A Non-Coherent Ultra-Wideband Receiver: Algorithms and Digital Implementation

by

Sinit Vitavasiri

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author ____

Department of Electrical Engineering and Computer Science
May 25, 2007

Certified by ____

Anantha P. Chandrakasan
Professor of Electrical Engineering
Thesis Supervisor

Accepted by ____

Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

A Non-Coherent Ultra-Wideband Receiver: Algorithms and Digital Implementation

by

Sinit Vitavasiri

Submitted to the Department of Electrical Engineering and Computer Science

May 25, 2007

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Ultra-wideband (UWB) communication is an emerging technique for wireless transmission in the 3.1-10.6 GHz unlicensed band with signal bandwidths of 500 MHz or greater. A non-coherent receiver based on energy collection reduces complexity, cost, and power consumption at the cost of channel spectral efficiency. The receiver collects the signal energy in two time windows and determines the transmitted bits based on which window has greater energy.

This thesis explains the implementation of low-complexity detection, synchronization, and decoding algorithms for a non-coherent ultra-wideband receiver. The receiver is modeled in MATLAB to measure performance. The UWB receiver performs effectively in noisy channels. At the signal-to-noise ratio (SNR) of 0 dB, the receiver achieves a detection miss rate of 2.1% and a false alarm rate of 1.2%. The synchronization error (within ±2 chip periods) rate is 0.5%. The bit error rate is 8.6%, but it drops sharply to 0.1% at an SNR of 5 dB. Moreover, the detection and the synchronization processes take 19.72 μs and 22.53 μs, respectively. The digital system is implemented in Verilog, which is mapped to hardware (FPGA). In the final system, a radio frequency and an analog front-end interface with the FPGA, resulting in a complete radio receiver.

Thesis Supervisor: Anantha P. Chandrakasan
Title: Professor of Electrical Engineering

Acknowledgements

I would like to thank Professor Anantha Chandrakasan, my academic advisor and thesis supervisor, for all help and support. Thanks to Manish Bhardwaj for the receiver algorithms, for comments on the design and implementation, and for technical support. Thanks to Denis Daly and Patrick Mercier for their comments and suggestions on the digital baseband architecture. Thanks to Nathan Ickes for setting up the FPGA lab kit. Thanks to Brian Ginsburg for help on the pattern generator and logic analyzer automation.

Contents

List of Figures

List of Tables

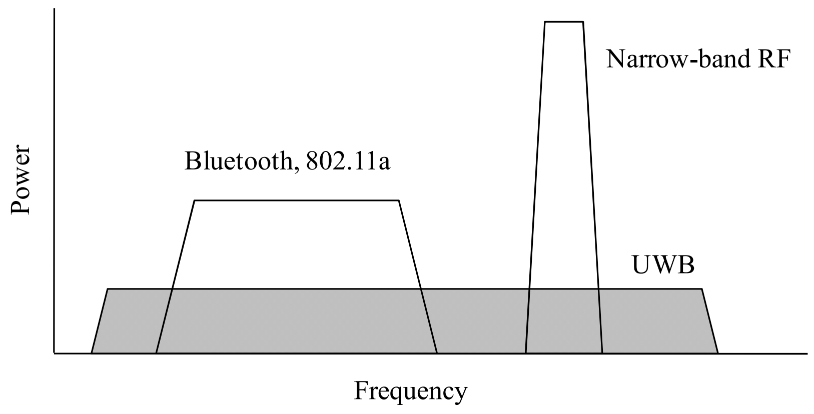
Chapter 1

Introduction and System Overview

1.1 Ultra-Wideband Technology Overview

Ultra-wideband (UWB) communication is an emerging technique for wireless transmission in the 3.1-10.6 GHz unlicensed band with bandwidths of 500 MHz or greater [1]. The emergence of commercial wireless devices based on ultra-wideband radio technology is widely anticipated. This novel technology has recently received much attention for major advances in wireless applications such as wireless communication, networking, radar, imaging, and positioning systems. Ultra-wideband technology brings the convenience and mobility of wireless communications to high-speed interconnects in devices throughout the digital home and office. Designed for short-range wireless personal area networks (WPANs), UWB is an emerging technology for freeing people from wires, enabling wireless connection of multiple devices for transmission of video, audio, and other high-bandwidth data [2].

UWB differs substantially from conventional narrowband radio frequency (RF) and spread spectrum technologies (SS), such as Bluetooth Technology and IEEE 802.11a/b/g, as shown in Figure 1-1. An ultra-wideband (UWB) device transmits sequences of information carrying pulses of very short duration, about 0.1 to 2 nanoseconds, thus spreading the signal energy from near DC to a few gigahertz. The corresponding receiver then translates the pulses into data by listening for a familiar pulse sequence sent by the transmitter. Specifically, UWB is defined as any radio technology having a spectrum that occupies a bandwidth greater than 20 percent of the central frequency, or a bandwidth of at least 500 MHz.



Note: Figure is not to scale

Figure 1-1: Comparison of UWB and other technologies.

Figure 1-2 compares UWB radio devices with conventional short-range wireless systems in terms of the achievable spatial capacity and the maximal transmission range. Although its transmission range is within 10 meters or about 30 feet, UWB radio devices have a very high spatial capacity for transmitting information [1]. Therefore, UWB, short-range radio technology, can complement other longer-range radio technologies such as Wi-Fi, WiMAX, and cellular wide-area communications. It can be used to relay data from a host device to other devices in the immediate area.

Modern UWB systems use other modulation techniques, such as Orthogonal Frequency Division Multiplexing (OFDM), to occupy these extremely wide bandwidths.

In addition, the use of multiple bands in combination with OFDM modulation can provide significant advantages to traditional UWB systems.

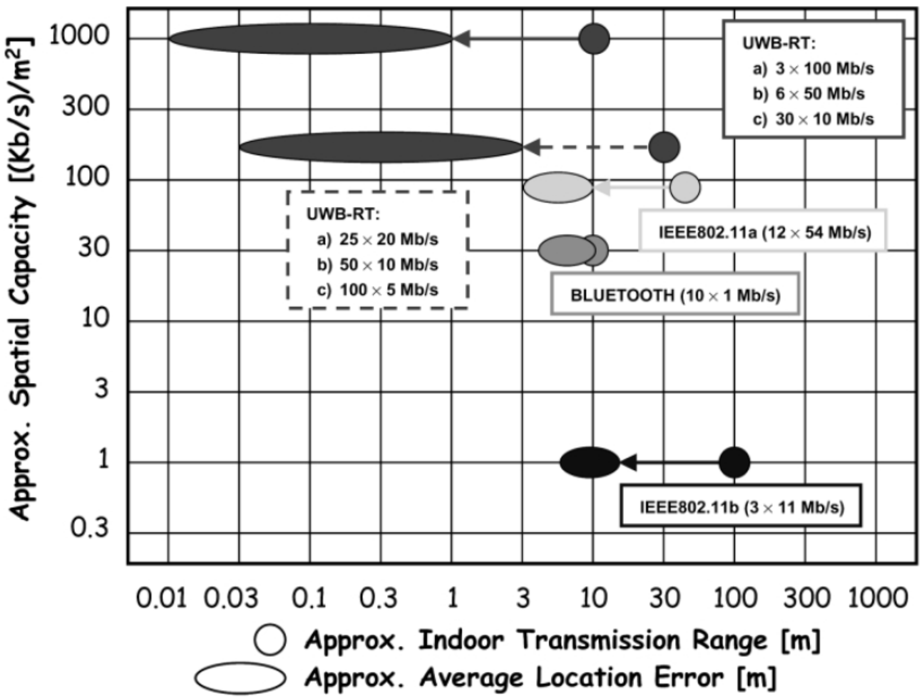


Figure 1-2: Comparison of UWB radio devices
and conventional short-range wireless systems [1].

UWB's combination of broader spectrum and lower power improves speed and reduces interference with other wireless spectra. In the United States, the Federal Communications Commission (FCC) has mandated that UWB radio transmissions can legally operate in the range from 3.1 GHz up to 10.6 GHz, at a limited transmit power of -41 dBm/MHz. Consequently, UWB provides dramatic channel capacity at short range that limits interference [2]. Therefore, the ultra-wideband radio technology is not only applicable to communications, imaging, and ranging, but it also alleviates the problem of scarce spectrum resources. The ultra-wideband radio technology potentially enables implementation of wireless platforms that support a variety of operating modes such as data transmission, precision positioning and tracking, and radar sensing. The technology can be used in wireless personal area networks (WPANs) and wireless local area networks (WLANs) with integrated position location and tracking capabilities. Table 1-1 summarizes features and benefits of the ultra-wideband technology in WPAN entertainment and personal computer environments.

Feature	Benefit
High-speed throughput	Fast, high-quality transfers
Low power consumption	Long battery life of portable devices
Silicon-based, standard-based radios	Low cost
Wired connectivity options	Convenience and flexibility

Table 1-1: Features and benefits of UWB

1.2 Problem Statement

Many of the approaches for implementing UWB receivers use a *coherent* receiver, which correlates the received signal with a well-designed template signal. It has been shown that a coherent receiver is optimal over AWGN (additive white Gaussian noise) and non-ISI (non-intersymbol interference) multipath channels. This type of receiver, however, has to cope with great design challenges. First, to correlate the received signal with the template signal, the receiver needs to achieve very precise pulse-level synchronization. Thus, despite some fast synchronization algorithms, the synchronization process continues to take long. Secondly, a precise template signal design is required to maximize the signal-to-noise ratio (SNR). This coherent design is difficult to achieve because of the distortions on the pulse shape over wireless channels. Finally, multipath energy combining requires a RAKE matched-filter receiver, which leads to high receiver complexity of the receiver design. A high-speed and precise clock may also be required.

A *non-coherent* receiver based on energy collection reduces complexity, cost, and power consumption at the cost of channel spectral efficiency [3-5]. The energy-collection based receiver utilizes binary pulse position modulation (BPPM). A receiver collects the signal energy in two time windows and determines the transmitted bits based on which window has greater energy. Table 1-2 summarizes key features of a coherent and a non-coherent receiver [6]. Because many wireless applications require energy efficiency, the non-coherent method is used in this project.

Feature	Coherent	Non-Coherent
Description	Correlates the received signal with a well-designed template signal	Based on energy collection
Advantage	Optimal over AWGN and multipath channels	Low complexity, low cost, low power consumption
Disadvantage	High complexity	SNR degradation

Table 1-2: Comparison of a coherent and a non-coherent receiver

In order for energy-collection decoding to work efficiently, a receiver has to know the beginning of a bit period. Therefore, pre-determined preamble signals need to be transmitted before actual data. An algorithm that synchronizes the system is also needed. Moreover, a non-coherent UWB receiver must be able to distinguish signals from noise (detection). UWB wireless system designs must balance tradeoffs among high bandwidth efficiency, low transmission peak power, low complexity, flexibility in supporting multiple rates, and reliable performance as expressed in terms of bit error rate (BER) [7].

This thesis proposes low-complexity detection, synchronization, and decoding algorithms for a non-coherent ultra-wideband receiver. The parameters of the algorithms are chosen to maximize the performance in AWGN and multipath channels. The receiver is modeled in MATLAB to measure performance. This thesis also aims to implement a digital system that receives a train of binary pulse position modulation signals and produces decoded bits. The digital baseband is implemented in Verilog, which is mapped to hardware (FPGA). In the final system, a radio frequency (RF) and an analog front-end will interface with the FPGA, resulting in a complete radio receiver.

1.3 Previous Works

A time modulated UWB receiver block diagram is presented in [8], where the implementation requirements of an integrated correlator are determined. However, [8] does not present the power consumption of the UWB-IR transceiver. Another UWB digital receiver, based on the frequency domain approach, is presented in [9]. This architecture requires a large number of low noise amplifiers (LNAs) and filter banks, which translates into increased power consumption. In [10] and [11], a digital UWB transmitter and a subbanded UWB receiver are implemented in 90 nm CMOS technology, respectively. Moreover, a complete UWB-IR transceiver architecture for tag-based wireless sensor networks in 0.35 μm BiCMOS process is presented in [12]. The theoretical framework for a non-coherent UWB receiver is developed in [13], [14], and [15].

1.4 Thesis Outline

This chapter describes the system model and the receiver structure. The synchronization, the detection, and the decoding algorithms for a non-coherent ultra-wideband receiver are explained in Chapter 2. The synchronization algorithm is also analyzed. Chapter 3 presents the MATLAB implementation of the receiver. The synchronization algorithm is simulated, so that its parameters may be chosen to minimize the synchronization error. The detection algorithm is simulated in order to minimize the probability of missed detection and false alarm. The decoding algorithm simulation is also presented in order to verify the robustness and the efficiency of a non-coherent UWB receiver. Chapter 4 describes the digital baseband architecture of a UWB receiver. Each module in the system is discussed, and the whole system is fully tested. The hardware testing system and the digital system performance are discussed in Chapter 5. Finally, Chapter 6 presents the conclusion.

1.5 Signal Model

The transmitted signal used for this paper is based on the Binary Pulse Position Modulation (BPPM) [16]. The bit interval T_b is divided into two equal time slots with length $T_b/2$. The pulses in the first time slot define a “0” transmitted symbol, while the pulses in the second slot define a “1” symbol. The width of each pulse is T_c. The BPPM signal is illustrated in Figure 1-3.

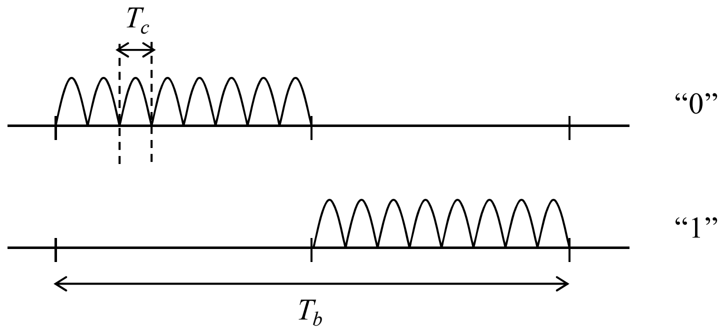


Figure 1-3: Binary pulse position modulation (BPPM) signal.

In this scenario, the transmitted signal from the transmitter is given by:

$$s(t) = \sum_{i=-\infty}^{\infty} c_i \cdot w_{tr}(t - iT_b - a_i T_b/2),$$

where $w_{tr}(t)$ is a burst of transmitted pulses in half a bit period. The c_i's are pseudo-random binary sequences ($c_i = \pm 1$) that serve to smooth out the power spectral density of the transmitted signal. The a_i's are binary independent and identically distributed data symbols taken from the alphabet 0 or 1 (i.e., $a_i \in \{0,1\}$) and T_b is the symbol period. Note that if the a_i's are all zero, a pulse burst will always appear at the beginning of a symbol interval. This is the case for the simple preamble sequence used in this project. Vice versa, when the a_i's are either 0 or 1, the pulse burst starts either at the beginning or at the midpoint of the interval. The data rate is defined by $1/T_b$.

The received signal after the Rx antenna is modeled as:

$$r(t) = \sum_{m=0}^{M} A_m \sum_{i=-\infty}^{\infty} w_{rx}(t - iT_b - a_i T_b/2 - \tau) + n(t),$$

where $w_{rx}(t)$ is the first derivative of $w_{tr}(t)$, M is the number of resolvable paths, A_m defines the gain for path m, and $n(t)$ is a zero-mean additive Gaussian noise. Finally, τ represents an unknown arrival delay at the receiver.

1.6 Receiver Structure

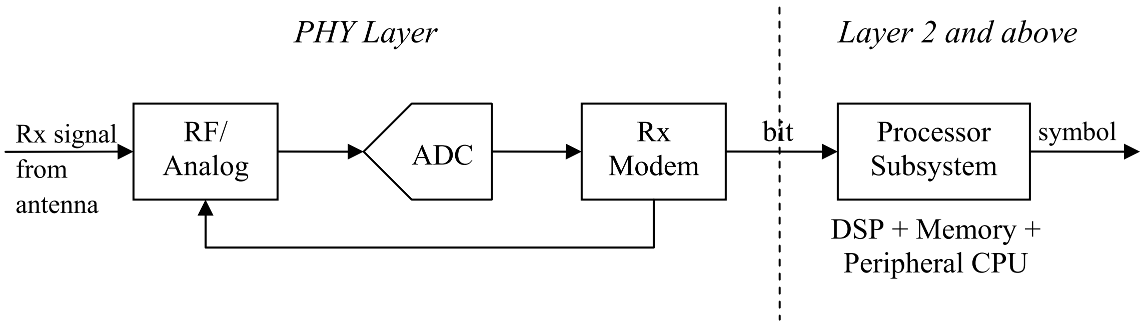


Figure 1-4: High-level block diagram of a UWB receiver.

Figure 1-4 presents the high-level structure of the UWB receiver. This paper focuses mostly on the PHY layer. The detection process, which is executed after the signal is received at an antenna and passed through a band-pass filter, is based on a non-coherent, energy-collection structure (Figure 1-5). For the BPPM signal, the receiver squares and integrates the signal in both time slots to detect the received energy. The decoder calculates the following:

$$z_m = \int_{\hat{t}_{sync}+mT_b/2}^{\hat{t}_{sync}+(m+1)T_b/2} r^2(t)dt\,,$$

for $m = 0$ and $m = 1$, where $\hat{t}_{synch}$ is the integration starting point for the first integration time slot. The decision device sets $\hat{a}_k = 0$ or $\hat{a}_k = 1$ according to the rule:

$$\hat{a}_k = \begin{cases} 0, & \text{if } z_0 > z_1 \\ 1, & \text{otherwise} \end{cases}.$$

Specifically, the receiver measures the energy of the received signal $r(t)$ in the two parts and selects the symbol corresponding to the maximum energy.

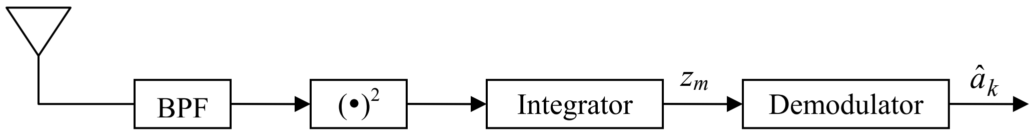


Figure 1-5: Block diagram of the receiver front-end.

1.7 Channel Model

The analysis of the synchronization and the detection algorithms is based on AWGN (additive white Gaussian noise) and non-ISI (non-intersymbol interference) multipath channels. The noise signal is generated for different signal-to-noise ratio (SNR) values. The unknown arrival delay at the receiver is also modeled as a random variable. The time-dispersive effect of the channel plays a fundamental role in the achievable data rate of the system.

Chapter 2

Receiver Algorithms and Analysis

This chapter describes the synchronization, detection, and decoding algorithms for a non-coherent ultra-wideband receiver. The synchronization algorithm is proposed in [4]. The author's key contribution is on the detection and the decoding algorithms. The receiver constantly decides whether the pre-determined preamble signal is present. If the preamble signal is detected, synchronization begins and the system looks for the right instant, at which to start integrating the received signal for energy-collection decoding. The receiver produces decoded bits after the system is synchronized. The system then compares bits with the 11-bit Barker code. This sequence is used to mark the start of the header bits and is called the start frame delimiter (SFD). If the received bits match the SFD Barker code, then header and payload bits follow. The header bits specify the length of the payload. Specifically, the 8-bit header tells how many bytes there are in the payload section. Figure 2-1 illustrates the signal packet structure.

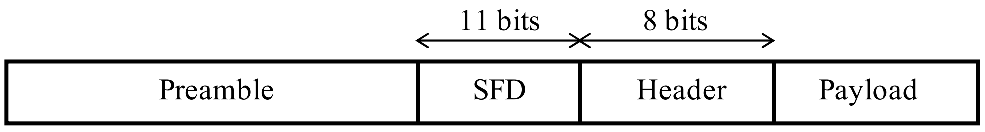


Figure 2-1: Packet structure.

2.1 Synchronization Algorithm and Analysis

2.1.1 Synchronization Algorithm

This section discusses a possible synchronization scheme based on heuristic arguments. In a non-coherent UWB receiver, the synchronization stage should be based on the energy-collection approach, as should the receiver decision scheme, in order to maintain the low complexity of the receiver [13-15]. The synchronization algorithm presented in this paper is developed from the energy-collection scheme proposed in [17] and [4]. We first define the synchronization time delay t_{sync} as the delay that leads to the maximum information signal energy collection for the transmitted symbol in the associated data symbol time slot. Ideally, for an additional white Gaussian noise (AWGN) single-path channel, the synchronization point corresponds to the beginning of the data symbol slot, where all the received signal energy appears in one integrator. For a multipath channel, the correct synchronization time is the delay that maximizes the information signal energy collection.

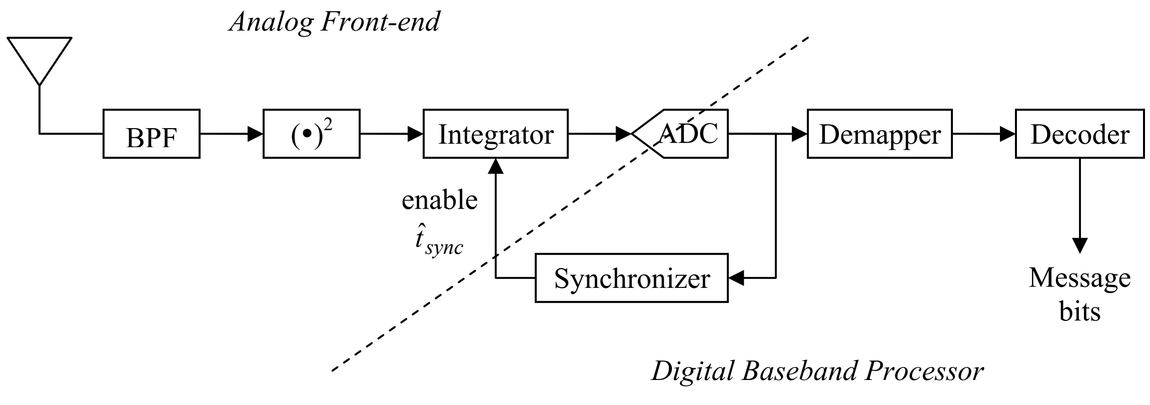


Figure 2-2: Block diagram of a receiver with synchronization process.

The synchronizer performs a serial search and selects the maximum digitized energy corresponding to each integrating window frame. The synchronizer is implemented entirely in the digital domain and produces an output $\hat{t}_{sync}$, which lies in the range [0, T_b]. This output adjusts the starting point of integration of the transmitted signal energy by enabling the integrator after a delay of $\hat{t}_{sync}$ (Figure 2-2). For a single-path channel, the synchronization time $\hat{t}_{sync}$ enables the integrator exactly at the beginning of the data symbol slot, where all the received signal energy appears in one integrator.

