University of Alberta

An Energy-efficient, Wide-band Asynchronous Transceiver for Wireless Sensor Networks

by

Malihe Ahmadi Najafabadi

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

Master of Science

in

Communications

Department of Electrical and Computer Engineering

©Malihe Ahmadi Najafabadi
Spring 2012
Edmonton, Alberta

To my beloved parents, Akram and Morteza for their endless support and love

Abstract

Medium access in applications of wireless sensor networks is often uncoordinated while sensor nodes communicate bursty flows of data. Therefore random-access packet-based communication schemes are suitable for such networks. Preamble detection is an important task in packet-based communication protocols. The implementation of a previously proposed preamble detection scheme for low-power, wide-band, asynchronous packet communications is proposed that has built-in characterization features. A digital baseband design for the transmitter part of this scheme was fabricated on IBM Corporation's 130-nm digital CMOS process. Silicon prototypes of the fabricated design were successfully tested.

The receiver design of the packet-based communications was prototyped on a Xilinx FPGA. The main goal of this thesis was to first measure the performance of the preamble detector at hardware speed. The second goal was to optimize the detector to reduce its area and power consumption. The optimized preamble detection design was also implemented on a Xilinx FPGA. Test measurements showed that its performance closely follows the non-optimized preamble detection design with half the area and power usage. The presented optimized preamble detection design can be utilized in low-power, high-data-rate applications of wireless sensor networks.

Acknowledgement

I would like to express my sincere gratitude to Dr. Christian Schlegel and Dr. Bruce Cockburn for their invaluable time and support and advice. Working under their supervision has been an honor for me.

I am grateful to University of Alberta, Electrical and Computer Engineering Department's faculty member for making my time in school intriguing and memorable.

I wish to record my gratitude to my beloved parents for their continuous support. Special thank goes to my colleagues at HCDC and VLSI lab. and University of Alberta Electrical Engineering Department's staff.

Contents

List of Tables

List of Figures

List of Acronyms

ADC	Analog-to-Digital Converter
API	Application Programmer's Interface
AWGN	Additive White Gaussian Noise
BCI	Brain Computer Interface
BER	Bit Error Rate
BPSK	Binary Phase Shift Keying
cSNR	Chip Signal-to-Noise Ratio
CMOS	Complementary Metal Oxide Semiconductor
DAC	Digital-to-Analog Converter
DBPSK	Differential Binary Phase Shift Keying
DCM	Digital Clock Manager
DRC	Design Rule Check
DSSS	Digital Spread Spectrum Sequence
DV	Design Vision
DR	Data Register
FE	First Encounter
FIR	Finite Impulse Response
FP	Front Panel
FPGA	Field-Programmable Gate Array
GDS	Graphical Design System
GUI	Graphical User Interface
HDL	Hardware Description Language
IR	Instruction Register
JTAG	Joint Test Action Group
LEF	Layout Exchange Format
LNA	Low Noise Amplifier
LSB	Least Significant Bit
LUT	Look-Up Table
MAC	Medium Access Control
MSB	Most Significant Bit
OOK	On-Off Keying
PA	Power Amplifier
PLH	Physical Layer Header
PLL	Phase-Locked Loop
pSNR	Preamble Signal-to-Noise Ratio
P&R	Place and Route
RC	Resistive Capacitive
RF	Radio Frequency

RRC	Root Raised-Cosine
RTL	Register-Transfer Level
RX	Receiver
SDC	Synopsys Design Constraints
SDF	Standard Delay Format
SPEF	Standard Parasitic Exchange Format
SPICE	Simulation Program For Integrated Circuit Emulation
SR	Shift Register
sSNR	Symbol Signal-to-Noise Ratio
TAP	Test Access Port
TX	Transmitter
UWB	Ultra-Wide Band
WSN	Wireless Sensor Networks

Chapter 1

Introduction

1.1 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) have lately attracted the attention of numerous researchers because of their widespread potential application to medical monitoring, to environmental sensing, to military surveillance, and in many other areas. A WSN consists of spatially-distributed nodes that communicate over radio links. A sensor node or mote is a node in a wireless sensor network that is capable of gathering information from its surroundings (from motion to temperature to electrochemical signals produced by neurons in implanted devices), performing data processing, and then communicating results with other nodes in the network using a radio transmitter/receiver.

