Sequence Spread-Spectrum Systems Based on Matched Filtering," *IEEE Transactions on Communications*, vol. 36, no. 4, pp. 519-527, April 1998.

[48] E. Brigant and A. Mammela, "Adaptive Threshold Control Scheme for Packet Acquisition," *IEEE Transactions on Communications*, vol. 46, no. 12, pp. 1580-1582, December 1998.

[49] H. Simon, *Communications Systems*, Fourth Edition ed., 2001.