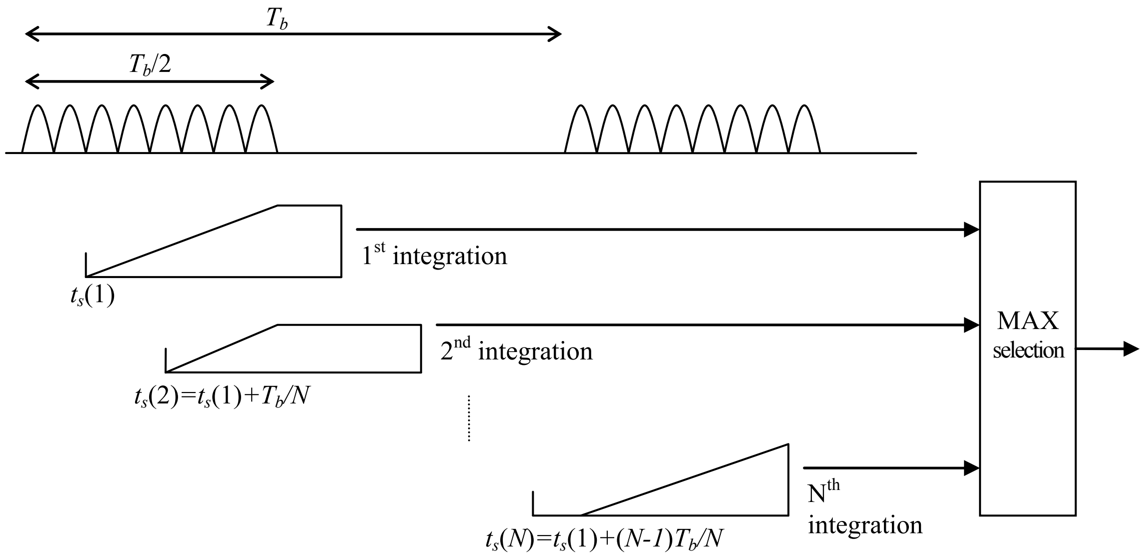


Figure 2-3: Sequential-search synchronization algorithm.

The synchronization process starts after the detector detects a train of preamble symbols, which contain Z bits of all 0's; that is, a pulse always appears at the beginning of a symbol interval. In the digital implementation, the synchronization stage uses one integrator. The integrator has an integration window of $T_b/2$, where T_b is the symbol interval. Let N be the number of integration starting points or "integration phases." The space between each integration phase is, therefore, $\lfloor T_b/N \rfloor$. According to Figure 2-3, the synchronization algorithm selects the starting time that maximizes the integral of the received signal energy as the synchronization point. The starting point of the i^{th} integration is given by:

$$t_s(i) = t_s(1) + (i-1)T_b/N ,$$

where $i \in \{1,2,\ldots,N\}$ and $t_s(1)$ is the integration starting point of the first integration. At the end of the preamble (i.e., after time ZT_b), the synchronizer computes the sum of the integrals at each starting integration point over the entire preamble period:

$$R_i = \sum_{j=0}^{Z-1} \left[\int_{t_s(i)+jT_b}^{t_s(i)+T_b/2+jT_b} r^2(t)dt \right] ,$$

for $i \in \{1,2,\ldots,N\}$. The synchronizer selects the maximum energy collection from these integral values. Therefore, the synchronization is correctly achieved when

$$\alpha = \arg\max_i R_i \text{ and } R_\alpha = \max_i R_i .$$

The synchronization point is thus given by the following:

$$\hat{t}_{sync} = t_s(1) + (\alpha - 1)T_b/N .$$

The accuracy of the synchronization algorithm is proportional to the number of integration phases, N. However, the complexity of the receiver increases as the number of phases increases. With N integration phases in an AWGN channel, the serial search algorithm produces the synchronization point value within the error range:

$$\hat{t}_{sync} \in [t_{sync} - \tfrac{T_b}{2N}, t_{sync} + \tfrac{T_b}{2N}],$$

where t_{sync} is the true optimal synchronization point [12]. As N increases, the synchronization algorithm becomes more accurate. However, the implementation of the digital baseband for the synchronization process becomes more complex with more power consumption and larger circuit area. This project aims to determine the optimal number of integrators (N) and the number of “0” bits (Z), which produce a reasonable synchronization performance and maintain the low complexity of a non-coherent UWB receiver. Chapter 3 presents the synchronization algorithm simulation in MATLAB and determines the optimal number of integration phases. The usual energy-collection decoding (section 1.6) is used once the synchronized starting time of integration $\hat{t}_{sync}$ is determined.

2.1.2 Synchronization Analysis in AWGN Channel

The delay of the synchronization starting point after the starting point of the first integrator is to be chosen from the set $\{0,\ T_b/N,\ T_b/2N, \ldots,\ (N\text{-}1)T_b/N\}$. The probability that the synchronization is correct is the probability that the first integral R_1 is greater than the other integral values R_2, R_3,…, R_N [4]. The probability that the first integrator output is the largest is:

$$P_{s|t_s} = \Pr(R_1 > R_2, R_1 > R_3, \ldots, R_1 > R_N \mid t_s),$$

where $t_s \in [0,\ T_b/N]$. The probability of synchronization is obtained by:

$$P_s = 2\int_0^{T_b/2N} P_{s|t_s}\, p_I(t_s)\, dt_s \text{ , where } p_I(t_s) \sim U[0, T_b/2N].$$

$$\therefore\ P_s = \tfrac{4N}{T_b} \int_0^{T_b/2N} P_{s|t_s}\, dt_s \,.$$

This project aims to choose the optimal value for N by plotting the probability of failure versus the signal-to-noise ratio (SNR) for different values of N. The probability of failure is essentially one minus the probability of synchronization discussed above. The desirable number of integration phases N must achieve a low probability of synchronization failure for a given level of signal-to-noise ratio. The SNR is defined by:

$$SNR = E_b / N_0 - B_w T_b \ ,$$

where E_b is the signal energy, N_0 is the noise energy, and B_w is the signal bandwidth [4]. We can also plot BER performance of the receiver versus SNR to measure synchronization error for an AWGN channel.

Chapter 3 determines the probability of synchronization error for various values of SNR by simulation. The simulation results illustrate how the performance of the receiver changes when the condition of the channel varies.

2.2 Detection Algorithm

As illustrated in Figure 2-1, synchronization only begins when the receiver detects the preamble signal. The detection process determines whether the preamble signal or noise is received. In a non-coherent UWB receiver, the detection stage should be based on the energy-collection approach, as should the receiver decision scheme, in order to maintain the low complexity of the receiver. The detection algorithm is similar to the synchronization algorithm described in the previous section. The detector runs continuously.

The detection process decides whether the preamble signal is received and triggers the synchronization process when the transmitter sends a train of the preamble signals, which contains bits of all 0's; that is, a pulse always appear at the beginning of a symbol interval. The detection stage integrates over N_d phases, where $N_d < N$. That is, the number of integration phase used in the detection process is less than that in the synchronization process. We do not need accuracy to determine the exact integration interval during the detection process. However, we need to make sure that the incoming signal is a sequence of all "0" BPPM bits, not just an AWGN noise. As in the synchronization stage, each integrator has an integration window of $T_b/2$, where T_b is the symbol interval. Therefore, the space between each integration phase is $\lfloor T_b / N_d \rfloor$.

For an AWGN channel, the detection algorithm selects one of the N_d integration phases that maximizes the integral of the received signal energy. A "winner" $\{\alpha_k\}_{k=1}^{W}$ is defined as the phase that has the maximum energy when the energy collection process covers Z_d bits of all 0's. The process of choosing a "winner" is repeated W times. The receiver declares that it detects the preamble signal when one particular phase wins D times. The detection process is halted, and the synchronization process then begins. If there is no phase that "wins" at least D times, the receiver declares that it does not detect the preamble signal. The detection process is then repeated until the receiver detects the preamble signal.

The analysis for the energy-collection detection scheme is similar to the analysis for the synchronization algorithm. The main difference is that the detector needs to keep track of the phase "winners." According to Figure 2-3, the starting point of the i^{th} integration is given by:

$$t_d(i) = t_d(1) + (i-1)\, T_b / N_d \,,$$

where $i \in \{1,2,\ldots,N_d\}$ and $t_d(1)$ is the integration starting point of the first integration. At the end of the preamble (i.e., after time Z_dT_b), the detector computes the sum of the integrals at each starting integration point over the entire period of length Z_dT_b:

$$R_i = \sum_{j=0}^{Z_d-1} \left[\int_{t_d(i)+jT_b}^{t_d(i)+T_b/2+jT_b} r^2(t)dt \right] ,$$

for $i \in \{1,2,\ldots,N_d\}$. The detector selects the maximum energy collection from these integral values. Therefore, a phase "winner" is determined by:

$$\alpha_k = \arg\max_i R_i \text{ and } R_{\alpha_k} = \max_i R_i ,$$

for $k \in U = \{1,2,\ldots,W\}$. That is, the process of choosing a "winner" over a window period of Z_dT_b is repeated W times. Note that $\alpha_k \in \{1,2,\ldots,N_d\}$ for all $k \in U = \{1,2,\ldots,W\}$.

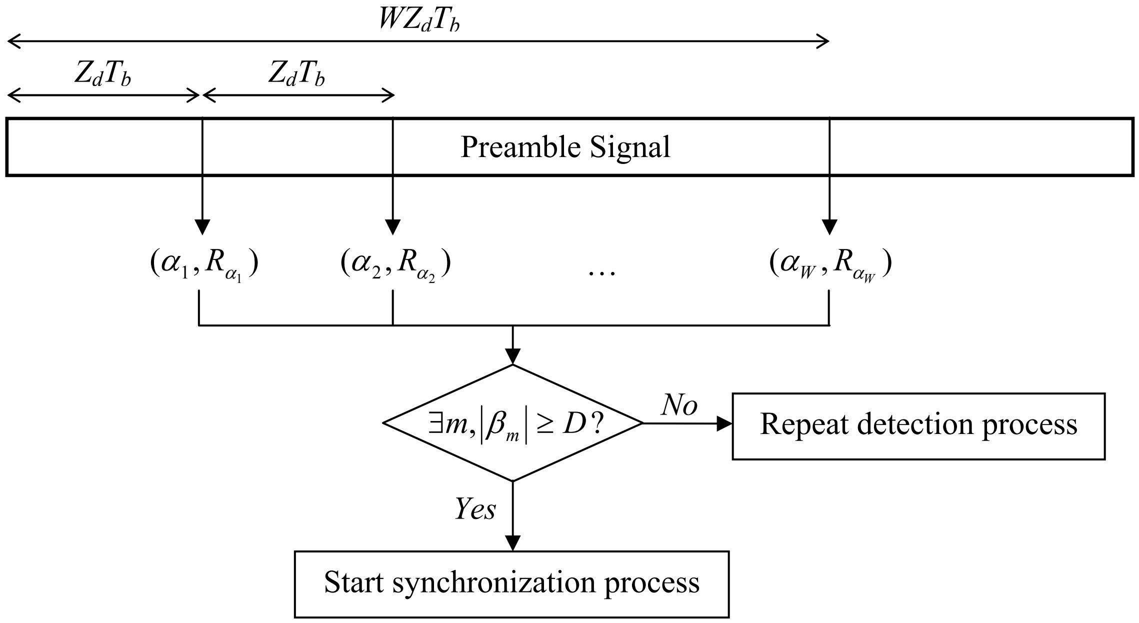
Let $\beta_1 = \left\{ k \mid \alpha_k = 1, \forall k \in U \right\}$,

$\beta_2 = \left\{ k \mid \alpha_k = 2, \forall k \in U \right\}$,

…

and $\beta_{N_d} = \left\{ k \mid \alpha_k = N_d, \forall k \in U \right\}$.

If there exists $m \in \{1,2,\ldots,N_d\}$ such that $|\beta_m| \geq D$, then the receiver declares that it detects the preamble signal. The detection process is halted, and the synchronization process then begins. If there does not exist $m \in \{1,2,\ldots,N_d\}$ such that $|\beta_m| \geq D$, then the receiver declares that it does not detect the signal and the detection process is then repeated. If the preamble signal is transmitted, the phase "winners" should be consistent and $|\beta_m| \geq D$ should be satisfied. On the other hand, if the preamble signal is not transmitted, the phase "winners" will randomly vary and the condition $|\beta_m| \geq D$ will not be satisfied for all values of $m \in \{1,2,\ldots,N_d\}$. The diagram of the overall detection algorithm is presented in Figure 2-4.



WZ_dT_b
Z_dT_b
Z_dT_b
Preamble Signal
(α_1, R_{α_1})
(α_2, R_{α_2})
$\ldots$
(α_W, R_{α_W})
$\exists m, |\beta_m| \geq D?$
No
Repeat detection process
Yes
Start synchronization process

Figure 2-4: Overall detection algorithm.

Chapter 3 aims to determine the optimal number of integration phases (N_d), the optimal number of bits of "0" (Z_d), the number of windows to declare a phase "winner" (W), and the number of "winners" to declare detection (D) that minimize the probability of missed detection and false alarm. The four parameters must maintain low complexity of a non-coherent UWB receiver. The detection algorithm is simulated in MATLAB in order to specify the four optimal parameters.

2.3 Decoding Algorithm

After the signal is synchronized, the decoding process begins and the receiver outputs bits. The bits sent by a transmitter contain an 11-bit Barker start frame delimiter (SFD) code, an 8-bit header, and payload bits as shown in Figure 2-1. The goal of the decoding algorithm is to minimize the payload bit error rate. The energy-collection decoding algorithm is explained in section 1.6.

2.3.1 SFD Matching Algorithm

After the synchronization process, the most recent eleven bits are compared to the known 11-bit Barker code, *a[10:0]* = 00011101101. If all eleven bits match the Barker code, then the receiver knows that the next eight bits belong to the header section. Figure 2-5 shows the diagram of the SFD matching algorithm.

According to Figure 2-5, the SFD matching algorithm begins by operating XNOR on each of the eleven most recent decoded bits, *x[10:0]*, with each corresponding bit of the 11-bit Barker code. If the bits are matched, then the result from the XNOR operator is 1; otherwise, the result is 0. The results from all eleven XNOR operators are accumulated. If the sum of the results is eleven, then the SFD codes are detected and the receiver starts decoding the header and the payload bits. If the sum of the results is less than eleven, then the receiver declares that it does not detect the SFD code. The SFD matching process is then restarted with the updated decoded bits (i.e. shifted to the left). If the SFD matching process continues until the timeout limit is reached, then the receiver declares that it does not detect the header and the payload. The header and the payload bits are, therefore, not decoded. The receiver system then starts over once again with the detection process. The simulation of the SFD matching algorithm is presented in Chapter 3.

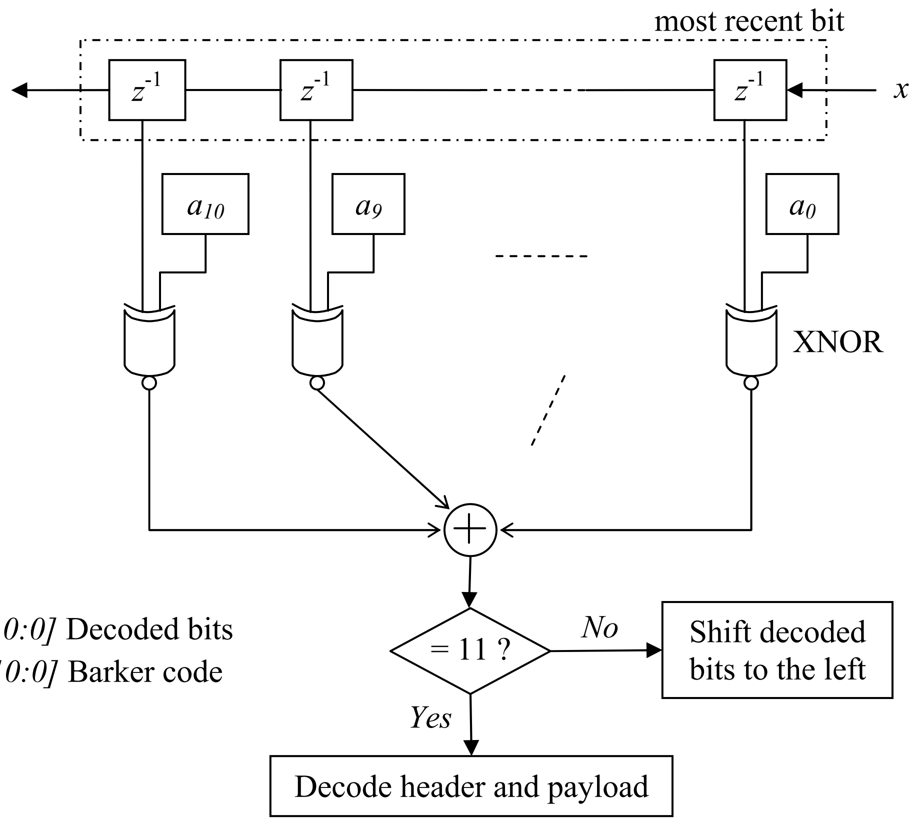


Figure 2-5: SFD matching algorithm.

2.3.2 Header and Payload

The 11-bit SFD Barker code is followed by eight header bits. The header bits specify the length of the payload data bits. Specifically, the header bits specify the number of bytes of the payload. The payload bits constitute information sent by the transmitter.

Chapter 3

Receiver Implementation in MATLAB

3.1 Synchronization Algorithm Simulation

3.1.1 MATLAB Simulation System

This section focuses on the synchronization algorithm and the system simulation in MATLAB for a non-coherent ultra-wideband receiver. The synchronization algorithm must be able to detect the position of the signal in a pulse and to calculate the synchronization point. The usual energy detection (section 2.3) is then performed when the synchronized starting time of integration $\hat{t}_{sync}$ is specified.

It is difficult to achieve very precise synchronization required by a coherent ultra-wideband receiver. However, a non-coherent ultra-wideband receiver has less stringent requirement for the synchronization accuracy. Thus, the synchronization algorithm for a non-coherent ultra-wideband receiver can be developed to achieve synchronization with higher inaccuracy but much lower implementation complexity than a coherent receiver. Therefore, this section aims to simulate the parallel search synchronization discussed in Chapter 2 so that the optimal number of integration phases (N) and the number of preamble bits used in the process (Z) can be determined. The optimal parameters, which are determined by MATLAB simulation results, should produce a reasonable synchronization performance and maintain the low-complexity nature of a non-coherent ultra-wideband receiver.

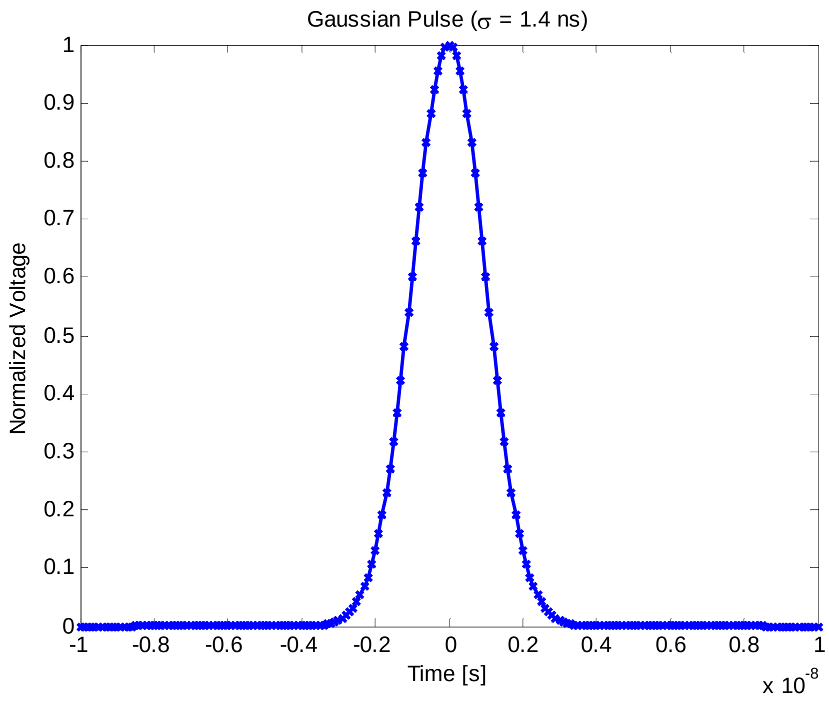


Figure 3-1: Gaussian pulse with σ = 1.4 ns and T_c = 2 ns.

The binary pulse position modulation (BPPM) received signal is generated by a MATLAB function. A pulse in the time domain is modeled as a Gaussian distribution with a standard deviation σ of 1.4 ns. In Figure 3-1, a pulse centered at time τ can be modeled according to the following equation:

$$y(t) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(t-\tau)^2/2\sigma^2}.$$

The chip period T_c is defined as the width of the pulse up to the point where the signal decays. For the MATLAB simulation, the chip period T_c is set to 2 ns, which corresponds to a pulse with a one-sided bandwidth of 250 MHz. When the signal is modulated up to passband by the carrier with frequency f_c, the signal bandwidth is 500 MHz. Figure 3-2 shows the power spectral density of the Gaussian pulse in Figure 3-1. For the ultra-wideband technology, the carrier frequency f_c is comparable to the signal passband bandwidth of 500 MHz. The incoming signal is over-sampled in the time domain so that the Gaussian-shaped pulses are modeled accurately in MATLAB. Furthermore, the bit period T_b is modeled to be $32 \times T_c$ or 64 ns. That is, each time slot in the chip period consists of 16 consecutive Gaussian-shaped pulses.

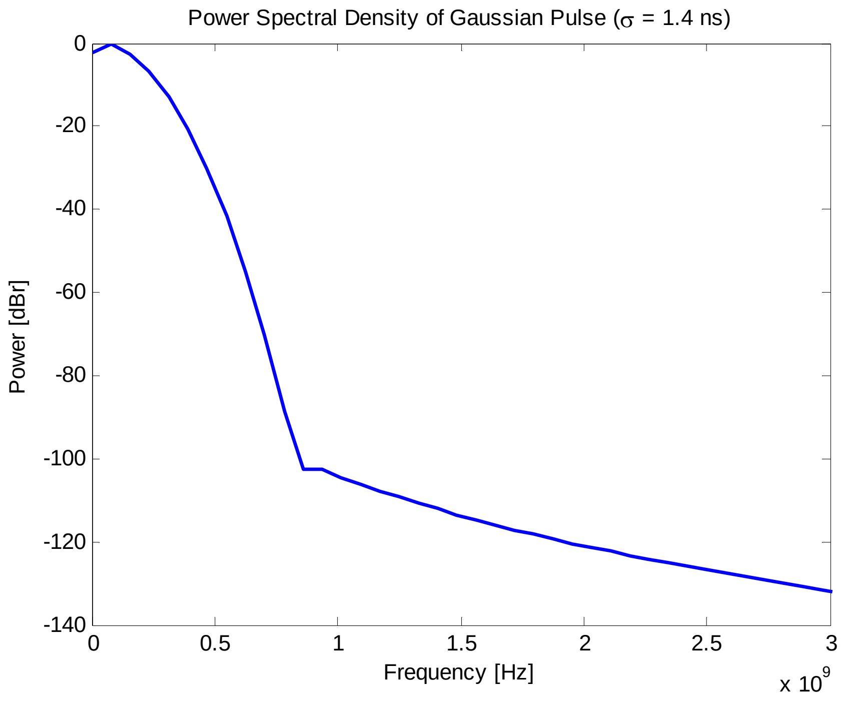


Figure 3-2: Power spectral density of Gaussian pulse with σ = 1.4 ns.

Figure 3-3 depicts the transmitted signal sequence in details. The preamble signal, which consists of a train of all 0's, is used to test the functionality of the detector and the synchronizer. The burst length and pulse width are specified according to Figure 3-1.