Several medium access control (MAC) protocols have been proposed for WSNs. Since the control of a WSN is decentralized (with no global controller), random-access communication protocols provide the most flexibility and are suitable matches. These protocols require nodes to listen to the channel at all times so that the receiver can detect packets which are sent at arbitrary times. WSNs should also be able to deal with potential interference caused by the concurrent transmission of multiple nodes within the same radio pass-band. Random access protocols are often based on the original ALOHA protocol [5] [6].

Among the challenges introduced by applications of WSN, operating within the constraints of a finite battery lifetime is one of the most important ones which requires low power design for communications. Decentralized communication protocols are slightly less complex since they don't require synchronization at both communication ends, and are thus more power friendly [7] [8]. In most of the published WSN applications, the nodes have a bursty flow of data, therefore random-access, packet-based communications protocols are appropriate for WSNs. Since a random-access protocol requires the receiver to constantly monitor the channel, a key requirement in its design is to minimize the power consumption of the packet detection algorithm.

1.2 Motivations and Significance of the Project

A wireless sensor network is a collection of sensor nodes that are randomly and spatially distributed in the environment. Sensors are capable of wireless communications and transmit the sensed and possibly processed data via the wireless medium. Network nodes can be mobile, stationary, or a combination of the two.

Mote sensors can be used in self-organizing wireless networks for such innovative applications as medical and health monitoring, environment and habitat monitoring, industrial process screening, security perimeter monitoring, and agricultural applications like greenhouse monitoring. The key significance of sensor networks is that they can be implemented anywhere with no existing communication infrastructure. They can form the essential building blocks for an ad-hoc wireless network. Communication protocols based on random channel access are appropriate candidates for these types of networks since there is no central coordinator to synchronize their communication. They can be used in environments where pre-deploying communications infrastructure is hard or impossible (e.g., military applications) and also in cases where a previously existing infrastructure is no longer available (e.g., disaster recovery applications).

A medical sensor network is able to wirelessly monitor vital signs of the patient and to control drug delivery, e.g. insulin pumps, thus providing more effective, adjustable, economical, prompt treatment without sacrificing the patient's comfort and mobility [9] [10]. Medical mote sensors have the potential to offer patients earlier departure from expensive hospital facilities and could facilitate independent home care.

Applications of mote sensors pose challenges such as reliability, limited battery life, small size, and restricted data storage and processing capability. In order for these applications to be viable, e.g. in order to be embedded in a patient's body, mote sensors have to be small, light-weight and battery-powered. Therefore, the energy consumption of the underlying hardware is of paramount importance. The design of mote sensors needs to include self-testability to guarantee reliable operation in remote locations.

In wireless sensor networks, channel access is uncoordinated, which means packet transmissions are asynchronous and occur at random time instants that are unknown at the receiver. An asynchronous low-power scheme for wireless packet detection, processing and communication is thus necessary. A novel hypothesis-based preamble detection method for uncoordinated, high-density packet-based communication has been developed [11] for power-constrained wireless sensor networks. This method proposes the use of a preamble positioned at the beginning of each packet transmission to facilitate the detection process and timing recovery. The goal of this research is to enhance this energy-aware physical-layer protocol and to implement the transmitter circuits in a prototype silicon chip.

Main objectives of this research are:

- To investigate the first custom chip implementation of the asynchronous transmitter. The underlying hardware has to be simple and efficient in order to keep the power consumption at a satisfactory low level.
- To investigate suitable (effective and compact) design-for-testability and design-for-characterization features.
- To enhance the receiver design, in terms of both area and speed, and to measure the performance of the implemented enhanced design.

As the first step toward integration, a bit-true implementation of the algorithm on a field-programmable gate array (FPGA) was performed previously [12] to demonstrate the functionality of the algorithm on hardware and to provide a promising starting point for further integration and other improvements. The next step is the custom implementation of the algorithm in silicon. In addition to demonstrating the correct and at-speed operation of the new packet transmitter in silicon, another primary milestone of this research was to investigate on-chip features that simplify and speed up design characterization and prototype testing.

We fabricated the transmitter design of the algorithm in silicon using commercial design tools and technologies subsidized by CMC Microsystems (Kingston, ON). The target CMOS technology for this chip was IBM Corporation's 130-nm digital process [13] [14]. Prototype testing included verifying the operation of the design at different power supply voltages, and experimentally measuring its performance and limits. In order to simplify the process of testing, the prototype chip consists of a baseband transmitter and the built-in self-test (BIST) logic. The embedded BIST circuitry [15] in our chip accelerated the characterization of the design.

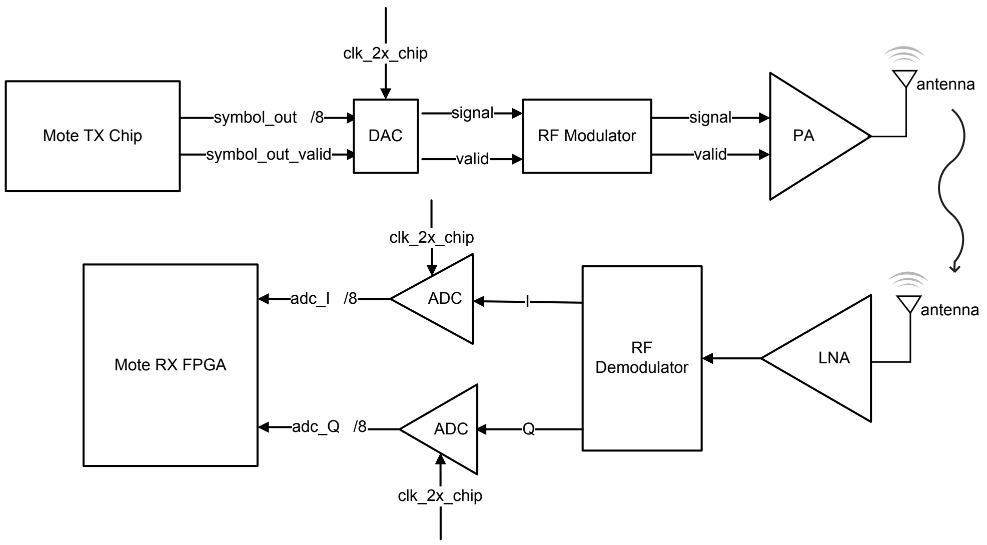


Figure 1.1: Transceiver Baseband + RF Link.