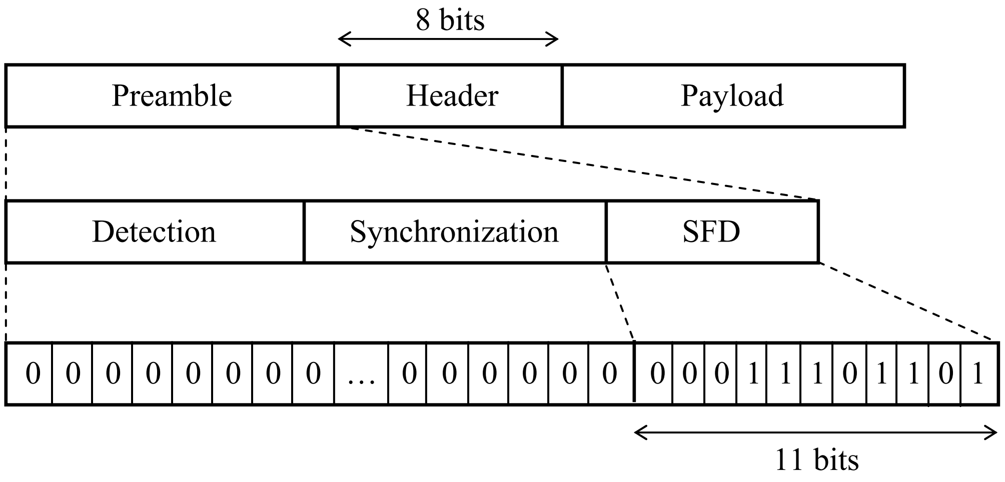


Figure 3-3: Transmitted signal sequence.

The goal for the MATLAB synchronization simulation is: 1) to determine the optimal number of integration phases (*N*) and the number of preamble bits used for synchronization (*Z*) that minimize the probability of synchronization error and the time to synchronize, and 2) to plot the probability of synchronization versus SNR for different number of integration phases (*N*) used for the synchronizer. The low-complexity nature of a non-coherent UWB receiver has to be maintained.

3.1.2 MATLAB Simulation Results

The MATLAB simulation results for the synchronization algorithm explained in section 2.1 are presented in this section. The optimal number of integration phases (*N*) and the optimal number of preamble bits (*Z*) are determined from MATLAB simulation. The function *synchNonCoherent* in MATLAB models the synchronization algorithm. The MATLAB code of the functions can be found in the appendix.

The function *synchNonCoherent* has two main parameters and an output. The two important parameters are *phaseSpace*, which is the space between each integration phase, and *numAve*, which is the number of “0” bits in the preamble signal used for synchronization. Note that *phaseSpace* corresponds to $\lfloor T_b/N \rfloor$ as explained in Chapter 2. *numAve* is exactly the variable *Z* described in the previous chapter. The output of the function is the time index of the synchronized point. The receiver jumps to that point and begins decoding the bits after the synchronization process is finished. The simulation is run 1,000 times for each set of parameter values to determine the probability of synchronization error.

The synchronization error within $\pm 2T_c$ means that the synchronization function fails if the time index of the synchronized point determined from *synchNonCoherent* function differs greater than $\pm 2T_c$ with respect to the ideal synchronization point. The optimal parameters are determined when the probability of synchronization error is less than or equal to 0.5 percent. Tables A1 to A5 in the appendix present MATLAB simulation results for the probability of synchronization error when *numAve* varies. The numbers in bold indicate the minimum *numAve* such that the synchronization error is less than 0.5 percent.

For each simulation in Tables A1 to A5, the beginning of the preamble signal is truncated randomly over the interval of T_b to model random start. Specifically, the MATLAB simulation models the system in a way that the detection process starts anywhere over the first interval of time T_b with uniform probability distribution. Note that the signal-to-noise ratio (SNR) is fixed to 0.0 dB for all simulations in Tables A1 to A5. The optimal parameters determined from the simulations would work even in a very noisy AWGN channel because we use 0.0 dB SNR for channel simulation. Figure 3-4 plots the results in Table A1. The more "0" bits the receiver covers in the integration process, the less probability of synchronization error the receiver achieves.

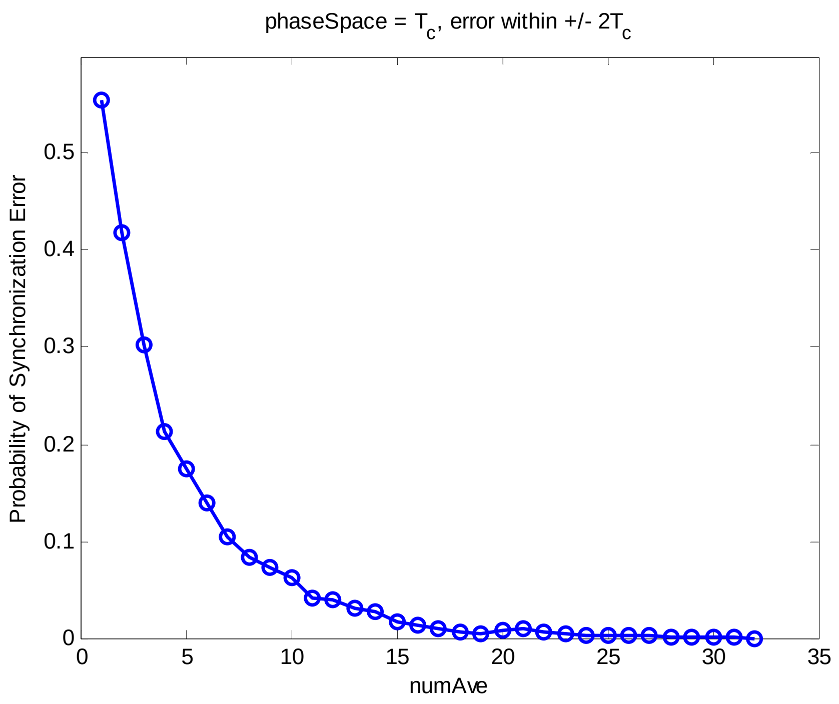


Figure 3-4: Synchronization simulation with *phaseSpace* = T_c and error within $\pm 2T_c$.

The time to synchronize the receiver is given by $numAve \times T_b$. That is, the synchronization time grows linearly with the number of bit periods of integration. Table 3-1 summarizes the results from Tables A1 to A5 by presenting the number of "0" bits in the minimum preamble signal that make the synchronization error probability less than 0.5 percent for each *phaseSpace* value. For an error within $\pm 2T_c$, the *phaseSpace* of T_c requires 22 bits of "0"; the *phaseSpace* of $2T_c$ requires 29 bits of "0"; and the *phaseSpace* of $3T_c$ requires 39 bits of "0". It is difficult to achieve a synchronization error probability of less than 0.5 percent for *phaseSpace* of $4T_c$ or greater. This is because the synchronization error lies between –*phaseSpace*/2 and *phaseSpace*/2 with a uniform probability distribution. Therefore, in order to achieve the synchronization error probability of less than 0.5 percent, we need to allow an error within $\pm 3T_c$ for the *phaseSpace* of $4T_c$.

phaseSpace	Error within	*numAve*	% synch error
T_c	$\pm 2T_c$	22	0.4
$2T_c$	$\pm 2T_c$	29	0.5
$3T_c$	$\pm 2T_c$	39	0.5
$4T_c$	$\pm 2T_c$	-	-
$4T_c$	$\pm 3T_c$	17	0.5

Table 3-1: Synchronization simulation to determine the minimum number of "0" bits in the preamble signal that makes the synchronization error rate no more than 0.5% at 0 dB SNR

According to the simulation results, the synchronization error probability of less than 0.5 percent for a phase space of T_c and error within $\pm 2T_c$ can be achieved with *numAve* of 22 so that the minimum time to synchronize is $\frac{1}{2} \times 22 \times 32 \times T_b$ (note that the integrator integrates over a period of $T_b/2$; so we can collect energy of two phases in one bit period), which equals 22.53 μs. For the synchronization process, we use *phaseSpace*, which equals $\lfloor T_b/N \rfloor$, of T_c and *numAve*, which equals Z, of 22. The phase space for the synchronization scheme is T_c so that the error rate within T_c still remains low. We cannot achieve an error rate within T_c if the phase space is $2T_c$. Therefore, the integration phases (N^*) is $T_b/T_c = 32$. These parameters are determined for the case when the signal-to-noise ratio (SNR) is 0.0 dB. Figures 3-5 and 3-6 plot the probability of synchronization versus the SNR of the AWGN channel. The actual results can be found in Tables A6 and A7 in the appendix. Note that for a very low SNR, the probability of synchronization varies linearly with the logarithm of SNR.

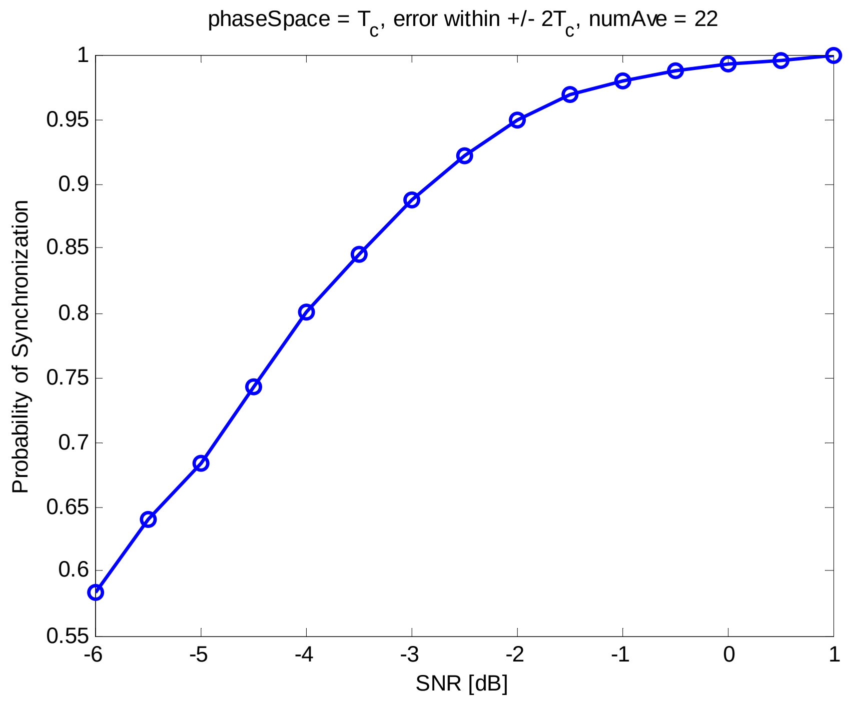


Figure 3-5: Probability of synchronization with *phaseSpace* = T_c, error within $\pm 2T_c$, and *numAve* = 22.

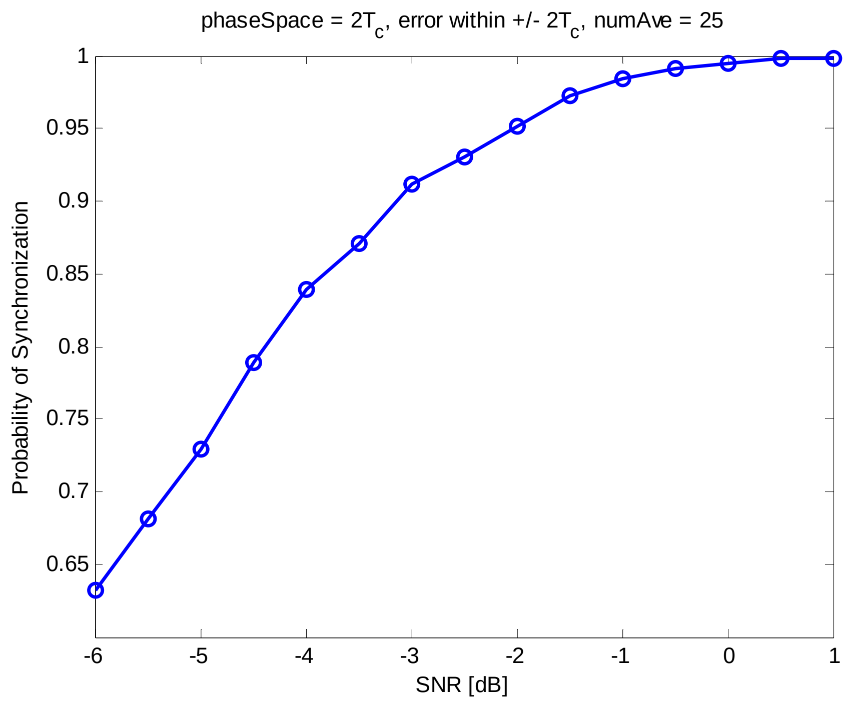


Figure 3-6: Probability of synchronization with *phaseSpace* = $2T_c$, error within $\pm 2T_c$, and *numAve* = 25.

3.2 Detection Algorithm Simulation

The MATLAB simulation results for the detection algorithm explained in section 2.2 are presented in this section. The goal for the MATLAB detection simulation is: 1) to determine the optimal number of integration phases (N_d), the optimal number of "0" bits (Z_d), the number of windows to declare the phase "winners" (W), and the number of "winners" to declare detection (D) that minimize the probability of missed detection and false alarm, and 2) to plot the probability of detection error versus *numDetect* for various conditions. The low-complexity nature of a non-coherent UWB receiver has to be maintained.

The function *detection* in MATLAB models the detection algorithm. The MATLAB code of the function can be found in the appendix. The function *detection* has four main parameters and two outputs. The four important parameters are: 1) *phaseSpace*, which is the space between each integration phase; 2) *numAve*, which is the number of "0" bits in the preamble signal used to determine a phase "winner"; 3) *windowSize*, which is the number of windows to declare the phase "winners", and 4) *numDetect*, which is the number of "winners" to declare detection.

phaseSpace corresponds to $\lfloor T_b / N_d \rfloor$ as explained in Chapter 2. *NumAve*, *windowSize*, and *numDetect* are exactly the variables Z_d, W, and D described in the previous chapter, respectively. The main output of the function is a Boolean indicating whether the receiver detects the preamble signal. The receiver will begin the synchronization process if it declares detection of the preamble signal. If not, the receiver will repeat the detection process. Similar to the synchronization simulation, the detection simulation is run 2,000 times for each set of parameters value: 1,000 times where the preamble signal is transmitted and another 1,000 times where the preamble signal is not transmitted, in order to determine the probability of detection error.

Tables A8 to A19 in the appendix present the MATLAB detection simulation results. The four parameters discussed above are varied so that the minimum probability of detection error can be achieved. The probability of detection error is defined as:

$$\begin{aligned}\Pr(\text{error}) = {} & \Pr(\text{preamble signal transmitted}) \times \Pr(\text{missed detection}) \\ & + \Pr(\text{preamble signal not transmitted}) \times \Pr(\text{false alarm}),\end{aligned}$$

where Pr(missed detection) is the probability of not declaring detection when the preamble signal is transmitted, and Pr(false alarm) is the probability of declaring detection when no preamble signal is transmitted. For all simulations, *windowSize* is fixed to 11 so that a detection process is time efficient. The minimum probability of detection error is chosen for each set of parameters.

phaseSpace	*numAve*			
	4	5	6	7
$4T_c$	3.65% (5)	3.05% (5)	1.90% (6)	**1.25% (6)**
$6T_c$	4.35% (6)	4.25% (6)	3.30% (6)	3.20% (6)
$8T_c$	6.45% (6)	7.45% (6,7)	6.95% (6)	6.25% (7)

Table 3-2: Minimum probability of detection error with *windowSize* = 11

Note: The optimal *numDetect* values that minimize the probability of detection error for each case are reported in parentheses below the minimum probability of detection error. Assume equal probability of transmitting the preamble signal.

Table 3-2 summarizes the results reported in Tables A8 to A19. The probability that the preamble signal is transmitted and the probability that the preamble signal is not transmitted are both set to 0.5. Also, the SNR is fixed to 0.0 dB to model a noisy channel. The optimal *numDetect* is chosen to minimize the probability of detection error for each set of the three parameters: *numAve*, *phaseSpace*, and *windowSize*. In the simulations, *numAve* are varied from 4 to 7 so that the detection time is not too long. According to Table 3-2, the minimum probability of detection error can be achieved when *phaseSpace* is $4T_c$, *numAve* is 7, and *numDetect* is 6. The optimal probability of error is 1.25 percent.

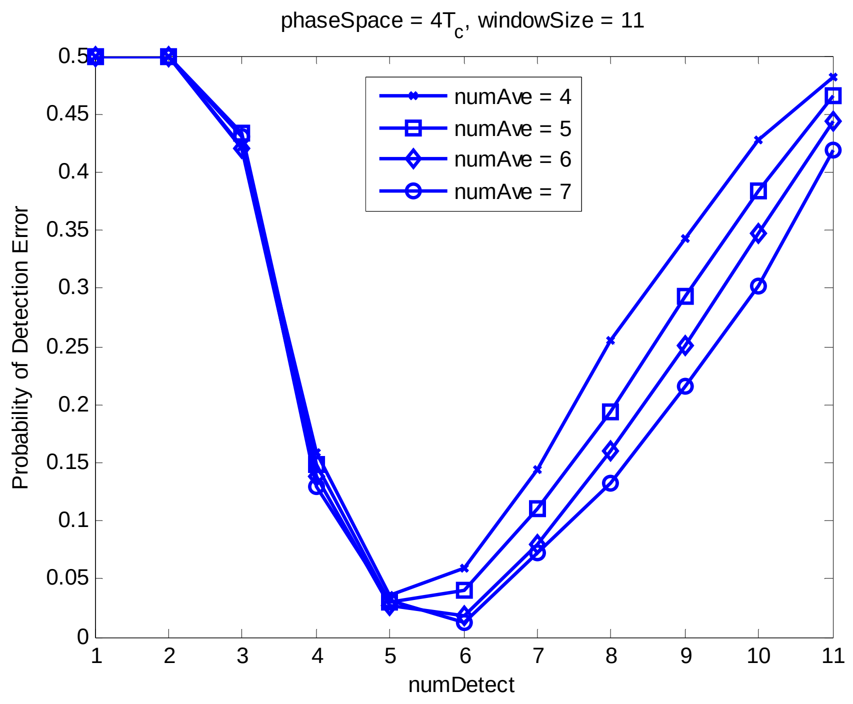


Figure 3-7: Probability of detection error with *phaseSpace* = $4T_c$ and *windowSize* = 11.

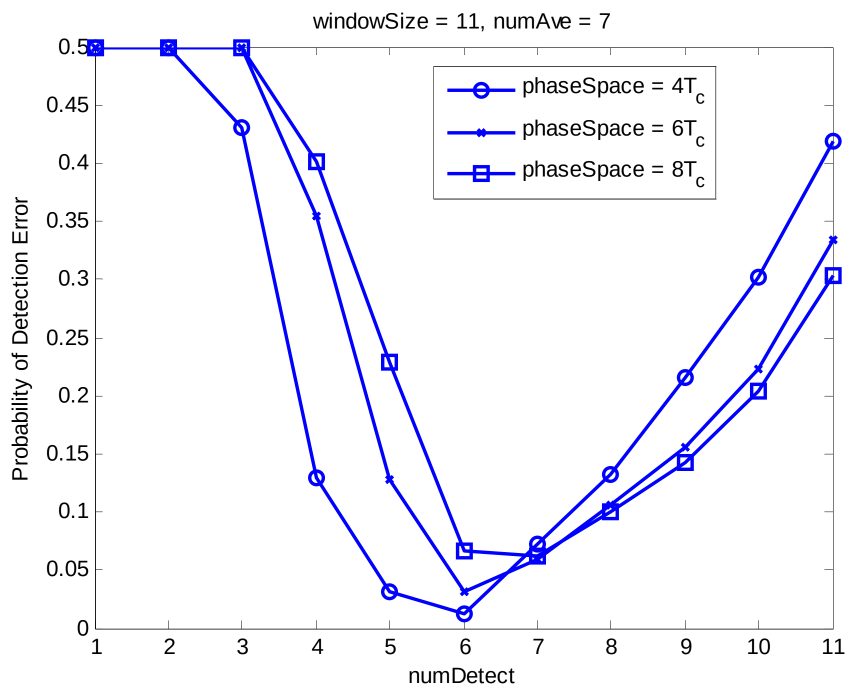


Figure 3-8: Probability of detection error with *windowSize* = 11 and *numAve* = 7.

The four optimal parameters for the detector can thus be determined. Because *phaseSpace* is equal to $\lfloor T_b / N_d \rfloor$, the optimal number of integration phases (N_d^*) is $T_b/4T_c = 8$. The optimal number of "0" preamble bits (Z_d^*) is 7; the number of windows to declare the phase "winners" (W^*) is set to 11; and the optimal number of "winners" to declare detection (D^*) is 6.

Figure 3-7 plots the probability of detection error with *phaseSpace* equal to $4T_c$ and *windowSize* equal to 11. The four curves correspond to different *numAve* values. The more number of preamble bits is used in the detection algorithm, the less probability of detection error the receiver achieves. Figure 3-8 plots the probability of detection error versus *numDetect* for three *phaseSpace* values. For a small *phaseSpace*, the probability of false alarm is small, but the probability of missed detection is large. On the other hand, for a large *phaseSpace*, the probability of false alarm is large, but the probability of missed detection is relatively small.

The probability of detection, which equals 1 minus the probability of detection error, is plotted versus the signal-to-noise ratio (SNR) in Figure 3-9. The four optimal parameters are used in the MATLAB simulation. The simulation results for this plot can be found in Table A20 in the appendix. Note that the probability of missed detection increases as the SNR decreases. However, the probability of false alarm is constant over SNR from -6.0 to 2.0 dB. Therefore, the probability of detection error increases as the SNR decreases because of the missed detection. In other words, as the SNR increases (i.e. less noisy channel), the probability of detection increases because the energy integration values become more accurate.

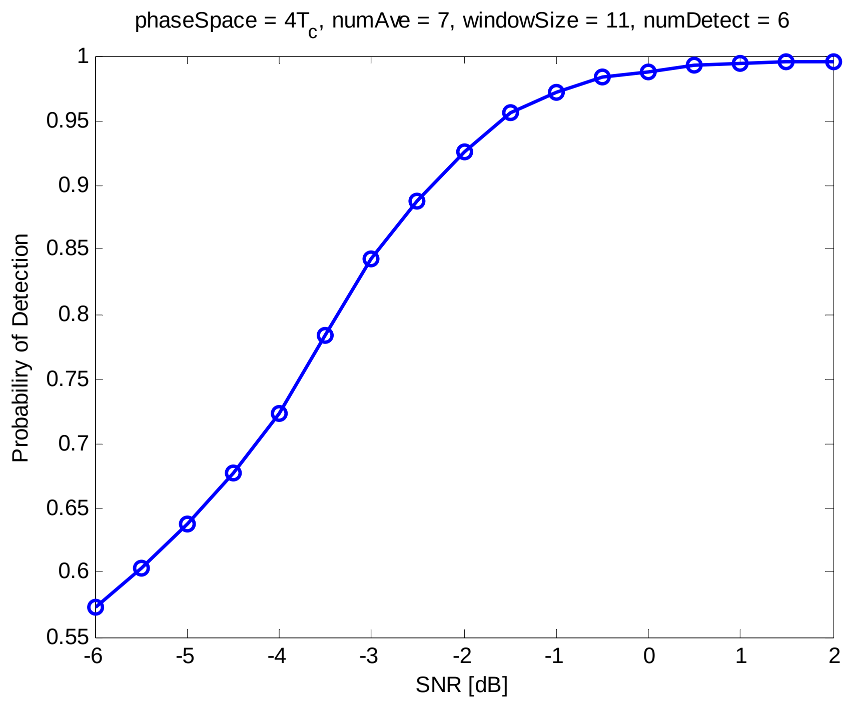


Figure 3-9: Probability of detection error versus SNR with optimal parameters.

3.3 Decoding Algorithm Simulation

Function *uwbSim* shown in the appendix implements and simulates the SFD matching algorithm and decoding algorithm for the header and the payload bits. The sub-function *rxNonCoherent* receives the signal and produces decoded bits by the energy collection scheme explained in section 2.3. The important input to this function is the signal after the synchronization point. That is, the input signal includes the SFD code, the header, and the payload bits. The output of *rxNonCoherent* is the decoded bits determined by the energy collection algorithm.