The specifications of the transceiver design presented here is defined based on

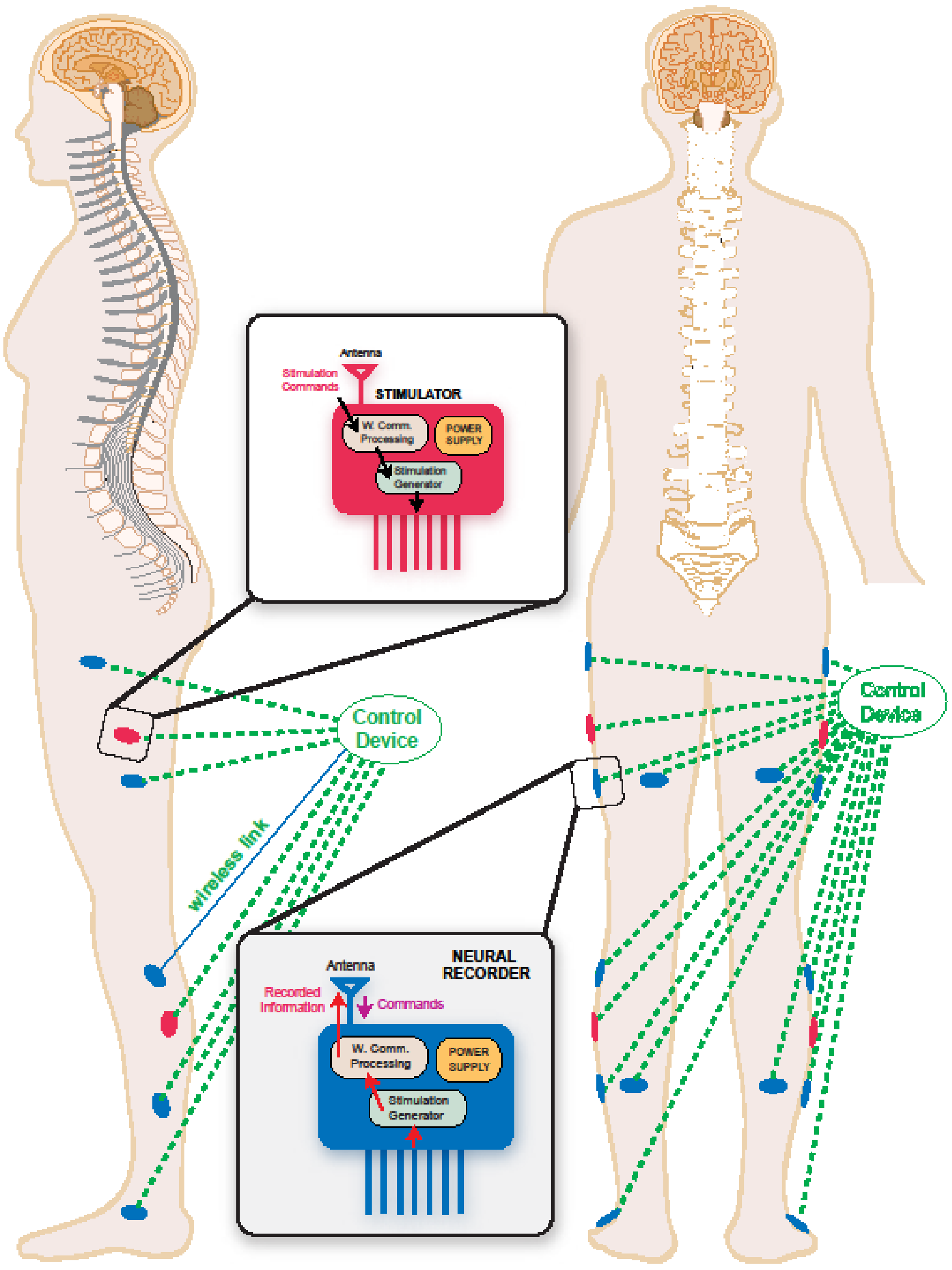


Figure 1.2: Medical Application, Body Sensors [1].

a potential medical application. In this application, as shown in Figure 1.2, the transmitter nodes are implanted sensors that capture neural signals and aim to communicate these signals to an external receiver which is implemented on an external FPGA board. At the transmitter, the digitized neural signals form the information data in the payload field of the packets. The receiver should be able to incorporate the extraction of the information data from different implanted nodes.

Figure 1.1 shows the transceiver link including RF and antenna. Since in the current state of the project, the payload data is predefined and is generated internally, the mote TX chip module doesn't capture the signal from outside. Instead it performs the required signal processing on the packets generated internally and passes them to the DAC and then the RF modulator and eventually the antenna. On the receiver side, the captured signal from antenna passes through the RF demodulator and then on to the ADC (actually two ADCs operating in parallel with 1/2 sample time phase offset). Then, the digital stream of samples is processed by the receiver design (implemented in the RX FPGA) to recover the transmitted packet.

1.3 Literature Review

Depending on the nature of the application, wireless motes may have low data rates and bursty traffic, with relatively low duty cycles, for transmission. Often motes have to transmit data at random times, therefore random-access, packet-based communications protocols are appropriate for such networks. Several examples of WSNs in the research community, which have been constructed using wireless transceivers built to the IEEE 802.15.4 low-rate wireless personal area network standard, are available [16] [17] [18] [19].

The specifications of two examples of mote platforms and chips which are compliant with IEEE 802.15.4 are explained here:

- TelosB mote platform, from Crossbow [20]: This platform provides a data rate of 250 kbps within the 2.4-GHz to 2.4835-GHz ISM band. It includes a Texas Instrument microcontroller (TI MSP430) with 10 kB of RAM and integrated onboard antenna. It is suitable for low-power WSN experiments. It has an open-source operating system and optional integrated temperature, light and humidity sensor.

- Chipcon CC2420 chip from TI [21]: This is a single chip, 2.4-GHz ISM band RF transceiver for low-power and low-voltage wireless communications with an effective data rate of 250 kbps.

Medical applications such as visual and neural prostheses, or invasive brain-computer interface (iBCI), which collect a massive amount of data from the neural system and transfer the data across the skin to the outside of the body to control the patient after signal processing, have recently emerged [22] [23] [24]. These applications introduce new challenges in the design of the transceiver link such as extremely limited power, in some cases utilizing energy harvesting circuitries, and small size while establishing wide-band and robust connection.

In [25], the design of an implantable 2.4-GHz RF transmitter for wireless biotelemetry systems is presented. The presented design is an on-off keying (OOK) transmitter fabricated in a $0.18-\mu$m CMOS analog process. This transmitter achieves an energy efficiency of 22 pJ/bit with an associated bit error rate of 1.7×10^{-3} without utilizing any error correction scheme to transmit OOK data at a 136-Mbps rate.

It is argued in [26] that the commercially-available wide-band wireless protocols, such as Bluetooth and WiFi, normally don't match the requirements of these new medical applications. For example the choice of carrier frequency of these standards is inappropriate for implanted devices. Therefore, there is a need for a low power, wide-band, and robust wireless scheme which is applicable to high-performance medical applications. For example, [26] represents a low-power wireless transceiver, which operates based on pulse harmonic modulation, fabricated in a $0.5-\mu$m standard CMOS process. This transceiver achieves a 10.2-Mbps data rate with a bit error rate of 6.3×10^{-8} at 1 cm distance. The transmitter power efficiency is 345 pJ/bit at 1 cm distance.

In [27], a 128-channel wireless neural recording IC with on-the-fly spike sorting and a UWB transmitter fabricated in $0.35-\mu$m CMOS process is presented. This scheme is suitable for short range, high data rate wireless communication within the 3.1-GHz to 10.6-GHz spectrum, providing a data rate of 90 Mbps. The transmitter power efficiency is 18 pJ/bit.

Preamble detection and frame synchronization is an important task in packet communication and has been studied in literature [28] [29]. It was shown in [30] that optimal maximum likelihood (ML) detection for BPSK signaling performs about 3 dB better than correlation detection. The generalized results to M-ary coherent and non-coherent signaling is provided in [31]. Regardless of the sub-optimal performance of the correlation-based preamble detection, it is preferred because of its implementation simplicity. The performance of both parallel and serial non-coherent preamble detection methods have been studied. Although parallel detection methods [32] [33] can increase the speed of the code acquisition process significantly, their complexity goes up quickly. Therefore, there is a trade-off between acquisition time and implementation complexity. For low-cost implementation, the serial acquisition methods are preferred. In [34] and [35], a general theory for serial non-coherent code acquisition is demonstrated. A chip-differential code acquisition scheme is studied in [36] where the received signal is first differentially decoded and then added and compared to a threshold. It faces performance degradation due to an increased noise variance during chip-differential demodulation and detection.

The focus of this thesis is the implementation of a preamble detection scheme with low complexity for low-power, wide-band, high-data-rate WSN applications. The preamble structure implemented in this thesis is similar to that of the synchronization header found in the IEEE 802.15.4 standard. Both headers consist of 40 differentially-encoded symbols and use chip modulation employing direct sequence spreading. The 802.15.4 standard performs detection using a 32-symbol preamble and timing synchronization using an 8-symbol start-of-frame delimiter [37]. The preamble detection scheme presented in [11] performs both detection and frame synchronization jointly, which more efficiently uses the available symbols, by continuously computing decision statistic for the presence of the preamble at each timing instance. To lower the computational complexity of this process, rather than utilizing one long spreading sequence spanning the entire 40 symbols, the same sequence is repeated for each of the 40 symbols [12]. Although the design is a asynchronous detector, it was shown in [11] that when two samples per symbol are acquired for the detection, the detector performs close to the chip-synchronous detector. It was also shown that the design is quite robust to frequency and timing offset. The low complexity design of this transceiver provides the potential of high data-rate communication, and thus makes it a suitable candidate for newly emerged medical applications of WSNs. The objective here is to investigate the potentials of power saving in the implementation of the aforementioned communication scheme.

1.4 Thesis Overview

The focus of this research was to fabricate the baseband transmitter and to improve the implementation of a baseband receiver design of a MAC protocol, for WSNs, that was previously proposed by our research group. A brief description of the protocol and the MATLAB simulation results of the scheme are given in Chapter 2. Chapter 3 describes the fabricated transmitter design. It describes in detail the architecture of the fabricated chip, the testing results for the silicon prototypes. Chapter 4 describes the receiver design, and its FPGA implementation. It also includes the results of the evaluation of the transceiver design once it is implemented on the FPGA. Chapter 5 concludes the thesis. The digital design flow and the details of the steps followed to generate the layout for the transmitter design is provided in Appendices A and B and C and D.