The MATLAB simulation is run 1,000 times to determine the bit error rate (BER), which equals to the total number of bits in error divided by the total number of bits transmitted. One thousand independent identically-distributed binary bits are transmitted for each simulation. Table A21 in the appendix presents the payload decoding simulation results. The bit error rate (BER) is measured for various values of the signal-to-noise ratio (SNR). The condition of the channel affects the performance of the decoder. Specifically, the bit error rate increases, as the signal-to-noise ratio decreases.

Figure 3-10 plots the natural logarithm of the bit error rate, log(BER), versus the signal-to-noise ratio (SNR). The bit error rate drops sharply (around 2-4 orders of magnitude) as the SNR increases from 0 dB to 7 dB. We can conclude that the bit error rate is less than 10^{-5} for the SNR of greater than 7 dB. Consequently, the decoding algorithm works effectively in a normal AWGN channels.

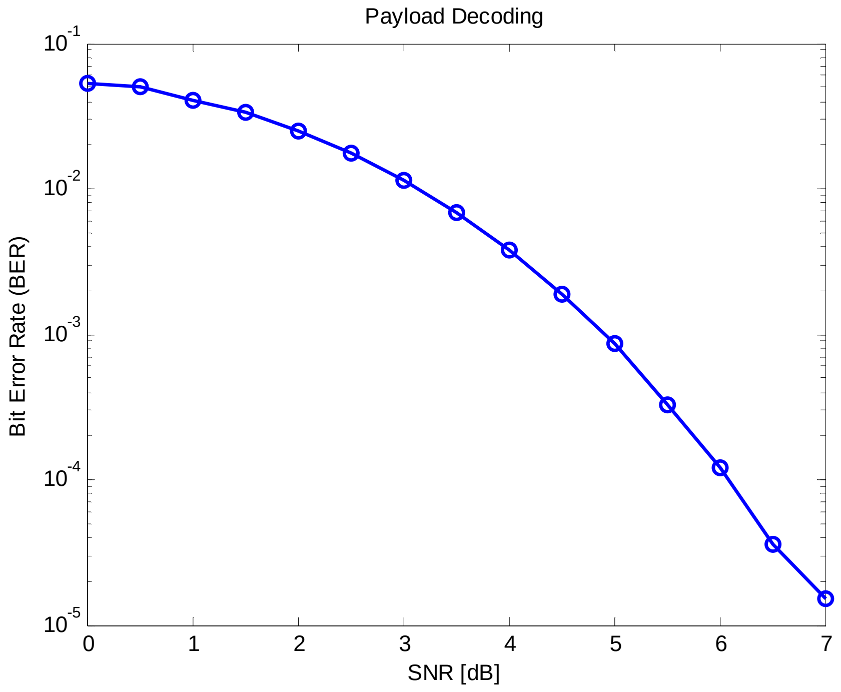


Figure 3-10: Payload decoding simulation.

3.4 Summary

In summary, this chapter determined the optimal parameters for the detection and the synchronization algorithms explained in the previous chapter. The receiver system and algorithms are simulated in MATLAB. Table 3-3 reports the optimal values of important parameters determined in sections 3.1 and 3.2. Chapter 4 describes the digital baseband design and implementation for a non-coherent ultra-wideband receiver.

Parameter	Description	Value
N^*	Number of integration phases for synchronization	32
Z^*	Number of preamble bits used for synchronization	22
N_d^*	Number of integration phases for detection	8
Z_d^*	Number of preamble bits used for detection	7
W^*	Number of windows to declare a phase "winner"	11
D^*	Number of "winners" to declare detection	6

Table 3-3: Optimal parameters determined from MATLAB simulations

Chapter 4

Digital Baseband Architecture

4.1 Digital System Overview

A non-coherent ultra-wideband receiver is implemented in Verilog, a hardware description language (HDL). The Verilog code is then mapped to hardware. This project utilizes the embedded system design technique for realizing the digital system. This technique leverages the advanced capabilities of today's integrated circuit technology by implementing many of the components of the system within a field programmable gate array (FPGA). An FPGA is a good choice for implementing a digital baseband system for an ultra-wideband receiver because it offers large logic capacity, exceeding several million equivalent logic gates, and includes dedicated memory resources. It is also capable of embedding special hardware circuitry that is often needed in digital systems, such as baseband digital signal processing blocks. The Verilog code is implemented along with the rest of the system by using the logic and memory resources in the FPGA fabric. In the final system to be implemented by other members of the research group, a radio frequency (RF) and an analog front-end will interface with the FPGA, resulting in a complete radio receiver.

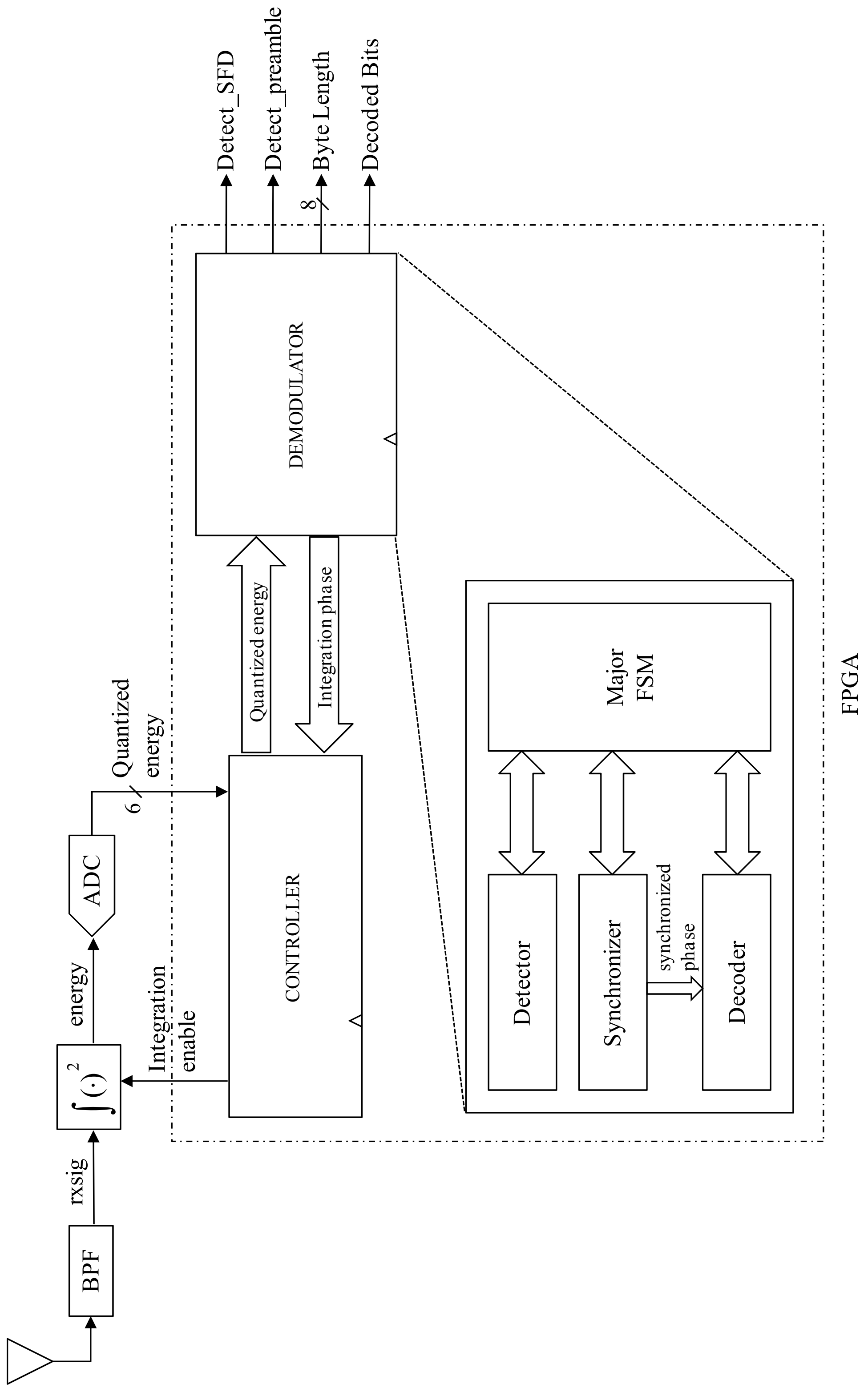


Figure 4-1: High-level block diagram.

Figure 4-1 shows the high-level block diagram of the FPGA along with the analog and the mixed-signal components. The receiver system receives an analog transmitted signal. The integrator and the analog-to-digital converter (ADC) accumulate the energy of the signal. The digital baseband in the FPGA contains a demodulator, which serially runs the detection, the synchronization, and the decoding processes described in the previous chapters. The FPGA outputs the decoded digital bits. With the optimized parameters for the algorithms described in Chapter 3, the bit error rate is small even in a presence of noise.

This chapter presents the digital baseband architecture and design for a UWB receiver. The finite state machines are designed separately for each operational block. The major finite state machine controls the operation and the interaction of all blocks. The digital design is implemented in Verilog.

4.1.1 System Organization

The ultra-wideband receiver system consists of three main modules: RX_MODEL, COUNTER, and DEMOD, as illustrated in Figure 4-2. All blocks except for the square-and-integrate module and the analog-to-digital converter are implemented in an FPGA. The system has eight input signals, and produces three output signals. One of the input signals, *rxsig*, is analog; the rest are digital bits. The receiver system receives a wirelessly transmitted signal and determines the payload bits and their length. Specifically, *decodedBit* signal outputs the payload bits, while *detect_payload* and *byte_length* are the qualifier of the payload (i.e., *detect_payload* goes high only when the *detect_payload* signal is valid) and its length in bytes, respectively.

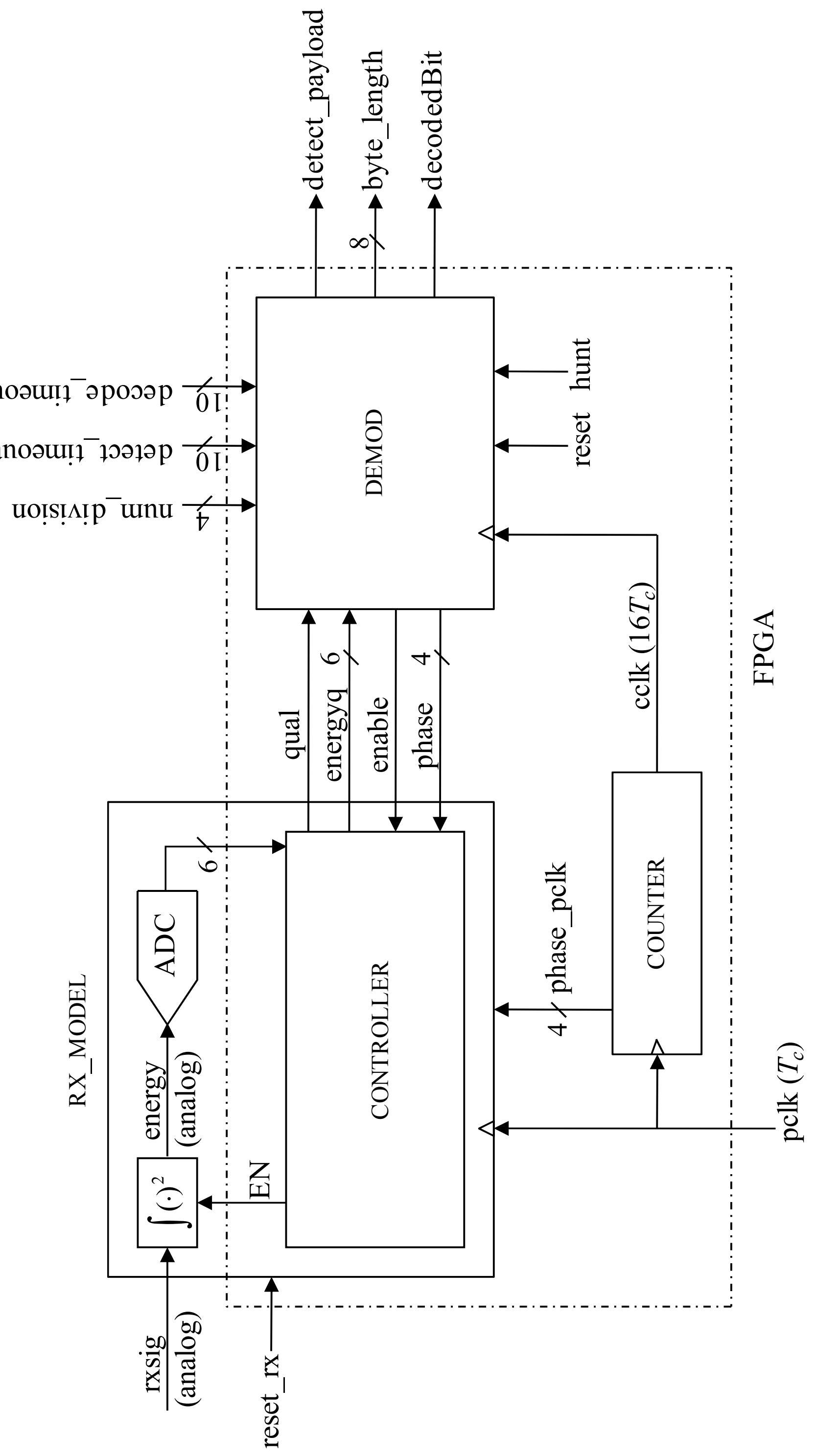


Figure 4-2: Overall block diagram of the receiver system.

The RX_MODEL module receives an analog transmitted signal, *rxsig*, and calculates the energy of the signal over half a chip period, $T_b/2$. This module operates at a clock frequency of $1/T_c$, which is approximately 500 MHz. In other words, the clock period is T_c or 2 ns. Because T_c is the phase space of the synchronization algorithm explained in Chapter 2, this clock is called "phase clock" (*pclk*). The RX_MODEL module controls the integration process and specifies the phase at which the integration begins. This block squares and integrates the signal in analog domain, and then quantizes the energy so that the output, *energyq*, is a 6-bit digital signal. *Qual* is the qualifier of the quantized energy output. All blocks in Figure 4-2 except for the square-and-integrate module and the analog-to-digital converter are implemented by the author.

The COUNTER module is both a phase counter and a clock divider. It counts the phase from 0 to 15 at every clock period T_c. This module also generates a synchronous clock signal with period $16T_c$. That is, the slower clock toggles at every $8T_c$. This clock operates the DEMOD module, which is the core module for the demodulating process. Therefore, this slower clock is called "chip clock" (*cclk*).

The DEMOD module processes the detection, the synchronization, and the decoding algorithms in order to determine the transmitted payload bits. The optimized parameter values determined in Chapter 3 are used in each process in order for the receiver to achieve reliable performance in a presence of additive white Gaussian noise. This module operates with the chip clock (*cclk*), whose frequency is $1/(16T_c)$ or approximately 31.25 MHz. This module runs the three processes only when the *hunt* signal is asserted high. *Num_division* specifies the maximum possible value of the phase for integration. *Detect_timeout* and *decode_timeout* signals indicate the maximum time that the module can execute the detection and the decoding process, respectively. Note that the bit size of each signal is parameterized so that the Verilog code is flexible for any change in bit resolution.

4.1.2 Receiver's Functionality

Figure 4-3 shows the major sequence of operations of the overall receiver system. The demodulator system is initialized when the *reset* signal is asserted. The demodulator then waits until the *hunt* signal goes high to begin the following processes serially. First, the detection process determines whether the receiver receives a train of preamble signals. If the preamble signal is detected, the synchronization process begins. If the preamble signal is not detected, then the detection process is repeated until the demodulator detects the preamble signal or until the detection time reaches the pre-specified timeout limit. Second, the synchronization process specifies the correct phase of integration, at which the integral of the preamble signal over half a chip period is maximized. Finally, the decoding process produces output bits by comparing the energy in the first and the second half of the chip period. If the last eleven bits match perfectly with the eleven-bit Barker SFD code, then the next eight bits specify the length in bytes of the payload. The demodulator repeatedly finds the SFD code from the decoded bits until the SFD searching time reaches the pre-specified timeout limit. After the demodulator finishes all three processes, it waits until the *hunt* signal goes high again.

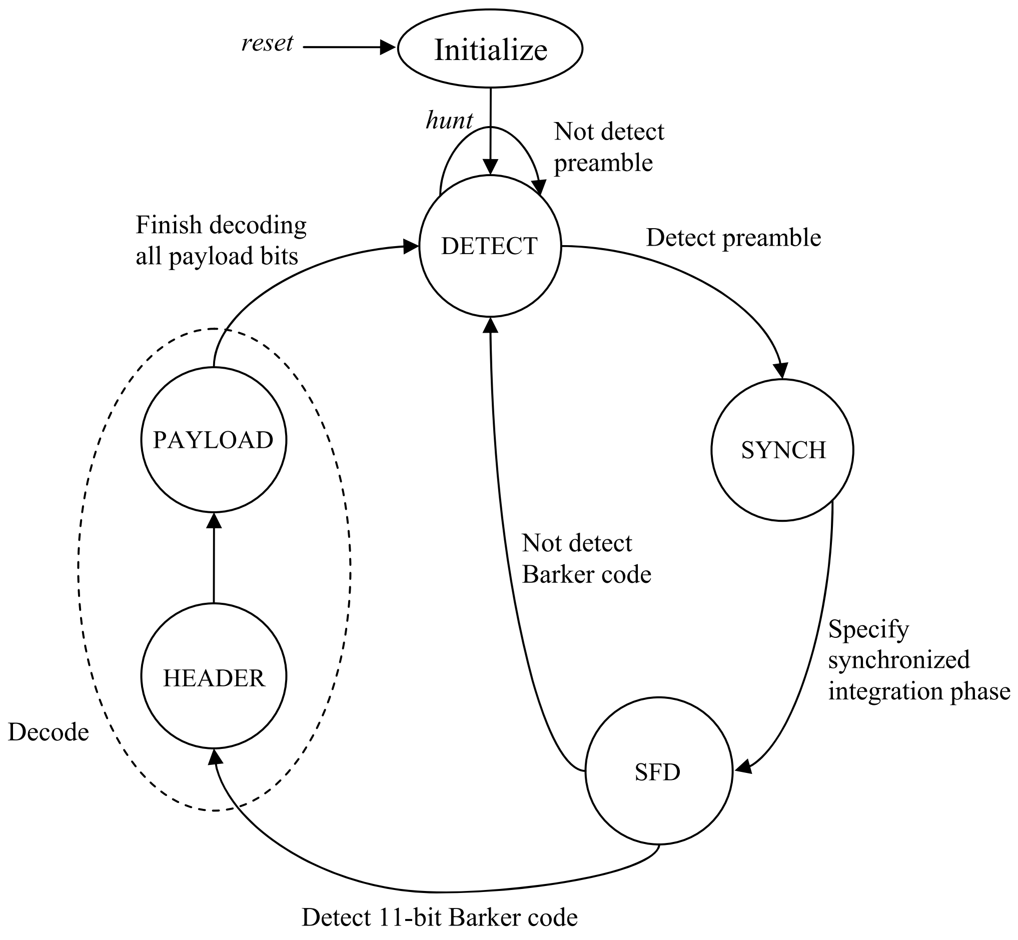


Figure 4-3: Overall control flow.

The timing diagram of the input and the output signals of the demodulator that interact with the RX_MODEL module is shown in Figure 4-4. The demodulator asserts the *enable* signal when it wants the integrator to start integrating the received signal at a specified *phase* number. The RX_MODEL module processes the request from the demodulator and outputs 6-bit quantized energy values along with a qualifier, *qual*. The delay of the integration scheme can be varied over different input/output conditions. Note that the implementation of the detection and the synchronization schemes only changes *phase* forward.

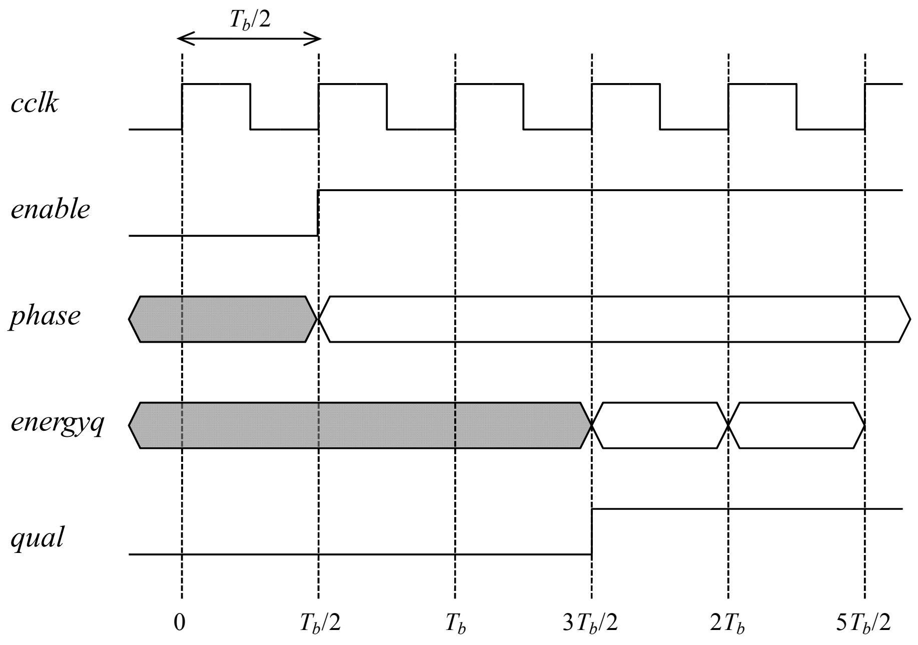


Figure 4-4: Timing diagram of the inputs and the outputs of the demodulator that interact with RX_MODEL module.

The timing characteristics of the output signals of the overall receiver are illustrated in Figure 4-5. The *detect_payload* signal goes high after the demodulator detects the eleven-bit SFD code and the eight-bit length (in bytes) of the payload is decoded. In other words, *detect_payload* is high from when the first payload bit is decoded. *DecodedBit* signal outputs bits at every T_b or $32T_c$. The propagation delay from when the preamble is received to when *detect_payload* goes high is variable and depends on the condition of the channel.

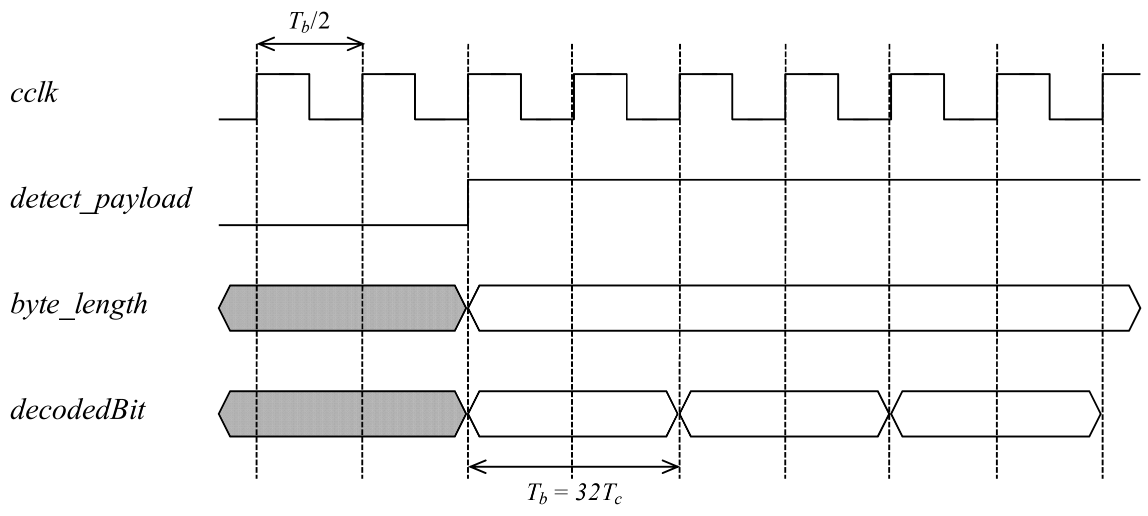


Figure 4-5: Timing diagram of the outputs of the receiver.