Chapter 2

Transceiver Design

In this chapter we review the theory behind asynchronous, packet-based wireless communication as presented in [11]. The details of the implementation of this scheme are described in the following chapters. This chapter is organized as follows. Section 2.1.1, 2.1.2 and 2.1.3 describe the transmitter, the channel and the receiver model. Section 2.2 presents the simulation results.

2.1 A Packet-based, Asynchronous Communication Scheme

A robust preamble detection algorithm is necessary in packet-based wireless sensor networks. With uncoordinated channel access, packet transmissions are asynchronous and occur at random time instants that are unpredictable at the receiver. To facilitate payload detection and demodulation, a preamble containing a known sequence of bits is transmitted immediately before the payload. The preamble is a binary sequence with low aperiodic correlation [38]. The function of the preamble is to inform the receiver about the presence of a packet and to enable the receiver to acquire the payload start time and to reliably detect it. A receiver searches continuously for the presence of a preamble and to recover timing and synchronization information [12].

The block diagram of the transceiver link, including channel, is shown in Figure 2.1. The details of the transmission and reception is as follows:

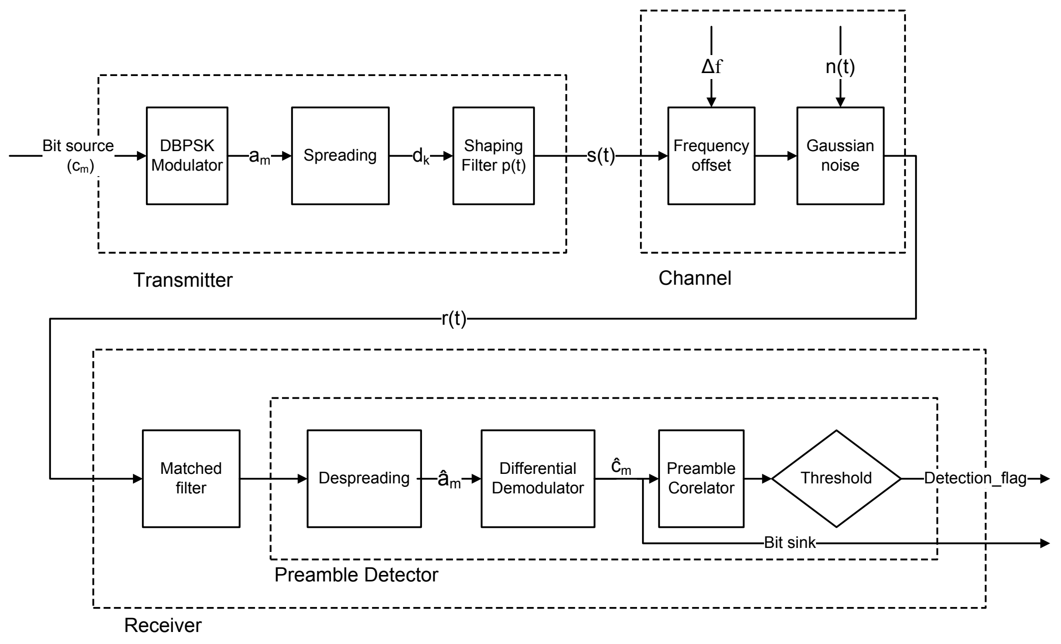


Figure 2.1: Transceiver Link System Model.

2.1.1 Transmitter Model

The communication scheme is packet-based where each packet consists of a preamble, a physical layer header (PLH) and a payload. The pre-known binary sequence with low aperiodic correlation, called the preamble, is prepended to each packet. The details of packet structure are explained in Section 3.1. The modulation scheme for preamble is binary phase shift keying (BPSK). Here we use the same symbols and definition as presented in [11] for the description of the system.