4.2 Receiver's Front-end Module Description and Implementation

The receiver's front-end contains two main modules: RX_MODEL and COUNTER. Figure 4-6 presents the block diagram of the receiver's front-end. There are five input and three output signals that feed into the demodulator module. *Rxsig* is the wirelessly transmitted signal, which is observed by the receiver. This signal is in continuous time and has analog value. The *reset_rx* signal resets and initializes the RX_MODEL module. *Pclk* is the phase clock with period T_c. In the actual system, T_c is approximately 2 ns, but a slower clock can be used to test the system operation. The *enable* and *phase* signals come from the demodulator to start the integrating process at the specified phase number. Since one chip period is divided into 16 phases, the *phase* signal contains 4 bits. *Cclk* is the chip clock with period $16T_c$. The receiver's front-end outputs 6-bit quantized energy, *energyq*, every chip period with a qualifier, *qual*.

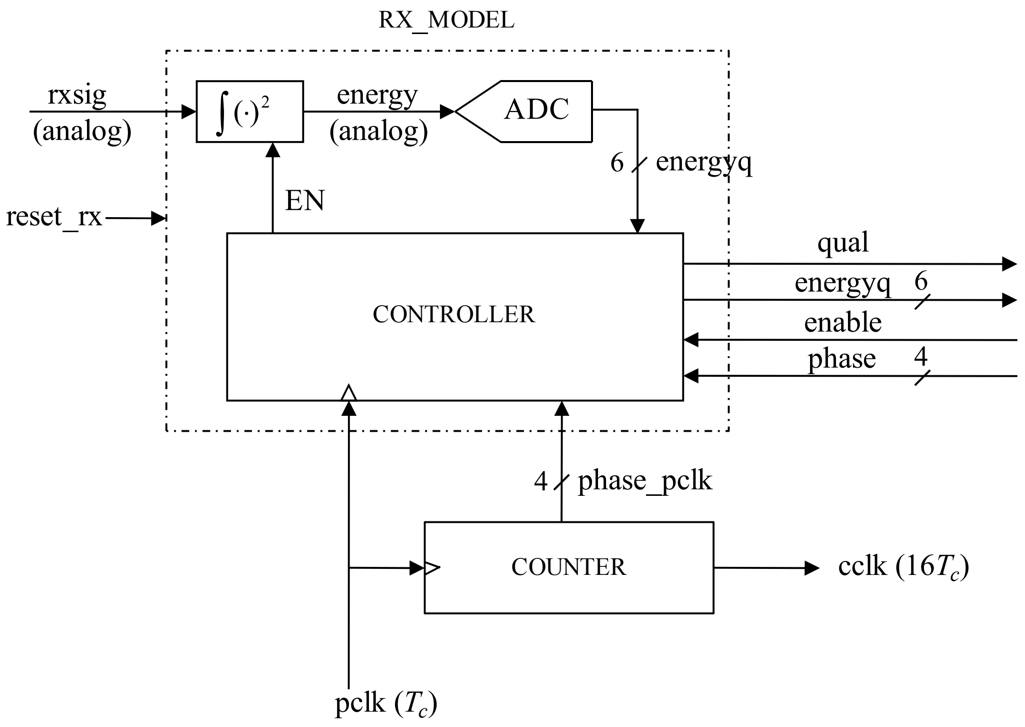


Figure 4-6: Receiver's front-end block diagram in the actual system.

4.2.1 Rx_model

In the final system, the RX_MODEL module consists of the analog square-and-integrate module, the analog-to-digital converter (ADC), and the main controller. Only the controller is implemented in digital baseband. The other two sub-modules are not implemented in this project. However, a digital block that squares and integrates incoming signals is modeled in digital baseband in order to test the RX_MODEL module. On a reset (*reset_rx* is high), the controller stops enabling the integrator (*EN* is low) so that *qual* and *energyq* remain zero. The square-and-integrate module first squares *rxsig* and then integrates the result while the *EN* signal is high. The typical integration period for a non-coherent ultra-wideband receiver is half a chip period, $T_b/2$. The analog *energy* output is the result of the square-and-integrate process. Note that *energy* is zero when *EN* is low to initialize the integrator.

The analog-to-digital converter samples *energy* at frequency 1/(16T_c) or approximately 31.25 MHz and quantizes the signal such that the resulting digitized energy, *energyq*, is a 6-bit digital signal. This *energyq* signal is an important input of the demodulator because the detection, the synchronization, and the decoding algorithms are based on the energy-collection scheme. Moreover, *energyq* changes value only at each positive edge of the chip clock (*cclk*).

The controller is a purely digital module, which operates at the phase clock with frequency T_c. It takes the *enable* and the 4-bit *phase* signals from the demodulator and enables the square-and-integrate module at the right phase. Specifically, *EN* goes high only when *enable* is asserted and the phase counter, *phase_pclk*, is equal to the *phase* input. The controller sets *qual* high when the *energyq* signal is valid (i.e., when the analog-to-digital converter finishes the process and outputs the digital bits).

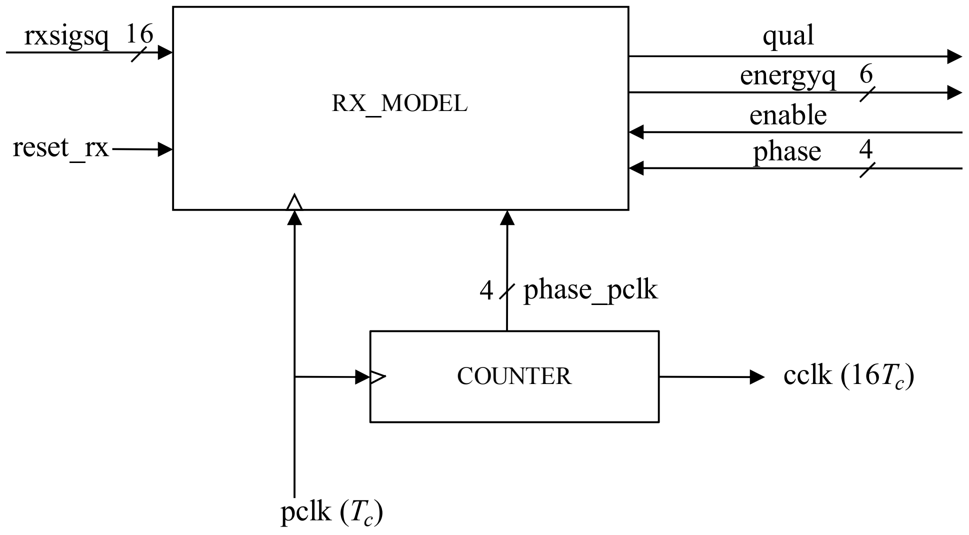


Figure 4-7: Receiver's front-end block diagram in the FPGA testing system.

In order to model the system for testing and debugging in the digital domain, the receiver's front-end is modeled in an FPGA, as shown in Figure 4-7. That is, the square-and-integrate sub-module is modeled in the digital domain. The RX_MODEL module is implemented in Verilog, which is mapped to hardware (FPGA). All inputs and outputs to the receiver's front-end are the same as those in the actual system, except for *rxsigsq*. The 16-bit *rxsigsq* signal is the square of *rxsig*, which is sampled at frequency $1/T_c$. The RX_MODEL module imitates the integral operation by summing the square of the received signal from a point where *phase_pclk* is equal to the *phase* input and the *enable* signal is asserted. The summing period is typically $16T_c$. Since we sum 16 samples of the 16-bit *rxsigsq* signal, the resulting energy has 16+4 = 20 bits. The module then outputs only six most significant bits of the 20-bit energy, resulting in the 6-bit *energyq* output signal. The controller sets *qual* high when the *energyq* signal is valid

4.2.2 Counter

The counter module has two main functions. First, it divides the clock period by a factor of 16. This resulting clock then operates the demodulator. Second, it keeps track of the phase number for each positive edge of the phase clock (*pclk*). The counter module has a 4-bit internal register, *counter[3:0]*, which increments at every positive edge of *pclk*. The *phase_pclk* output is literally equal to *counter*. Since *cclk* transitions every $8T_c$, it is equal to the most significant bit of *counter* (i.e., *cclk* equals to *counter[3]*).

4.3 Demodulator Module Description and Implementation

The demodulator has four main modules and two muxes, as shown in Figure 4-8. Although each module has a specific functionality and responsibility in the system, they interact with each other to perform the desired processes. Each module is discussed in this section.

The four main blocks in the demodulator are: 1) the detector, 2) the synchronizer, 3) the decoder, and 4) the major finite state machine (FSM). There are also two 4-to-1 muxes. The detector determines whether the preamble signal is received. The synchronizer specifies the correct integration phase for the decoder. The decoder produces bits and performs the SFD matching algorithm. The major finite state machine controls the three processes. There are eight input and five output signals for the demodulator. Two of the eight outputs, *enable* and *phase* are fed into the RX_MODEL module; the rest, *detect_payload*, *byte_length*, and *decodedBit*, are the output signals of the entire receiver system. The demodulator module operates at a clock frequency of $1/(16T_c)$. On a reset (*reset* is high), all operations are halted and the module is initialized.

The 4-bit *num_division* input specifies the number of phases per chip clock. *Energyq* and *qual* are the 6-bit quantized energy and its qualifier from the receiver's front-end, respectively. The demodulator waits until the *hunt* signal goes high to process the detection, the synchronization, and the decoding algorithms serially. *Detect_timeout* and *decode_timeout* specify the timeout limit for the detection and the decoding process, respectively. The demodulator starts the integration process of the receiver's front-end by asserting the *enable* signal and assigning the beginning phase of integration to the 4-bit *phase* output. *Detect_payload* is a qualifier of *decodedBit*. *Byte_length* is the length in bytes of the payload.

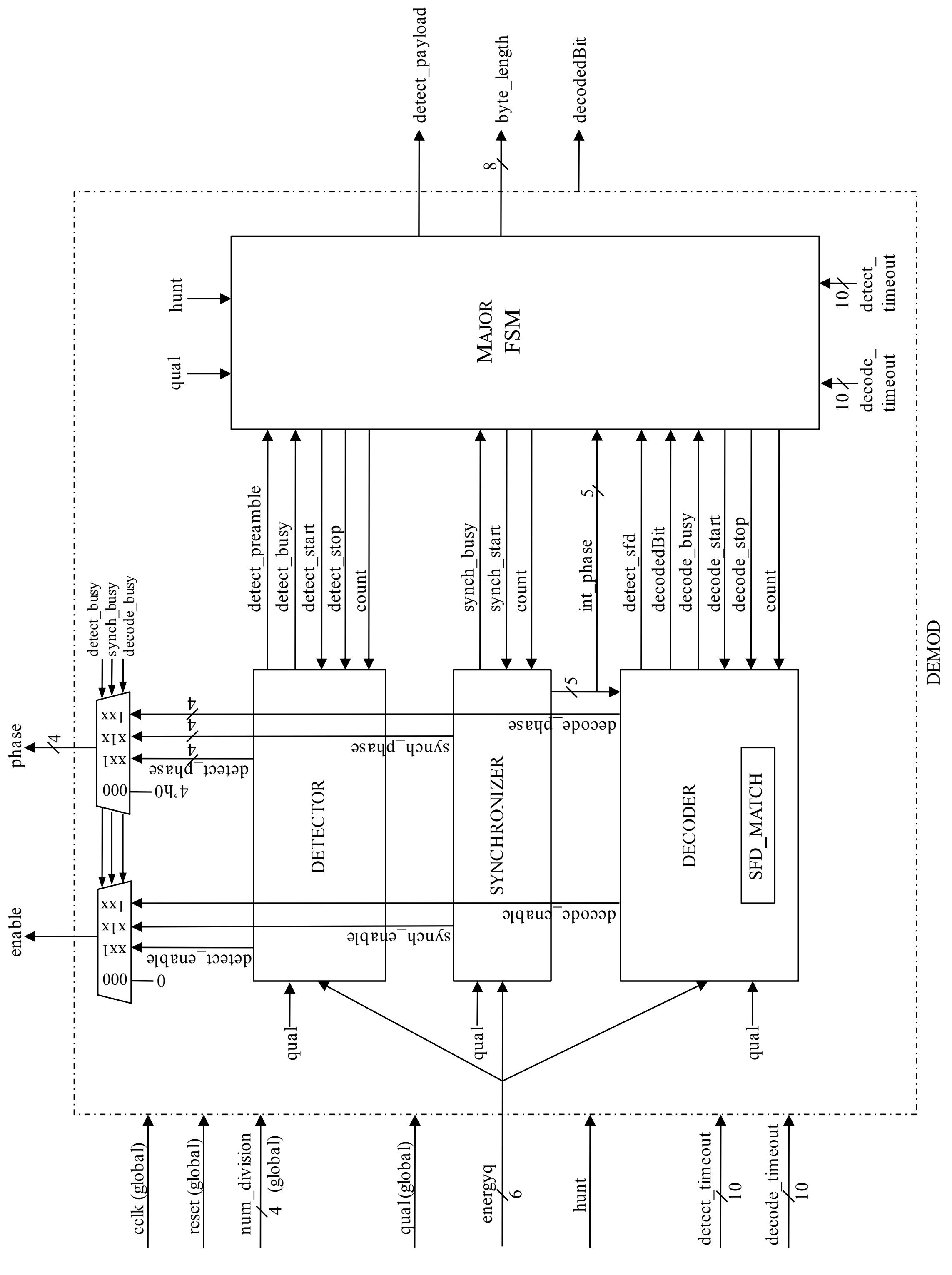


Figure 4-8: Demodulator block diagram.

The four main modules interact with one another so that the algorithms can be processed serially. The *detect_preamble* signal is sent from the detector to the major FSM in order to indicate whether the preamble signal is detected. Likewise, the *detect_sfd* signal is sent from the decoder to the major FSM to indicate whether the 11-bit Barker code in the received packet is detected. The synchronizer outputs the 5-bit *int_phase* signal, which specifies the correct phase of integration, to the decoder and the major FSM once the synchronization process is finished. The four lower bits of *int_phase* specify the phase number, and the most significant bit specifies whether we should start integrate in the first or the second half of the bit period. The *count* signal, which is generated by the major FSM, toggles between zero and one every chip clock period. *Count* indicates whether each chip period belongs to the first or the second half of the bit period, T_b.

The major-minor finite state machine abstraction method is utilized in the demodulator. The detection, synchronization, and decoding processes are subtasks of the demodulating process. The subtasks are encapsulated in "minor" finite state machines with common *reset* and *cclk*. The simple communication abstraction is used for the major FSM to control the operation of the minor FSMs. For example, *detect_start* tells the detector to begin the operation and *detect_busy*, which is an output from the detector, tells the major FSM whether the detection process is done. Another control signal from the major FSM, *detect_stop*, forces the detector to terminate its process. The major-minor finite state machine abstraction is good for a non-coherent ultra-wideband receiver because each minor process requires a variable period of time, which depends on the distortion level and the length of the packet.

There are three main sections for the Verilog code of each module: 1) defining states with parameter keyword, 2) defining a state register and some outputs in the sequential always block, and 3) specifying a next-state logic within a combinational always block. The state gets updated at every positive chip clock edge.

The two muxes select the *enable* and the *phase* signals from the detector, the synchronizer, and the decoder modules. If any of the three modules is in operation, then the *enable* and the *phase* signals from that particular module are selected by the muxes. If none of the three modules is in operation, then *enable* and *phase* are both zero. Since the three modules are serially processed, the two muxes always choose the correct signals from the only module that is in operation.

4.3.1 Detector

The detector determines whether the binary pulse position modulation (BPPM) preamble signal is received. The process is repeatedly executed unless the timeout limit is reached. The detection algorithm is based on energy collection, as explained in Chapter 2. For an AWGN channel, the detection algorithm selects one of the 8 integration phases (the detection phase space is $4T_c$) that maximizes the integral of the received signal energy. A "winner" is defined as a phase that has the maximum energy when the energy collection process covers 7 bits of all 0's. The process of choosing a "winner" is repeated 11 times. The receiver declares that it detects the preamble signal when any phase wins 6 times. The detection process is halted, and the synchronization process then begins. If there is no phase that wins at least 6 times, the receiver declares that it does not detect the preamble signal. The detection process is then repeated until the receiver detects the preamble signal. Since the detection process can be done with 6 winners, the number of preamble bits used by calling a detector once is at least ½ × 8 phases/winner × 7 bits/phase × 6 winners = 168 bits and at most ½ × 8 phases/winner × 7 bits/phase × 11 winners = 308 bits. The control flow of the detector is shown in Figure 4-9.

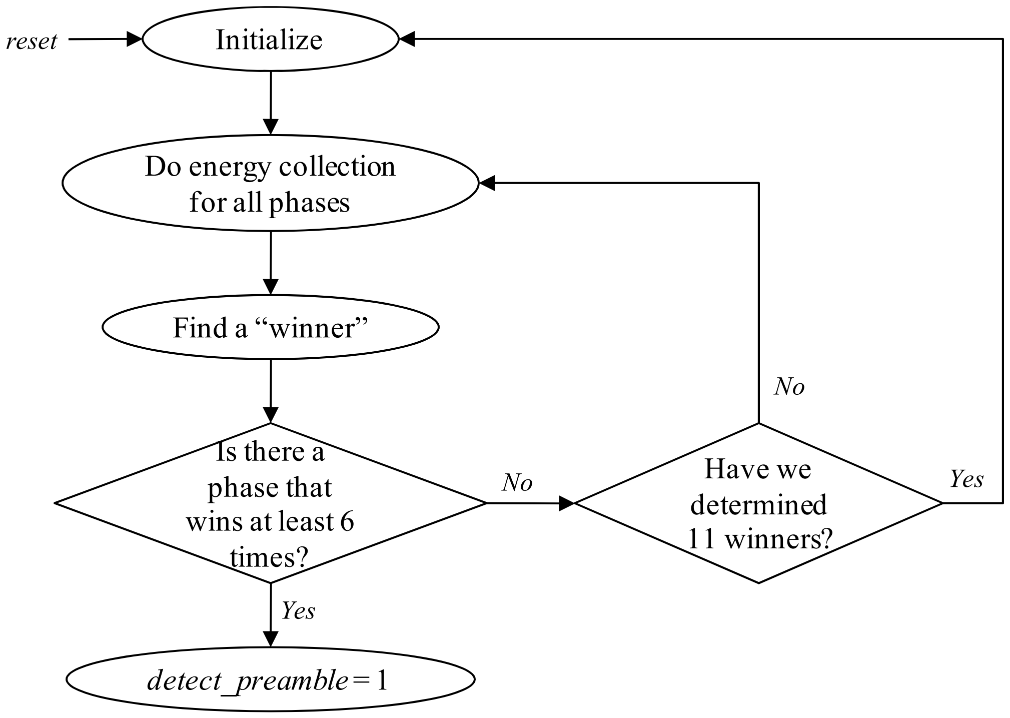

reset
Initialize
Do energy collection for all phases
Find a "winner"
Is there a phase that wins at least 6 times?
No
Have we determined 11 winners?
No
Yes
Yes
detect_preamble = 1

Figure 4-9: Control flow of the detector.

There are eight input signals going into the detector module: *cclk*, *reset*, *num_division*, *qual*, *energyq*, *detect_start*, *detect_stop*, and *count*. The module outputs four signals: a *detect_enable* and a 4-bit *detect_phase* signals to the two muxes; and *detect_preamble* and *detect_busy* to the major finite state machine. The most important output is *detect_preamble*, which is high when the detector finds a phase that "wins" at least 6 out of 11 times. This module implements a Mealy machine because the output signals depend on both the current state and the current inputs.

Internal registers are used as counters for various parameters. *CountZ* keeps track of the number of bits per phase, from which the detector should collect energy. *CountW* counts the number phase "winners" that the detector determines so far. *Winpoint[0]* to *winpoint[6]* hold the number of winning for all eight detection phases.

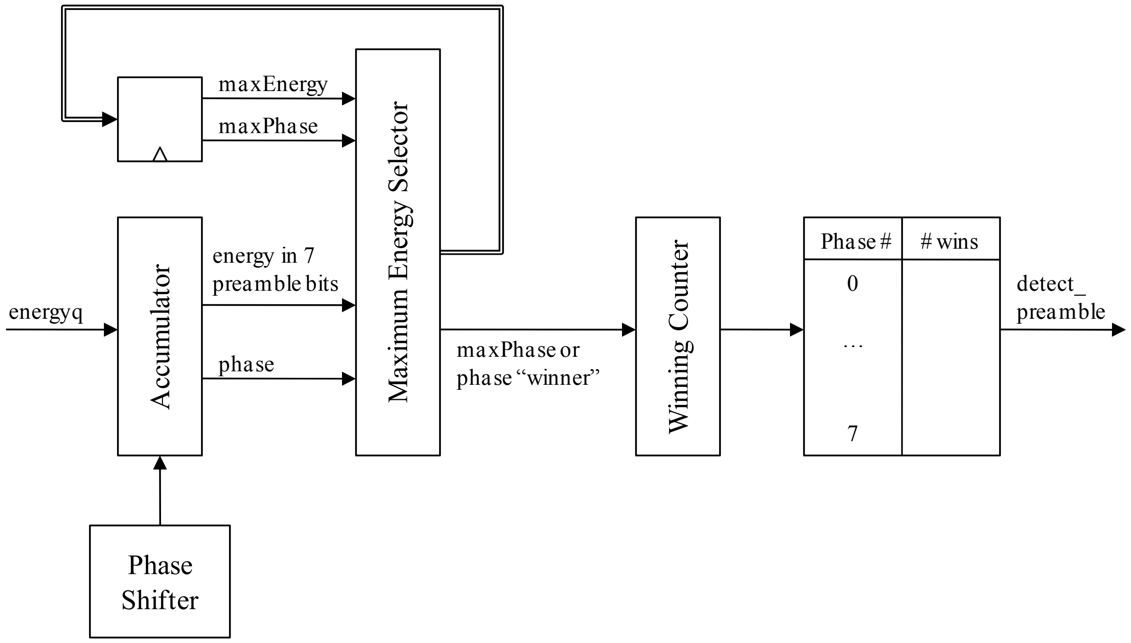


Figure 4-10: Detector block diagram.

Figure 4-10 shows the detector block diagram. The *energy* signal from the receiver's front-end is accumulated over 7 preamble bits. Next, the phase "winner" with maximum energy across all 8 phases is determined. The winning counter increments the number of wins for that phase. This process is repeated until one phase wins 6 times or until 11 winners are determined but no phase wins 6 times.

In the sequential always block, the state parameter gets updated at every positive edge of *cclk*. The detection process is implemented as a finite state machine with six states, as illustrated in Figure 4-11. If *reset* or *detect_stop* is asserted, the system is halted and is initialized in the IDLE state. In this state, all output signals are set to zero. The state machine transitions to the next state, COLLECT0, if the major FSM asserts *detect_start* and the integrator is not in operation. Another condition that *count* has to be zero is added so that the energy collection process begins at the first half of the bit period. Note that *detect_busy* is high in all states, except for IDLE, to tell the major FSM that the detector is in operation.

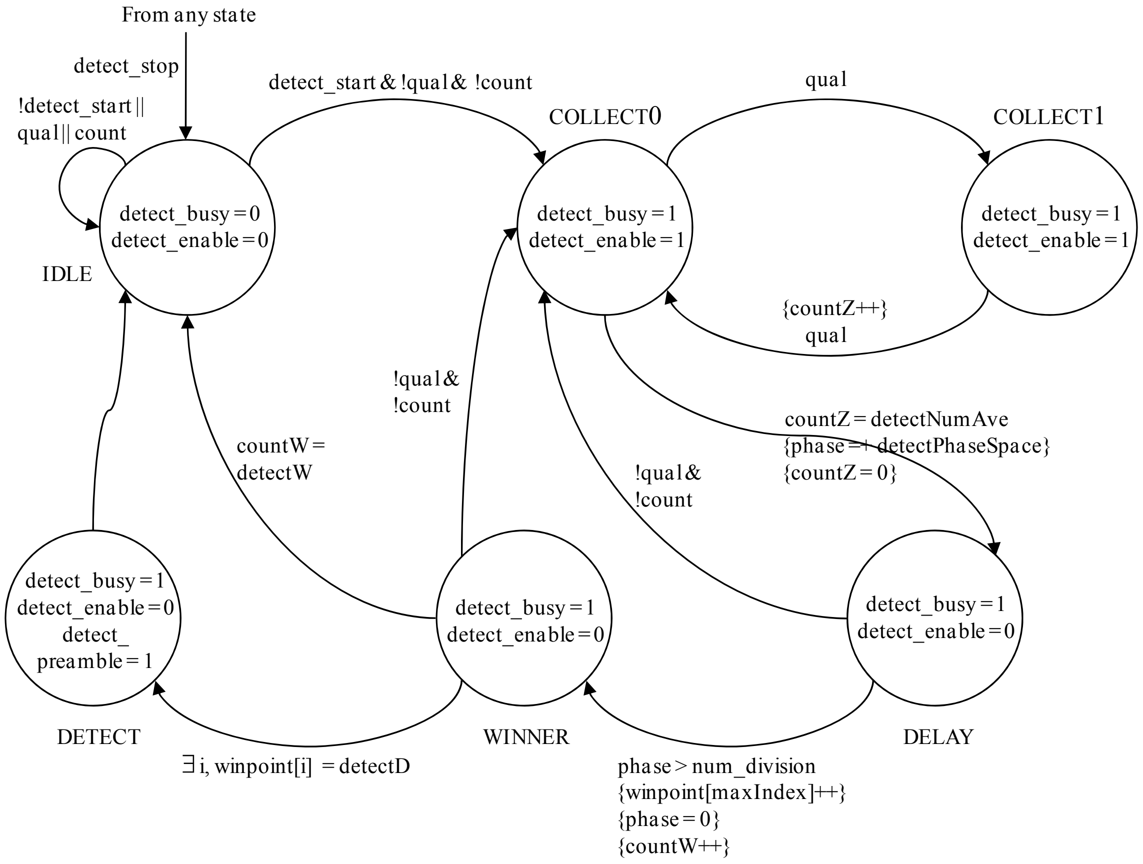


Figure 4-11: State transition diagram of the detector.

In COLLECT0 and COLLECT1, the energy in the first and the second half of the chip period, respectively, is accumulated for 7 times per phase. The *detect_enable* signal is asserted to start the integrator. For each phase value, the state machine toggles between the two states for 7 times. The detector then transitions to the DELAY state, where the phase that has maximum energy so far is determined and the phase of integration is incremented. The energy-collection process is repeated until the detector collects energy over all 8 phases. The state machine transitions to WINNER, where the first phase "winner" is identified and the winning point for that particular phase is incremented. Also, the counters and other internal registers are initialized before the energy-collection scheme is repeated all over again. If there is any phase that wins at least 6 times, then the *detect_preamble* goes high in the DETECT state. If 11 winners are determined but there is no phase that wins at least 6 times, then the detector declares that the preamble signal is not detected and the state machine loops back to the IDLE state.

4.3.2 Synchronizer

The synchronizer determines the phase of synchronization, where the energy of the binary pulse position modulation (BPPM) preamble signal is maximized. The process begins only if the preamble signal is detected. Similar to the synchronization algorithm, the detection scheme is based on energy collection, as explained in Chapter 2. For an AWGN channel, the synchronization algorithm selects one of the 32 integration phases (the synchronization phase space is T_c) that maximizes the integral of the received signal energy. The energy collection process for each phase covers 22 preamble bits of all 0's. The synchronizer processed only once before the decoding process begins. The number of preamble bits used by a synchronizer is ½ × 32 phases × 22 bit/phase = 352 bits. Therefore, the length of a detection process ranges from 48% to 88% of the length of a synchronization process. The control flow of the synchronizer is shown in Figure 4-12.

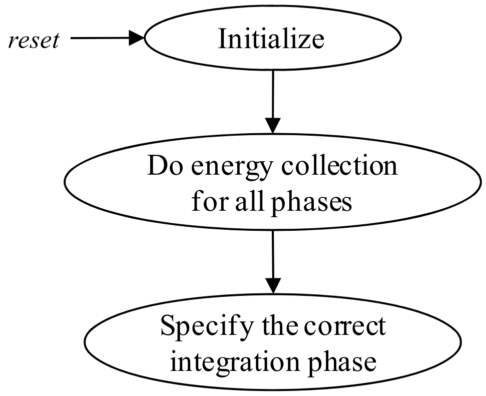

reset
Initialize
Do energy collection for all phases
Specify the correct integration phase

Figure 4-12: Control flow of the synchronizer.

There are seven input signals going into the detector module: *cclk*, *reset*, *num_division*, *qual*, *energyq*, *synch_start*, and *count*. The module outputs four signals: a *synch_enable* and a 4-bit *synch_phase* signals to the two muxes; *int_phase* to the decoder and the major finite state machine; and *synch_busy* to the major FSM. The most important output is the 5-bit *int_phase* signal, which returns the correct phase of integration for the decoding process. Similar to the detector, this module implements a Mealy machine because the output signals depend on both the current state and the current inputs. There is also an internal register, *countZ*, which keeps track of the number of bits per phase, from which the detector should collect energy.

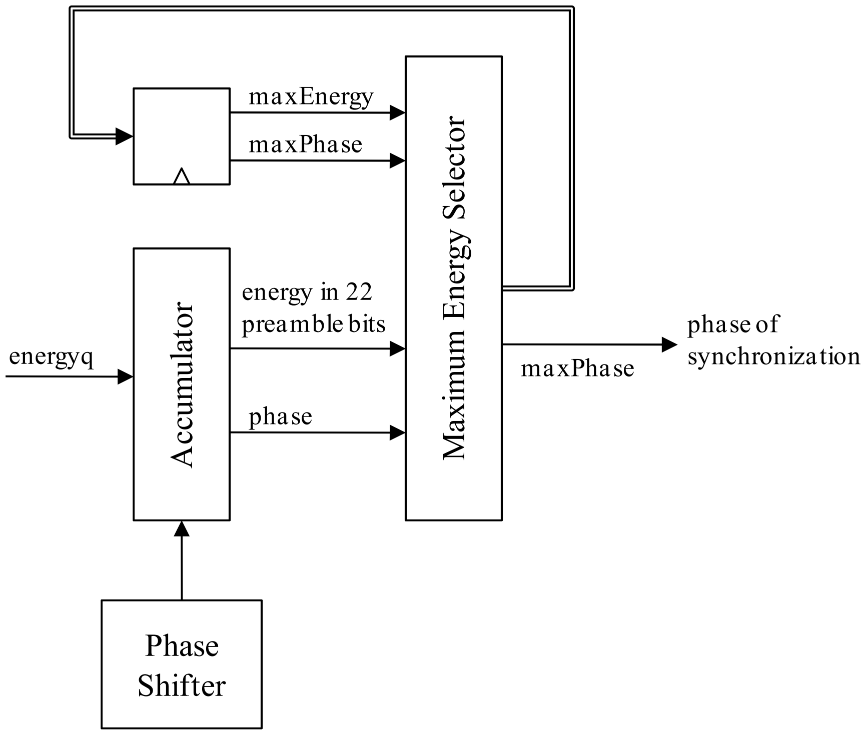


Figure 4-13: Synchronizer block diagram.

Figure 4-13 shows the synchronizer block diagram. The *energy* signal from the receiver's front-end is accumulated over 22 preamble bits. Next, the phase with maximum energy across all 32 phases is determined. This resulting phase is the correct phase of synchronization.

According to Figure 4-14, there are five states for the finite state machine that implements the synchronizer. On a reset, the system is halted and is initialized in the IDLE state. In this state, all output signals are set to zero. The state machine transitions to the next state, COLLECT0, if the major FSM asserts *synch_start* and the integrator is not in operation (i.e., *qual* is low). Also, *count* has to be low so that the energy collection process begins at the first half of the bit period. Note that *synch_busy* is high in all states, except for IDLE, to tell the major FSM that the synchronizer is in operation.

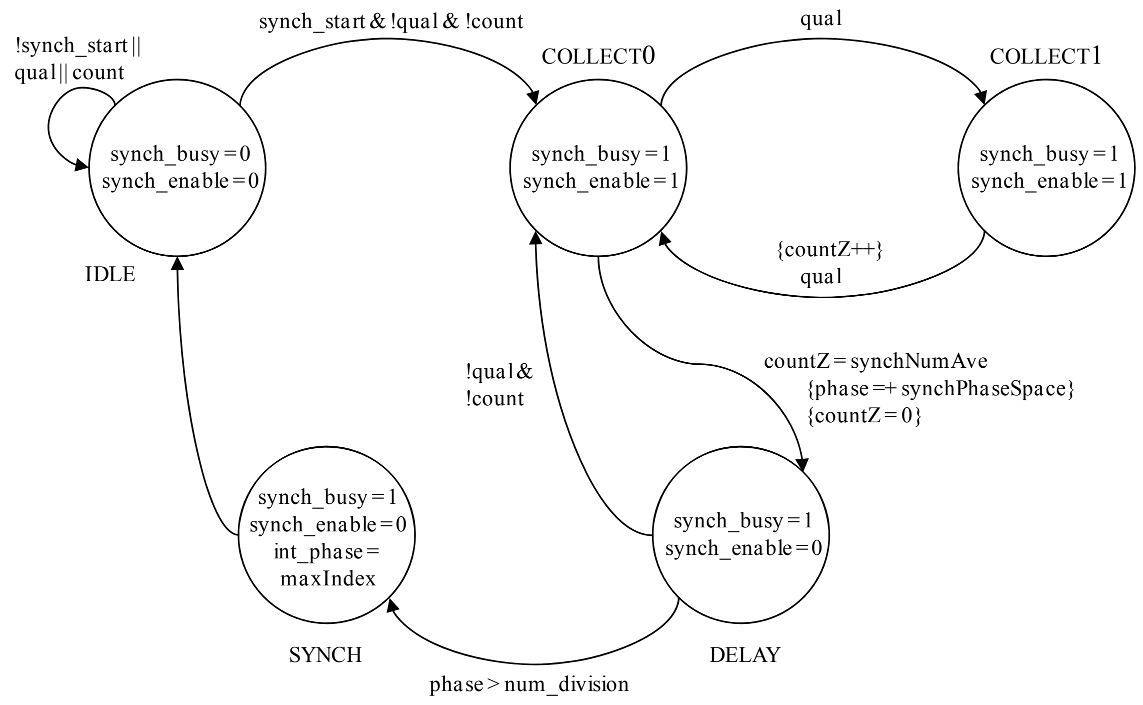


Figure 4-14: State transition diagram of the synchronizer.

In COLLECT0 and COLLECT1, the energy in the first and the second half of the chip period, respectively, is accumulated for 22 times per phase. That is, for each phase value, the state machine toggles between the two states for 22 times. The *synch_enable* signal is asserted to start the integrator. The synchronizer then transitions to the DELAY state, where the phase that has maximum energy so far is determined and the phase of integration is incremented. The energy-collection process is repeated until the synchronizer collects energy over all 32 phases. Note that two phases are done in each energy-collection loop. The state machine transitions to SYNCH when the phase of synchronization is determined and returned in the *int_phase* output signal. The state machine then loops back to the IDLE state.

4.3.3 Decoder

The decoder determines the received bits by energy collection. It compares the energy in the first half and the second half of the bit period with respect to the correct synchronization phase determined from the synchronizer. If the energy in the first half of the bit period is greater than that in the second half, then *decodedBit* is zero; otherwise, it is one. The detection process starts promptly after the synchronization process finishes. The decoder decodes preamble bits, an 11-bit SFD code, an 8-bit header that specifies the length in bytes of the payload, and payload bits sequentially. The decoding time thus depends on the length of the payload. If the SFD code is not found in the packet, the decoding process halts when the timeout limit is reached.

There are nine input signals going into the detector module: *cclk*, *reset*, *num_division*, *qual*, *energyq*, *int_phase*, *decode_start*, *decode_stop*, and *count*. The module outputs five signals: a *decode_enable* and a 4-bit *decode_phase* signals to the two muxes; *decode_busy*, *detect_sfd*, and *decodedBit* to the major FSM. *Detect_sfd* tells the major FSM that the 11-bit SFD code is detected in the received packet. The detector module implements a Mealy machine because the output signals depend on both the current state and the current inputs. There is also an internal register that holds the last eleven decoded bits in order to do the SFD matching algorithm.

The decoder module contains a sub-module, SFD_MATCH, as shown in Figure 4-8. This sub-module takes the last eleven decoded bits, along with the 11-bit pre-specified Barker SFD code, and outputs the sum of the results when the XNOR operator is executed on each corresponding bit of the two inputs. This sum indicates how many bits of the last eleven decoded bits match with the 11-bit Barker code. *Detect_sfd* goes high when the Barker code is found in the packet. It will stay high until the all the payload bits are decoded.

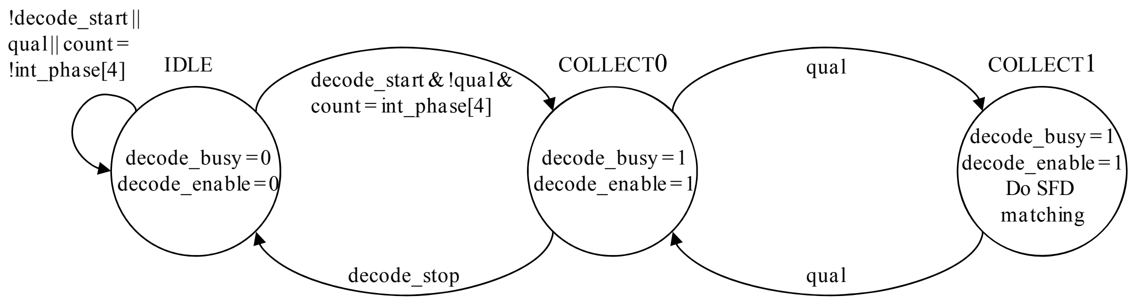


Figure 4-15: State transition diagram of the decoder.

The decoder is implemented as a small finite state machine with three states, as illustrated in Figure 4-15. If *reset* is asserted, the system is halted and is initialized in the IDLE state. In this state, all output signals are set to zero. The state machine transitions to the next state, COLLECT0, if the major FSM asserts *decode_start* and the integrator is not in operation (i.e., *qual* is low). Also, *count* has to be equal to the most significant bit of *int_phase* so that the energy collection process begins at the first half of the synchronized bit period. *Decode_busy* is high, when the decoder collects the energy from the received signal and produces bits, so that the major finite state machine recognized that the decoder is still in operation.

The energy in the first and the second half of the chip period is repeatedly accumulated in COLLECT0 and COLLECT1, respectively. The *decode_enable* signal is asserted to start the integrator. The state machine toggles between the two states when the *qual* signal from the integrator is high. *DecodedBit* is produced right after the energy in the second half of the bit period is collected. If the timeout limit is reached without finding the SFD code or all payload bits are decoded, then *decode_stop* signal is asserted by the major FSM and the decoder transitions back to the IDLE state.

4.3.4 Major Finite State Machine

The major finite state machine governs the operations of the detector, the synchronizer, and the decoder. As previously described in this chapter, the major-minor finite state machine abstraction is used as a method to control each operation. The three processes run sequentially. Therefore, the major FSM communicates to only one module at a time. Apart from controlling the overall operation of the demodulator, the major FSM keeps track of time spent in the detection and the decoding processes. If the timeout limit for the detector (*detect_timeout*) or the decoder (*decode_timeout*) is reached, then the demodulator halts and loops back to the initial state. Moreover, the major FSM extracts the header and the payload from *decodedBit,* which is determined by the decoder.

There are fourteen input signals going into the detector module: *cclk*, *reset*, *num_division*, *qual*, *hunt*, *int_phase*, *detect_preamble*, *detect_sfd*, *decodedBit*, *detect_busy*, *synch_busy*, *decode_busy*, *detect_timeout*, and *decode_timeout*. The module outputs eight signals: *detect_start* and *detect_stop* to the detector; *synch_start* to the synchronizer; *decode_start* and *decode_stop* to the decoder; *count* for the three modules above; and *detect_payload* and *byte_length* to an external user. The start/stop signals from the major FSM implement a major-minor finite state machine abstraction. The *count* signal, which toggles between zero and one every chip clock period, indicates whether each chip period belongs to the first or the second half of the bit period, T_b. *Detect_payload* is high when the *decodedBit* outputs payload bits. Moreover, the 8-bit *byte_length* output specifies the length in bytes of the payload.

Internal registers are used as counters for various parameters. *Timeout* counts time when the demodulator executes the detection or the decoding process. *CountBit* keeps track of the number of decoded payload bits. The 11-bit *bit_length* register holds the length in bits of the payload. Because one byte equals to eight bits, the eight most significant bits of *bit_length* are equal to *byte_length*; the tree least significant bits are all zeros. Each internal register gets updated at every positive edge of *cclk*.

The major FSM is implemented as a finite state machine with twelve states. Figure 4-16 shows the state transition diagram and some output assignments for the major finite state machine. If the *reset* signal is asserted, the process is halted and the state machine loops back to the IDLE state, where all parameters, except for the stop signals, are initialized to zero. The overall demodulation process starts when *hunt* is high. The major FSM starts each process serially. Three states are used for calling each module; for example, SYNCH0, SYNCH1, and SYNCH2 control the synchronizer. In the first state, the major FSM waits until the minor FSM is ready. Once the minor FSM is ready (the busy signal goes low), the major FSM transitions to the second state and triggers the minor module by asserting the start signal. If the minor module starts, the busy signal goes high and the major FSM transitions to the third state, where it waits until the minor FSM is done. That is, if the busy signal goes low, then the major FSM knows that the minor module completes the process and the next procedure can begin.

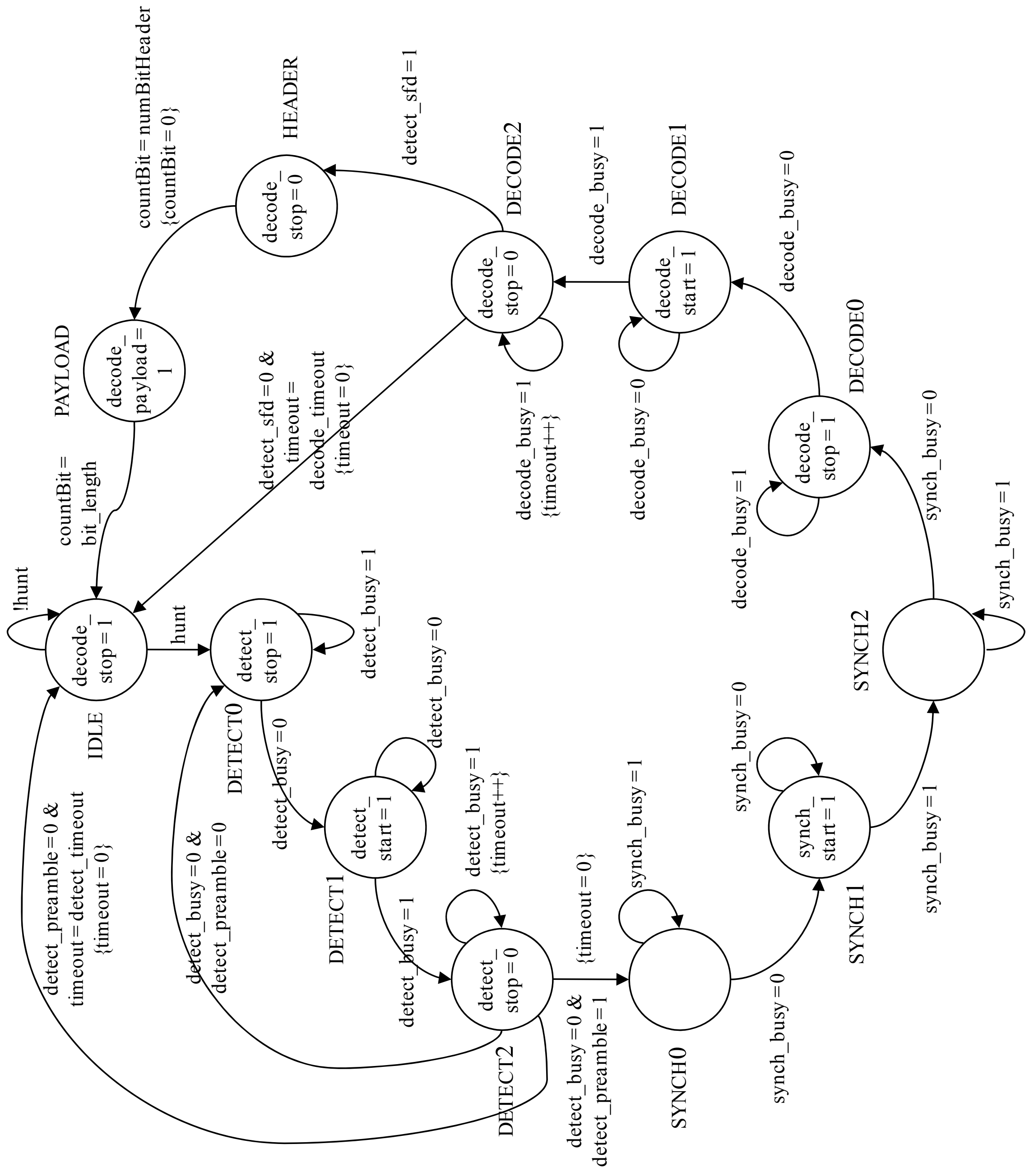


Figure 4-16: State transition diagram of the major finite state machine.

In the DETECT2 state, the major FSM waits for the detector to be done. If the demodulator is stuck during the detection process for too long without detecting the preamble signal, the major FSM goes to IDLE. Otherwise, if the preamble is not detected and the detector already finished the process, then the state machine loops back to DETECT0 in order to restart the detection process. If the detector is working, then *timeout* is incremented. The major FSM transitions to SYNCH0 to start the synchronization process, if the preamble signal is detected and the detector finishes its process. The decoder unconditionally starts after the synchronizer is done. In the DECODE2 state, if *detect_sfd* goes high, the state machine transitions to header in order to extract the header bits from *decodedBit*. If the timeout limit for decoding is reached, then the demodulator stops decoding and goes to the initial state. Again, if the decoder is processing, then *timeout* is incremented and the major FSM stays in the same state.

After the 11-bit SFD code is found in the received signal, the eight next decoded bits are stored in *byte_length* to specify the length of the payload. If *countBit* is eight, then the major FSM transitions to the next state, PAYLOAD. In this state, *decode_payload* is set to one to indicate that the payload bits are valid. If the demodulator decodes all payload bits (i.e., *countBit* equals to *bit_length*), then the overall demodulating process is finished and the major FSM loops back to the IDLE state. Otherwise, the FSM stays in the PAYLOAD state and counts how many bits the decoder produces so far.

4.4 Summary

This chapter explained the digital baseband architecture of a non-coherent ultra-wideband receiver. The Verilog implementation of the receiver is mapped to hardware. The embedded system design technique for realizing the digital baseband system, which implements many of the components of the system within a field programmable gate array (FPGA), is used to build a receiver. Chapter 5 describes the FPGA implementation, the system set-up for testing and debugging, and the performance of a non-coherent ultra-wideband receiver.

Chapter 5

Receiver Implementation in FPGA

5.1 FPGA Testing System

The receiver digital system is implemented in an FPGA. The total number of gates for the design is 6,673. The UWB receiver system is connected to a Tektronix Logic Analyzer machine in order to verify the implementation and measure the performance. A Tektronix Logic Analyzer consists of a pattern generator and a logic analyzer. A pattern generator generates output signals, which can be pre-specified in an input text file. On the other hand, a logic analyzer measures incoming signals and reports them in an output text file.

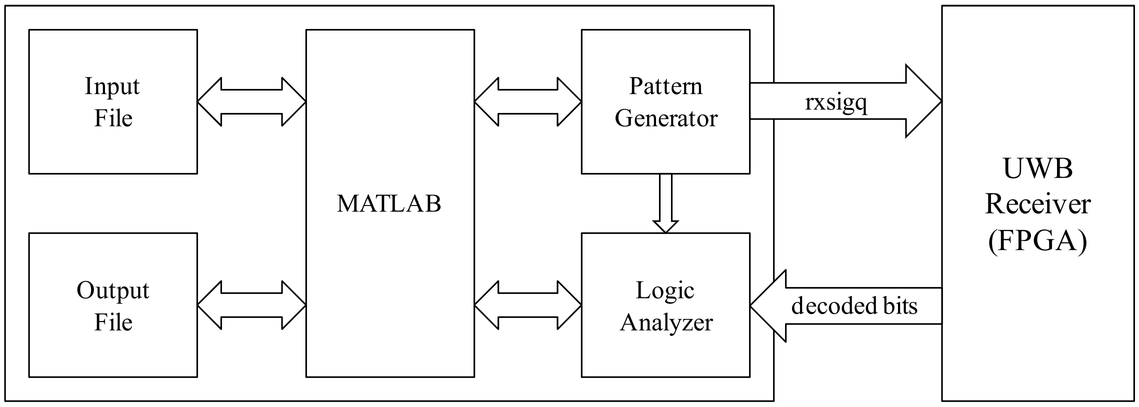


Figure 5-1: Test set-up for the receiver digital system.

The FPGA testing system for the UWB receiver is shown in Figure 5-1. The wirelessly received signal is modeled in MATLAB for different channel conditions (i.e., different signal-to-noise ratios) and is saved in text files. The received signal is modeled by adding the Gaussian noise, which is randomly generated in MATLAB for a particular SNR level, to the oversampled transmitted signal. The received signal, *rxsig*, is normalized such that the average power of the normalized signal, *rxsig**, is equal to 0.5. Specifically, *rxsig* is multiplied by a constant factor:

$$rxsig^* = \alpha \cdot rxsig, \quad \alpha = \sqrt{\frac{N}{2\sum_{i=1}^{N} rxsig_i^2}},$$

where N is the length of *rxsig* vector. The normalized signal is saturated and quantized to 8 bits. On average, the digitized signal lies in the middle of the 8-bit range.

In a Tektronix Logic Analyzer machine, MATLAB reads an input text file and starts the pattern generator. The pattern generator then outputs four main signals to the FPGA. The four signals are *reset*, *reset_rx*, *hunt*, and *rxsiqsq*, as shown in Figure 4.7. *Rxsigsq* is the square of the quantized and normalized received signals. The logic analyzer measures the output signals from the UWB receiver at pre-specified times and saves the results in a text file. The three outputs of the UWB receiver system are *decodedBit*, *byte_length*, and *detect_payload*, which is the qualifier of the first two signals. Figure 5-2 shows a logic analyzer screenshot when the signal packet is transmitted in a 3 dB SNR channel. The *detect_payload* signal goes high right after the header bits to indicate that the payload bits and the length outputs are both valid.

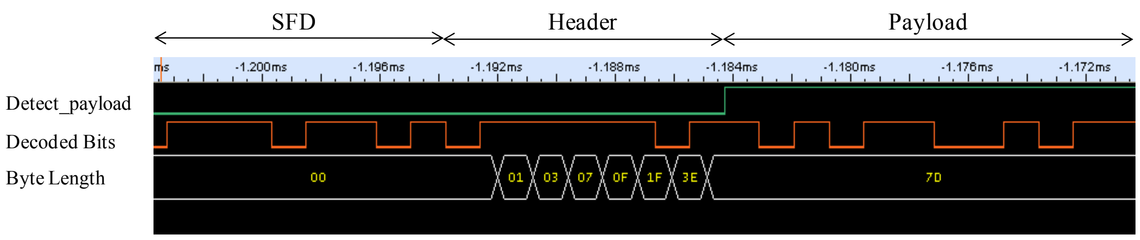


Figure 5-2: Logic analyzer screenshot.

5.2 Performance

The digital baseband implementation of the UWB receiver system in Verilog is fully tested. The performance of the digital system for detection, synchronization, and decoding processes is consistent with the performance measured from MATLAB simulation. Figure 5-3 presents the detection performance in MATLAB and Verilog. Note that the probability of transmitting a signal is assumed to be 0.5. The probability of detection increases, as the signal-to-noise ratio increases. Verilog shows a slightly worse probability of detection because we lose accuracy of data from quantizing the received signal and its energy. The maximum time for detection is 19.71 μs. Moreover, the UWB detector performs reasonably well in noisy channels. At 0 dB SNR, the missed detection rate is 2.1% and the false alarm rate is 1.2%. The data for receiver's performance in MATLAB and Verilog can be found in the appendix (section 7.5).

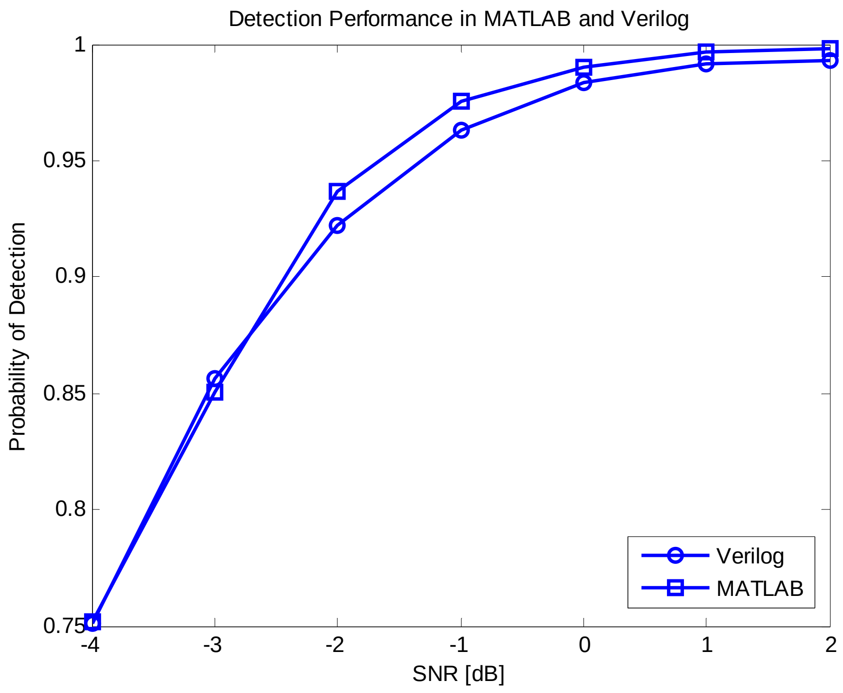


Figure 5-3: Detection performance.

The synchronization performance in MATLAB and Verilog is illustrated in Figure 5-4. The probability of synchronization increases with the signal-to-noise ratio because at high SNR, the energy received is approximately equal to the energy transmitted. Since the modeled signal is oversampled in MATLAB, the error within $\pm 2T_c$ should be less than the error within $\pm 2T_c$ but greater than the error within $\pm T_c$ measured from Verilog. The time to synchronize is 22.53 μs, which is longer than the detection time. At 0 dB SNR, the synchronization error rate (within $\pm 2T_c$) is 0.5%.

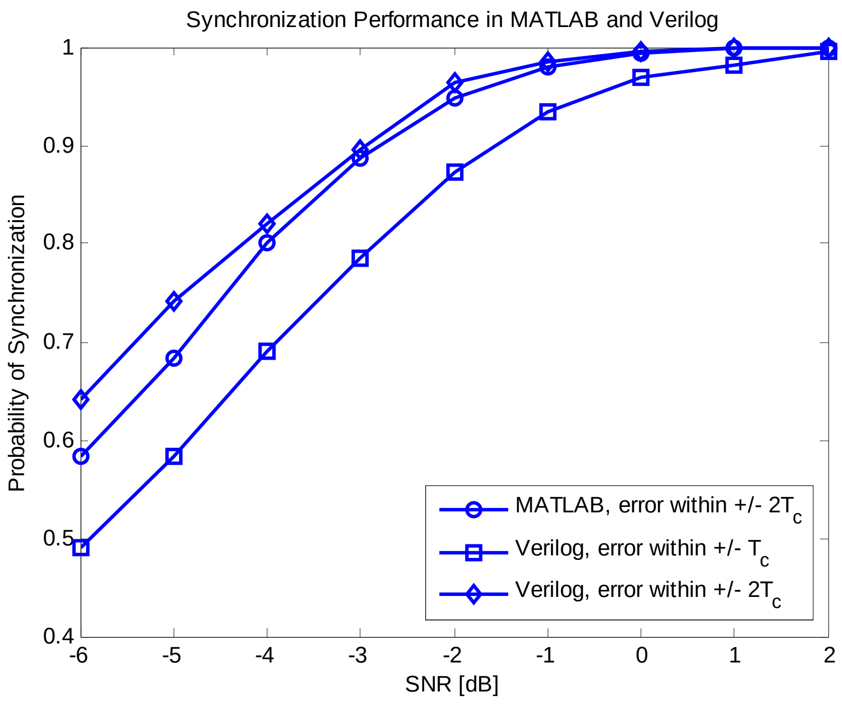


Figure 5-4: Synchronization performance.

Lastly, Figure 5-5 presents the decoding performance of the digital system comparing to MATLAB simulation results. The bit error rate (BER) increases, as the signal-to-noise ratio decreases. The performance measured in Verilog is similar to the performance obtained from MATLAB simulation. Because of digital bit quantization, the receiver system implemented in Verilog achieves slightly higher bit error rate. At 0 dB SNR, the bit error rate is 8.6%, but it drops sharply to 0.1% at an SNR of 5 dB.

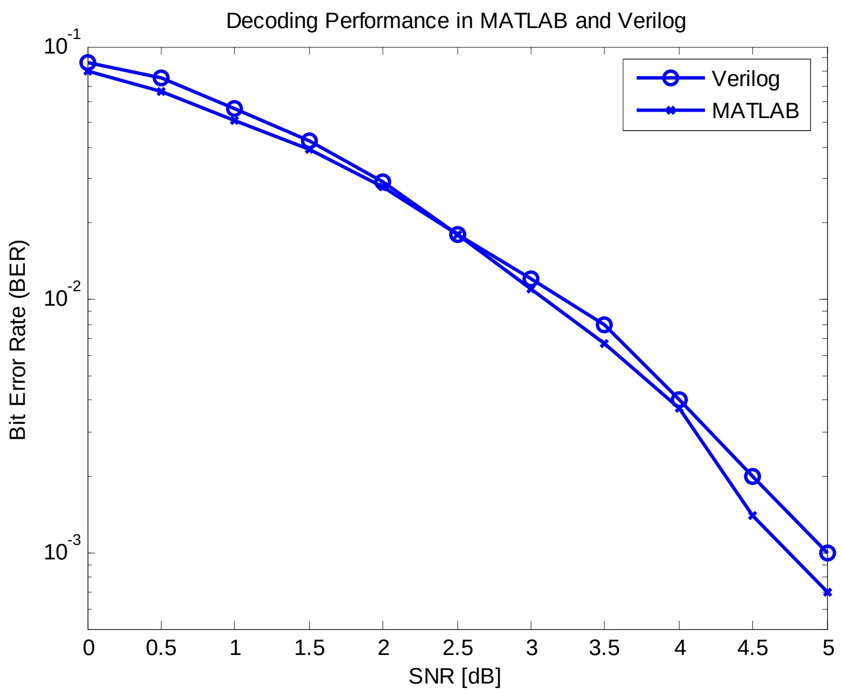


Figure 5-5: Decoding performance.

Chapter 6

Conclusion

Ultra-wideband (UWB) communication is an emerging technique for wireless transmission in the 3.1-10.6 GHz unlicensed band with signal bandwidths of 500 MHz or greater. This technology is capable of transmitting more data in a given time than other existing technologies. Therefore, UWB technology can be used in various wireless personal area network (WPAN) applications. A non-coherent receiver based on energy collection achieves low complexity, low cost, and low power consumption. Because many ultra-wideband technology applications require efficient energy consumption, the non-coherent method is used to implement a receiver.

This thesis presents the algorithms and the digital implementation for a non-coherent ultra-wideband receiver. The detection, synchronization, and decoding algorithms are implemented in MATLAB and mapped to hardware (FPGA). Crucial parameters for each of the three algorithms are determined from MATLAB simulations, so that the receiver performs efficiently in noisy channels and the digital implementation maintains low complexity.

For the digital system implementation of the UWB receiver system, the major-minor finite state machine abstraction is used since each process requires a variable period of time. The demodulator module controls the energy-collection process for each algorithm. In the final system, a radio frequency and an analog front-end will interface with the FPGA, resulting in a complete radio receiver.

The UWB receiver performs effectively in noisy channels. At the signal-to-noise ratio (SNR) of 0 dB, the receiver achieves a missed detection rate of 2.1% and a false alarm rate of 1.2%. The synchronization error (within ±2 chip periods) rate is 0.5%. The bit error rate is 8.6%, but it drops sharply to 0.1% at an SNR of 5 dB. Moreover, the detection and the synchronization processes take 19.72 μs and 22.53 μs, respectively.

Possible future research would be to implement more sophisticated detection and synchronization algorithms in order to improve performance in noisy channels. However, the ultra-wideband receiver should maintain low power consumption and quick processing time.

Chapter 7

Appendix

7.1 MATLAB Code for Receiver System Algorithm Simulation

Below is the hierarchy of the overall MATLAB simulation system.

- uwbSim.m
  - makeParam.m
  - makePreamble.m
  - makeSym.m

    Initializing parameters and signal

  - tx.m
  - channel.m

    Signal transmission and channel modeling

  - detection.m
  - synchNonCoherent.m
  - rxNonCoherent.m

    Detection, synchronization, and decoding algorithm

uwbSim.m

```
%
% Simulates an UWB system
%

clear all
close all

rand('state',0);
randn('state',0);

% System level parameters
Nbits = 1e2;    % Number of bits to transmit
SNRdB = 10.0;
Npkt = 1e2;     % Number of pkts to be simulated

% Modeling parameters
Fs = 1/(0.4e-9);

%%%%%% No programmable parameters below %%%%%%

fprintf('\n========================================= \n');
% Initialize parameter structures
makeParam;

tic

error = 0;
missed = 0;
totBerr = 0;

for pkt = 1:Npkt
    % Tx Modem: Bits -> modulated Tx packet
    dataBits = randint(Nbits,1);
    txSig = tx(dataBits, txParam);

    % Channel impairments
    rand('state',pkt);
    [rxSig, chParam] = channel(txSig, chParam);

    % Rx Modem: Rx signal -> bits
    detect = 0;
    i = 0;
    while detect==0 & i<1
        [detect,synchStart] = detection(rxSig, txParam, rxParam,
4*txParam.NsC, 7, 11, 6);
        rxSig = rxSig(synchStart:end);
        i = i+1;
    end

    if (detect==0)
        fprintf('Pkt:%d\tSignal not detected%d\t ',pkt);
        fprintf('\n');
        missed = missed + 1;
    else

    intStart = synchNonCoherent(rxSig, txParam, rxParam, txParam.NsC,
22);
    rxBits = rxNonCoherent(rxSig(intStart:end), txParam, rxParam);

    L = length(rxBits);
    j = 1;
    while (j+length(txParam.sfd)-1 <= L) &
(sum(xor(txParam.sfd',rxBits(j:j+length(txParam.sfd)-1))) > 0) % SFD
detection threshold
        j = j+1;
    end

    hdrStart = j+length(txParam.sfd);
    if (hdrStart > L-txParam.hdrBits)
        fprintf('Pkt:%d\tSFD not detected%d\t ',pkt);
        fprintf('\n');
    else
        bitlength = sum(rxBits(hdrStart:hdrStart+7).*[2^10 2^9 2^8 2^7
2^6 2^5 2^4 2^3]);
    end
    bitStart = hdrStart+txParam.hdrBits;
    rxDataBits = rxBits(bitStart:end);

    %% Record statistics

    % Synchronizer error
    if (chParam.truncSamp >= 160)
        synchErr = intStart + mod(synchStart-1,160) -
(2*length(preamble.code)*txParam.NsC - chParam.truncSamp); % +
mod(synchStart,160)
    else
        synchErr = intStart + mod(synchStart-1,160) -
(length(preamble.code)*txParam.NsC - chParam.truncSamp);
    end

    if (synchErr < -80 | synchErr > 160)
        synchErr = mod(synchErr,160);
    elseif (synchErr > 80)
        synchErr = -mod(-synchErr,160);
    end

    if (abs(synchErr)>2*txParam.NsC)
        error = error + 1;
    end

    % Bit error
    if (hdrStart > L-txParam.hdrBits)
        Berr = Nbits;
    else
        Berr = sum(xor(rxDataBits(1:Nbits)', dataBits));
    end

    if (Berr > 2)
        totBerr = totBerr + 1;
    end

    fprintf('Pkt:%d\tSynchError:%d\tBitError:%d\t ',pkt,synchErr,Berr);
    fprintf('\n');

    end
end % for pkt = 1:Npkt

fprintf('NumDetectMissed:%d\tProbDetectMissed(percent):%d\t
',missed,100*missed/Npkt);
fprintf('\n');
fprintf('NumSynchError:%d\tProbSynchError(percent):%d\t
',error,100*error/Npkt);
fprintf('\n');
fprintf('NumBitError:%d\tProbBitError(percent):%d\t
',totBerr,100*totBerr/Npkt);
fprintf('\n');

toc
```

makeParam.m

```
% ~mymatlab/modem/makeParam
%
% Initializes transmit, channel, and receive parameters
%
%

% Hop parameters
% Can be removed by defining 802.15.4a style scrambler.
hopStateInit    = 93758;
signStateInit   = 8247;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Transmit Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

txParam.Fs      = Fs;       %
txParam.Tc      = 2e-9;     % Chip period

txParam.NsC     = round(txParam.Tc*Fs);    % NsC = # of samples / chip

% Pulse parameters
pulse.type      = 0;        % 0 = Gaussian
pulse.width     = 1.4e-9;   %
[pulse.xt, BW10dB] = uwbPulse(pulse.type, pulse.width, Fs);

txParam.pulse   = pulse;



% Preamble parameters
preamble.code   =  [ones(1,16) zeros(1,16)]';
preamble.rep    = 128;   % # of code repetitions
preamble.sfd    = [1]; % Outer code for preamble code
preamble.NTc    = 1;   % Preamble chip in multiples of Tc

txParam.preamble = preamble;

% Header parameters
txParam.hdrBits = 8;

% UWB Symbol parameters
% Used to construct the header/payload
sym.maxSyms     = 2048;     % Max syms/pkt.
sym.Nb          = 32;

txParam.sym     = sym;
txParam.sfd     = [0 0 0 1 1 1 0 1 1 0 1]';
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Channel Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

chParam.Fs          = Fs;

% Impairment ON/OFF options
chParam.addAWGN     = 1;        % 1: Yes, 0: No
chParam.trunc       = 1;        % Randomly truncate packet
chParam.addDly      = 0;        % Add random delay before (truncated)
packet

% Impairment parameters
chParam.SNRdB       = SNRdB;    % Signal to noise ratio
chParam.BW10dB      = BW10dB;   % 10dB BW of signal of interest (one-
sided)
chParam.maxTruncNs  = 32;       % Truncation range
chParam.truncSamp   = 0;        % Samples truncated(set in channel.m)
chParam.maxDlyNs    = 1e3;      % Delay range
chParam.dlySamp     = 0;        % Samples delayed (set in channel.m)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Receive Parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Receiver block ON/OFF options
rxParam.synchBypass = 0;        % Bypass synchronizer
rxParam.hdrBypass   = 0;        % Bypass header decoding

% Parameters used when blocks are bypassed
rxParam.pktStart    = ...       % Expected start of pkt.
    (preamble.rep + length(preamble.sfd))*length(preamble.code)...
     *preamble.NTc*txParam.NsC + 1;

rxParam.pktLenByte  = 13;       % Pkt. length when header bypassed

% Synchronization parameters
synch.threshSFDInit = 84;

rxParam.synch = synch;

% Receiver state initialization (for scramblers etc.)
rxParam.hopState    = hopStateInit;
rxParam.signState   = signStateInit;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Assertions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Include all assumptions about parameters here.
% Integral number of samples in a chip period
tempNsC = txParam.Tc*Fs;
if tempNsC - round(tempNsC)
    warning('\n\n*** Warning: Non-integer # of samples in a chip
***\n\n');
end

% Start of SFD
if preamble.sfd(1) == 0
    fprintf(...
        '\n\n*** SFD Code: Synch might work incorrectly. ***\n\n');
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% Report important parameters
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fprintf('\n10dB BW of pulse (one-sided): %.3fMHz\n\n',BW10dB/1e6);

% Impairments
if ~chParam.addAWGN
    fprintf('\n*** Warning: No receiver noise added ***\n\n');
end

% Receiver bypass
if rxParam.synchBypass
    fprintf('\n*** Warning: Synchronizer bypassed ***\n\n');
end

if rxParam.hdrBypass
    fprintf('\n*** Warning: Header decoding bypassed ***\n\n');
end
```

makePreamble.m

```
function ySym = makePreamble(preamble)

%%MAKEPREAMBLE Generate a preamble.
%%
%% ySym = makePreamble(preamble, Fs)
%%
%% Inputs
%% ------
%% 1. preamble  : Structure specifying preamble parameters.
%%
%% Outputs
%% -------
%% 1. y         : Unmodulated preamble symbols.
%%
%% Notes
%%
%% i. Fields of the preamble structure.
%     i.1. code     : Preamble's PN code
%     i.2. rep      : # of times preamble is repeated
%     i.3. sfd      : Start of frame delimiter
%     i.4. Tc       : Preamble chip period

% Form preamble code
sync = repmat(preamble.code,preamble.rep,1);

% Make SFD
sfd = conv(preamble.code, ...
               upsample(preamble.sfd, length(preamble.code)));
sfd = sfd(1:end-(length(preamble.code)-1));

% Append and separate pulses with right preamble spacing
ySym = upsample([sync; sfd], preamble.NTc);
```

makeSym.m

```
function ySym = makeSym(bits)

%%MAKESYM Generates UWB symbols.
%%
%% ySym = makeSym(bits, symParam)
%%
%% Inputs
%% ------
%% 1. bits      : Data bits to be transmitted.
%%
%% 2. symParam  : Relevant UWB parameters.
%%
%% Outputs
%% -------
%% 1. ySym      : 2-PPM symbols.
%%
%% Notes
%%

yBits = [];
zero_code = [ones(1,16) zeros(1,16)];
one_code  = [zeros(1,16) ones(1,16)];

for i=1:length(bits)
    if (bits(i) == 0)
        yBits = [yBits zero_code];
    else
        yBits = [yBits one_code];
    end
end

ySym = upsample(yBits, 1);

ySym = ySym';

return;
```

tx.m

```
function y = tx(bits, txParam)

%% TX Generate a UWB packet.
%%
%% txSig = tx(txParam)
%%
%% Inputs
%% ------
%% 1) txParam   : Structure specifying parameters like
%%                packet length, preamble type etc.
%%                (see makeParam.m for details)
%%
%% 2) bits     : Vector of data bits to be transmitted.
%%
%% Outputs
%% -------
%% 1) txSig     : Packet sampled at spg6hnu8by7reecified rate.
%%
%% Notes
%%
%% i. Scramblers do not retain state across function boundaries.
%%

txDebug = 1;

% Global parameters
Fs              = txParam.Fs;    % Sampling frequency (Hz)
Tc              = txParam.Tc;    % Chip period

% Pulse, preamble, payload parameters
pulse           = txParam.pulse;
preambleParam   = txParam.preamble;
symParam        = txParam.sym;

%%%%%%%%%%%

% Make preamble
preSym = makePreamble(preambleParam);

% % Make payload
% paySym = makePayload(bits, payloadParam);

% Pad data bits to make complete bytes
N = length(bits);
bits = [bits; zeros(mod(N,8),1)];
if bits > symParam.maxSyms
    error('Payload exceeds capacity');
end

sfdSym = makeSym(txParam.sfd);
```

```
% Make header (8-bit field with pkt. length in bytes)
hdrSym = makeSym(dec2bin(length(bits)/8, 8) - '0');

% Make payload
paySym = makeSym(bits);

% Make packet
packetSym = [preSym; sfdSym; hdrSym; paySym];

% Modulate
NSampC = round(Tc*Fs);    % Number of samples in a chip period
y = conv(pulse.xt, upsample(packetSym, NSampC));
```

channel.m

```
function [y, chParamOut] = channel(x, chParam)

%% CHANNEL Add impairments to packet.
%%
%% [y, chParam] = channel(x, chParam)
%%
%% Inputs
%% ------
%% 1) x         : Modulated packet.
%%
%% 2) chParam   : Structure specifying impairments.
%%                (see makeParam.m for fields and explanation)
%% Outputs
%% -------
%% 1) y         : Packet with impairments.
%%
%% 2) chParamOut: chParam updated with actual values of impairments.
%%

Fs = chParam.Fs;

%% Impairment: Packet truncation
% Models receiver waking up in the middle of a packet
% or AGC chewing part of the packet etc.
if chParam.trunc
    chParam.truncSamp = floor(rand*chParam.maxTruncNs*2e-9*Fs)+13;
    yTrunc = x(chParam.truncSamp+1:end);
else
    yTrunc = x;
end

%% Impairment: Random, unknown delay
if chParam.addDly
    chParam.dlySamp = floor(rand*chParam.maxDlyNs*1e-9*Fs);
    yDly = [zeros(chParam.dlySamp,1); yTrunc];
else
    yDly = yTrunc;
end

%% Impairment: AWGN
% Always adds unit variance noise.
if chParam.addAWGN
    xPwr = mean(x.^2);
    SNR = 10^(chParam.SNRdB/10);
    pwrFactor = SNR/xPwr;
    oSamplingFactor = (Fs/2)/chParam.BW10dB;
    yNoise = yDly + sqrt(oSamplingFactor/pwrFactor)*randn(size(yDly));
else
    yNoise = yDly;
end
y = yNoise;
chParamOut = chParam;
```

detection.m

```
function [detect,synchStart] = detection(rxSig, txParam, rxParam,
phaseSpace, numAve, windowSize, numDetect)

%% DETECTION Non-Coherent detector.
%%
%% synchStart = detection(rxSig, txParam, rxParam, phaseSpace, numAve,
windowSize, numDetect)
%%
%% Inputs
%% ------
%% 1) rxSig         : Received packet sampled at specified rate.
%%
%% 2) tx/rxParam    : Structure specifying tx/rx parameters
%%
%% 3) phaseSpace    : Space between each integration phase
%%
%% 4) numAve        : Number of averaging periods per phase
%%
%% 5) windowSize    : Window size (number of results)
%%
%% 6) numDetect     : Number of windows that one phase has maximum
%%                    energy so that the detection declares that the
%%                    preamble signal is detected
%%
%% Outputs
%% -------
%% 1) synchStart    : Preamble start index
%%
%% 2) detect        : 1 if the preamble signal is detected; 0 otherwise
%%

% Extract relevant fields
Tc          = txParam.Tc;
Fs          = txParam.Fs;
pulse       = txParam.pulse;
preamble    = txParam.preamble;
sfdCode     = preamble.sfd;

NSampC = round(preamble.NTc*Tc*Fs);
NSampB = length(preamble.code)*NSampC;
Nint = round(NSampB/phaseSpace); %% Number of Integrations

k = 0;
F = 0;
maxArray = [];

while (F<numDetect & k<windowSize)
    R = zeros(1,Nint)';
    for j=1:numAve
        for i=1:Nint
            ts = 1 + k*NSampB*numAve + (i-1)*phaseSpace;
            R(i) = R(i) + sum(rxSig(ts+(j-1)*NSampB:ts+NSampB/2+(j-1)
*NSampB).^2);
        end
    end
    [maxValue,maxIndex] = max(R);

    maxArray = [maxArray maxIndex];
    [M,F] = mode(maxArray);
    k = k+1;
end

k = k-1;

if (F>=numDetect)
    synchStart = 1 + numAve*windowSize*NSampB + (M-1)*phaseSpace;
    detect = 1;
else
    synchStart = 1 + numAve*windowSize*NSampB;
    detect = 0;
end

return;
```

synchNonCoherent.m

```
function intStart = synchNonCoherent(rxSig, txParam, rxParam,
phaseSpace, numAve)

%% SYNCHNONCOHERENT Non-Coherent synchronizer.
%%
%% intStart = synchNonCoherent(rxSig, txParam, rxParam, phaseSpace,
numAve)
%%
%% Inputs
%% ------
%% 1) rxSig         : Received packet sampled at specified rate.
%%
%% 2) tx/rxParam    : Structure specifying tx/rx parameters
%%
%% 3) phaseSpace    : Space between each integration phase
%%
%% 4) numAve        : Number of averaging periods per phase
%%
%% Outputs
%% -------
%% 1) intStart      : Payload start index
%%                    ([] returned if no preamble found)

start = 1;
% Extract relevant fields
Tc = txParam.Tc;
Fs = txParam.Fs;
pulse       = txParam.pulse;
preamble    = txParam.preamble;
sfdCode     = preamble.sfd;
NSampC = round(preamble.NTc*Tc*Fs);
NSampB = length(preamble.code)*NSampC;
Nint = round(NSampB/phaseSpace); %% Number of Integrations

R = zeros(1,Nint)';
for j=1:numAve
    for i=1:Nint
        ts = start + (i-1)*phaseSpace;
        R(i) = R(i) + sum(rxSig(ts+(j-1)*NSampB:ts+NSampB/2+(j-1)
*NSampB).^2);
    end
end
[maxValue,maxIndex] = max(R);

if maxIndex ~= 1
    intStart = start + (maxIndex-1)*phaseSpace - 2*NSampC;
else
    intStart = start + NSampB - 2*NSampC;
end
if (intStart <= 0)
    intStart = intStart + NSampB;
end
```

```
return;
```

rxNonCoherent.m

```
function rxBits = rxNonCoherent(rxSig, txParam, rxParam)

%% RXNONCOHERENT Non-Coherent receiver
%%
%% rxBits = rxNonCoherent(rxSig, txParam, rxParam)
%%
%% Inputs
%% ------
%% 1) rxSig               : Received packet sampled at specified rate.
%%
%% 2) txParam, rxParam    : Structures specifying Tx, Rx parameters
%%
%% Outputs
%% -------
%% 1) rxBits              : Decoded bits (= [] if no packet found).
%%

L  = length(rxSig);
Tc = txParam.Tc;
Fs = txParam.Fs;
pulse  = txParam.pulse;

NSampC = round(Tc*Fs);
NSampB = txParam.sym.Nb*NSampC;

k = 0;
ts = 0;
rxBits = [];

while ts < (L - 2*NSampB)
    ts = 1 + k*NSampB;
    if (sum(rxSig(ts:ts+NSampB/2).^2) >
sum(rxSig(ts+NSampB/2+1:ts+NSampB).^2))
        rxBits = [rxBits 0];
    else
        rxBits = [rxBits 1];
    end
    k = k+1;
end

return;
```

7.2 MATLAB Simulation Results for Synchronization Algorithm

Table A1: Synchronization Simulation with *phaseSpace* = T_c and error within $\pm 2T_c$

numAve	% Synch Error	*numAve*	% Synch Error
1	55.6	17	1.0
2	41.8	18	0.7
3	30.3	19	0.8
4	21.4	20	0.8
5	17.4	21	0.6
6	14.0	**22**	**0.4**
7	10.5	23	0.4
8	8.4	24	0.3
9	7.3	25	0.2
10	6.3	26	0.3
11	4.1	27	0.2
12	3.9	28	0.1
13	3.1	29	0.1
14	2.7	30	0.1
15	1.7	31	0.1
16	1.3	32	0.0

Table A2: Synchronization Simulation with *phaseSpace* = $2T_c$ and error within $\pm 2T_c$

numAve	% Synch Error
23	1.4
24	1.0
25	1.0
26	1.2
27	1.3
28	0.7
29	**0.5**
30	0.3
31	0.3
32	0.2

Table A3: Synchronization Simulation with
phaseSpace = $3T_c$ and error within $\pm 2T_c$

numAve	% Synch Error
31	0.8
32	0.8
33	0.8
34	0.8
35	0.6
36	0.7
37	0.6
38	0.7
39	**0.5**
40	0.3
41	0.3

Table A4: Synchronization Simulation with
phaseSpace = $4T_c$ and error within $\pm 2T_c$

numAve	% Synch Error
30	7.1
40	5.8
50	5.5
60	4.7
70	3.7
80	3.4
90	3.3
100	3.0
110	2.7
120	2.6

Table A5: Synchronization Simulation with

phaseSpace = $4T_c$ and error within $\pm 3T_c$

numAve	% Synch Error
10	2.6
12	1.5
13	1.3
14	1.1
15	0.8
16	0.6
17	**0.5**
18	0.4
20	0.5
30	0.1

Table A6: Synchronization Simulation with

phaseSpace = T_c, error within $\pm 2T_c$, and *numAve* = 22

SNR (dB)	% Synch Error
-6.0	41.7
-5.5	36.0
-5.0	31.7
-4.5	25.7
-4.0	19.9
-3.5	15.4
-3.0	11.3
-2.5	7.8
-2.0	5.1
-1.5	3.1
-1.0	2.1
-0.5	1.2
0.0	0.7
0.5	0.4
1.0	0.1

Table A7: Synchronization Simulation with

phaseSpace = $2T_c$, error within $\pm 2T_c$, and *numAve* = 25

SNR (dB)	% Synch Error
-6.0	36.8
-5.5	31.9
-5.0	27.1
-4.5	21.1
-4.0	16.1
-3.5	12.9
-3.0	8.8
-2.5	7.0
-2.0	4.8
-1.5	2.8
-1.0	1.6
-0.5	0.9
0.0	0.5
0.5	0.2
1.0	0.2

7.3 MATLAB Simulation Results for Detection Algorithm

Table A8: Detection Simulation with

phaseSpace = $4T_c$, *windowSize* = 11, and *numAve* = 4

numDetect	% Missed	% False Alarm	% Total Error*
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	84.4	42.20
4	0.0	31.7	15.85
5	1.2	6.1	**3.65**
6	11.2	0.5	5.85
7	28.9	0.0	14.45
8	50.9	0.0	25.45
9	68.7	0.0	34.35
10	85.4	0.0	42.70
11	96.4	0.0	48.20

* Assume equal probability of transmitting and not transmitting the preamble signal
Note: Number in bold indicates the minimum total detection error.

Table A9: Detection Simulation with

phaseSpace = $4T_c$, *windowSize* = 11, and *numAve* = 5

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	86.8	43.40
4	0.0	29.8	14.90
5	0.5	5.6	**3.05**
6	7.2	0.9	4.05
7	22.1	0.0	11.05
8	38.7	0.0	19.35
9	58.7	0.0	29.35
10	76.8	0.0	38.40
11	93.2	0.0	46.60

Table A10: Detection Simulation with

phaseSpace = $4T_c$, *windowSize* = 11, and *numAve* = 6

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	84.2	42.10
4	0.0	27.7	13.85
5	0.0	5.3	2.65
6	2.7	1.1	**1.90**
7	15.6	0.3	7.95
8	32.1	0.0	16.05
9	50.1	0.0	25.05
10	69.4	0.0	34.70
11	88.7	0.0	44.35

Table A11: Detection Simulation with

phaseSpace = $4T_c$, *windowSize* = 11, and *numAve* = 7

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	86.0	43.00
4	0.0	26.0	13.00
5	0.0	6.3	3.15
6	2.1	0.4	**1.25**
7	14.4	0.0	7.20
8	26.6	0.0	13.30
9	43.0	0.0	21.50
10	60.4	0.0	30.20
11	83.9	0.0	41.95

Table A12: Detection Simulation with

phaseSpace = $6T_c$, *windowSize* = 11, and *numAve* = 4

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	71.6	35.80
5	0.0	28.7	14.35
6	3.1	5.6	**4.35**
7	15.1	0.9	8.00
8	30.2	0.0	16.10
9	44.6	0.0	22.30
10	67.6	0.0	33.80
11	86.8	0.0	43.40

Table A13: Detection Simulation with

phaseSpace = $6T_c$, *windowSize* = 11, and *numAve* = 5

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	72.0	36.00
5	0.0	26.4	13.20
6	1.9	6.6	**4.25**
7	12.1	0.6	6.35
8	25.6	0.0	12.80
9	40.2	0.0	20.10
10	56.6	0.0	28.30
11	77.8	0.0	38.90

Table A14: Detection Simulation with

phaseSpace = $6T_c$, *windowSize* = 11, and *numAve* = 6

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	70.5	35.25
5	0.0	24.2	12.10
6	0.7	5.9	**3.30**
7	10.1	0.9	5.50
8	21.5	0.2	10.85
9	33.6	0.0	16.80
10	48.9	0.0	24.45
11	70.8	0.0	35.40

Table A15: Detection Simulation with

phaseSpace = $6T_c$, *windowSize* = 11, and *numAve* = 7

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	70.8	35.40
5	0.0	25.5	12.75
6	0.4	6.0	**3.20**
7	10.7	1.0	5.85
8	21.2	0.0	10.60
9	31.0	0.0	15.50
10	44.5	0.0	22.25
11	66.7	0.0	33.35

Table A16: Detection Simulation with

phaseSpace = $8T_c$, *windowSize* = 11, and *numAve* = 4

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	90.4	45.20
5	0.0	42.5	21.25
6	0.9	12.0	**6.45**
7	13.6	2.2	7.90
8	26.6	0.3	13.45
9	41.4	0.1	20.75
10	61.4	0.0	30.70
11	81.5	0.0	40.75

Table A17: Detection Simulation with

phaseSpace = $8T_c$, *windowSize* = 11, and *numAve* = 5

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	91.3	45.65
5	0.0	45.7	22.85
6	0.5	14.4	**7.45**
7	11.8	3.1	**7.45**
8	22.7	0.4	11.55
9	35.9	0.0	17.95
10	52.6	0.0	26.30
11	71.7	0.0	35.85

Table A18: Detection Simulation with

phaseSpace = $8T_c$, *windowSize* = 11, and *numAve* = 6

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	92.3	46.15
5	0.0	45.8	22.90
6	0.4	13.5	**6.95**
7	12.0	3.5	7.75
8	21.4	1.2	11.30
9	31.6	0.3	15.95
10	44.8	0.1	22.45
11	66.1	0.0	33.05

Table A19: Detection Simulation with

phaseSpace = $8T_c$, *windowSize* = 11, and *numAve* = 7

numDetect	% Missed	% False Alarm	% Total Error
1	0.0	100.0	50.00
2	0.0	100.0	50.00
3	0.0	100.0	50.00
4	0.0	80.4	40.20
5	0.0	45.8	22.90
6	0.2	13.2	6.70
7	10.1	2.4	**6.25**
8	19.7	0.3	10.00
9	28.6	0.0	14.30
10	40.7	0.0	20.35
11	60.7	0.0	30.35

Table A20: Detection Simulation with

phaseSpace = $4T_c$, *numAve* = 7, *windowSize* = 11, and *numDetect* = 6

SNR (dB)	% Missed	% False Alarm	% Total Error
-6.0	84.8	0.7	42.75
-5.5	78.6	0.7	39.65
-5.0	71.7	0.7	36.20
-4.5	64.0	0.7	32.35
-4.0	54.7	0.7	27.70
-3.5	42.6	0.7	21.65
-3.0	30.8	0.7	15.75
-2.5	21.7	0.7	11.20
-2.0	14.2	0.7	7.45
-1.5	8.0	0.7	4.35
-1.0	4.9	0.7	2.80
-0.5	2.6	0.7	1.65
0.0	1.7	0.7	1.20
0.5	0.8	0.7	0.75
1.0	0.5	0.7	0.60
1.5	0.2	0.7	0.45
2.0	0.1	0.7	0.40

7.4 MATLAB Simulation Results for Decoding Algorithm

Table A21: Decoding Simulation

SNR (dB)	% Bit Error Rate (BER)
0.0	5.2847
0.5	5.0669
1.0	4.1108
1.5	3.3382
2.0	2.5036
2.5	1.7526
3.0	1.1462
3.5	0.6813
4.0	0.3807
4.5	0.1888
5.0	0.0862
5.5	0.0323
6.0	0.0122
6.5	0.0036
7.0	0.0015

7.5 Receiver’s Performance in MATLAB and Verilog

Table A22: Detection Performance in MATLAB and Verilog

SNR (dB)	% Missed	% False Alarm	% Total Error*
Verilog			
-4.0	48.6	1.2	24.90
-3.0	27.5	1.2	14.35
-2.0	14.3	1.2	7.75
-1.0	6.2	1.2	3.70
0.0	2.1	1.2	1.65
1.0	0.5	1.2	0.85
2.0	0.2	1.2	0.70
MATLAB			
-4.0	49.3	0.4	24.85
-3.0	29.5	0.4	14.95
-2.0	12.2	0.4	6.30
-1.0	4.5	0.4	2.45
0.0	1.6	0.4	1.00
1.0	0.3	0.4	0.35
2.0	0.0	0.4	0.20

* Assume equal probability of transmitting and not transmitting the preamble signal

Table A23: Synchronization Performance in MATLAB and Verilog

SNR (dB)	% Synchronization Error		
	MATLAB (error within $\pm 2T_c$)	Verilog (error within $\pm T_c$)	Verilog (error within $\pm 2T_c$)
-6.0	41.7	50.9	35.9
-5.0	31.7	41.7	25.8
-4.0	19.9	31.0	18.0
-3.0	11.3	21.5	10.5
-2.0	5.1	12.7	3.6
-1.0	2.1	6.6	1.5
0.0	0.7	3.0	0.5
1.0	0.1	1.8	0.1
2.0	0.0	0.5	0.0

Table A24: Decoding Performance in MATLAB and Verilog

SNR (dB)	% Bit Error Rate (BER)	
	Verilog	MATLAB
0.0	8.6	8.0
0.5	7.5	6.6
1.0	5.7	5.1
1.5	4.2	3.9
2.0	2.9	2.8
2.5	1.8	1.8
3.0	1.2	1.1
3.5	0.8	0.67
4.0	0.4	0.37
4.5	0.2	0.14
5.0	0.1	0.07

References

[1] Hirt, W. “Ultra-Wideband Radio Technology: Overview and Future Research,” *Computer Communications* 26 (2003): 46-52.

[2] Yang, L. and Giannakis, G. “Ultra-Wideband Communication: An idea whose time has come,” *IEEE Signal Processing Magazine* (2004): 26-54.

[3] Rabbachin, A.; Tesi, R.; and Oppermann, I. “Bit Error Rate Analysis for UWB Systems with a Low Complexity, Non-Coherent Energy Collection Receiver,” *Proceedings of International Symposium on Telecommunications* (2004), Lyon, France.

[4] Rabbachin, A. and Oppermann, I. “Synchronization Analysis for UWB Systems with a Low-Complexity Energy Collection Receiver,” *Ultra Wideband Systems* (2004): 288-292.

[5] Stoica, L.; Repo, H; Tiuraniemi, T.; and Oppermann, I. “An Ultra Wideband Low Complexity Circuit Transceiver Architecture for Sensor Networks,” *Proceedings of IEEE Circuits and Systems Conference* (2005), Kobe, Japan.

[6] Durisi, G. and Benedetto, S. “Comparison between Coherent and Noncoherent Receivers for UWB Communications,” *Journal on Applied Signal Processing* 3 (2005): 359-368.

[7] Barrett, T. “History of Ultrawideband (UWB) Radar and Communications: Pioneers and Innovators,” *Proceedings of Progress in Electromagnetics Symposium* (2000), Cambridge, MA.

[8] Dickson, D. and Jett, P. “An Application Specific Integrated Circuit Implementation of a Multiple Correlator for UWB Radio Applications,” *Proceedings of IEEE Military Communications Conference* (1999): 1207-1210.

[9] Lee, H.J.; Ha, D.S.; and Lee, H.S. “Toward Digital UWB Radios: Part 1 – Frequency Domain UWB Receiver with 1-bit ADCs,” *Ultra Wideband Systems* (2004): 248-252.

[10] Wentzloff, D. and Chandrakasan, A. “A 47pJ/pulse 3.1-to-5GHz All-Digital UWB Transmitter in 90nm CMOS,” *IEEE International Solid-State Circuits Conference Digest of Technical Papers* 50 (2007): 118-119.

[11] Lee, F. and Chandrakasan, A. “A 2.5nJ/b 0.65V 3-to-5GHz Subbanded UWB Receiver in 90nm CMOS,” *IEEE International Solid-State Circuits Conference Digest of Technical Papers* 50 (2007): 116-117.

[12] Stoica, L.; Rabbachin, A.; Repo, H.; Tiuraniemi, T.; and Oppermann, I. “An Ultrawideband System Architecture for Tag Based Wireless Sensor Networks,” *IEEE Transactions on Vehicular Technology* 54-5 (2005): 1631-3645.

[13] Carbonelli, C. and Mengali, U. “Low Complexity Synchronization for UWB Noncoherent Receivers,” *IEEE 61st Vehicular Technology Conference* 2 (2005).

[14] He, N. and Tepedelenlioglu, C. "Performance Analysis of Non-Coherent UWB Receivers at Different Synchronization Levels," *IEEE Global Telecommunications Conference* 6 (2004): 3517-3521.

[15] Maravic, I.; Kusuma, J.; and Vetterli, M. "Low-Sampling Rate UWB Channel Characterization and Synchronization," *Journal of Communications and Networks KOR, special issue on ultra-wideband systems* 5-4 (2003): 319-327.

[16] Azakkour, A.; Regis, M.; Pourchet, F.; and Alquie, G. "Challenges for a New Integrated Ultra-Wideband (UWB) Source," *Proceedings of IEEE Ultra Wideband Systems and Technologies Conference* (2003): 433-437.

[17] *IEEE 802.15.4a Draft Standard* (version D4).