The preamble sequence of length W, $c = (c_1, c_2, \cdots, c_W)$, is composed of independent and identically distributed BPSK symbols from the binary alphabet $\{-1, 1\}$, where the probability $P(c_i = 1) = p = 1 - P(c_i = -1)$. The BPSK symbols are differentially encoded into a second sequence $\{a_m\}$, where $a_m = c_m a_{m-1}$ with initial value $a_1 = c_1$. The direct-sequence spread spectrum (DSSS) transformation is then performed to spread the bits into the faster bit-rate chips. The random but known sequence of length L_b, $b = (b_1, b_2, \cdots, b_{L_b})$, is called the spreading (chip) sequence. In our work $L_b = 16$. The sequence $\{a_m\}$ is thus oversampled by a factor of L_b and then multiplied by the spreading sequence (m-sequence) $\{b_k\}$ resulting in the chip sequence $\{d_k\}$, for $k = 1, \cdots, L_b W$. The time interval is given by $T_b = L_b T_c$, where T_c denotes the chip duration and the T_b denotes the bit duration. The spreading sequence $\{b_k\}$ consists of independent and identically distributed random variables with $P(b_k = 1) = P(b_k = -1) = 1/2$. The envelope of the transmitted signal $s(t)$ can be formulated as:

$$s(t) = \sqrt{E_c} \sum_{k=1}^{L_b W} d_k p(t - kT_c - i_0 T_c), \tag{2.1}$$

where E_c is the energy per chip. Following convention, the pulse shaping function $p(t)$ is normalized to unit energy. The assumption is that the preamble starts at time i_0 in the discrete time interval $[0, \infty)$.

2.1.2 Channel Model

We assume that the communication occurs over an additive white Gaussian noise (AWGN) channel. Thus the received signal before sampling is represented as

$$r(t) = e^{j(2\pi \triangle f t + \phi)} s(t) + n(t), \tag{2.2}$$

where $\triangle f$ and ϕ denote the unknown carrier frequency offset and phase offset, respectively. The frequency offset is assumed to be a constant value but the phase offset is a uniform random variable in the interval $[0, 2\pi)$. The additive noise $n(t)$ is a zero-mean complex-valued Gaussian white noise process with independent real and imaginary components, each with variance $N_0/2$.

2.1.3 Receiver Model

Assuming perfect chip timing knowledge at the receiver, after chip-matched filtering and sampling, the baseband output is formulated as

$$r(k) = e^{j(2\pi \triangle f k T_c + \phi)} s(k) + n(k). \tag{2.3}$$

To obtain $\hat{a}_m$, a complex-valued estimate of a_m, L_b consequent samples are despread and summed, i.e.,

$$\hat{a}_m(k) = \frac{1}{\sqrt{L_b}} \sum_{l=(m-1)L_b+1}^{mL_b} r(l+k)\, b_l, \quad m = 1, \cdots, W. \tag{2.4}$$

The next step towards preamble detection is differentially correlating the sequence $\{\hat{a}_m(k)\}$ with the known preamble sequence to obtain the correlation statistic. The correlation statistic at the kth time interval is represented by

$$\eta_k = \sum_{m=2}^{W} \hat{a}_m(k)\, \hat{a}^*_{m-1}(k)\, c_m. \tag{2.5}$$

We note that the real part of the correlation statistic η_k is a sufficient statistic [11]. The preamble detection algorithm computes $Re(\eta_k)$ for every considered time shift. The delay between subsequent time shifts is equal to or greater than half the nominal chip period. A correlation statistic $Re(\eta_k)$ of greater than a threshold G indicates the presence of a preamble defined as: $min\{k \in [0, \infty) : Re(\eta_k) \geq G\}$, where $G \in \Re$ is the threshold. The data-decoding functions are initiated right after preamble detection.

The block diagram of the receiver architecture is shown in Figure 4.1. As reported in [11], a detector with two samples per chip performs essentially as well as a synchronous system. Thus the ADC's sampling rate is set to twice the expected chip rate. These samples are sent over two parallel sections that perform preamble detection. Despreading the received samples with the known m-sequence results in estimates of the transmitted symbols $\hat{a}_m$. These estimates are then differentially decoded and correlated with the known preamble sequence. Only the real part of the correlation statistic is used and compared with a threshold to make a decision. A detection in either of the two parallel sections results in detecting a preamble. Each of the two parallel sections involves the operations of symbol despreading, differential decoding and preamble correlation.

Figure 2.2 shows the implementation of the scheme in our simulations, assuming

$L_b = 16$. The received samples (chips) are stored in a shift register (SR) of length L_b. For every sample, despreading with the known m-sequence is performed and the resulting estimation of $\hat{a}_m$ is stored in another set of shift registers. Starting at index zero and taking every 16th value, a differential decoding is performed. Then the cross-correlation of the resulting bit sequence $\hat{c}_m$ with the known preamble sequence c_m is computed and compared to a suitable threshold to determine if a preamble is detected.

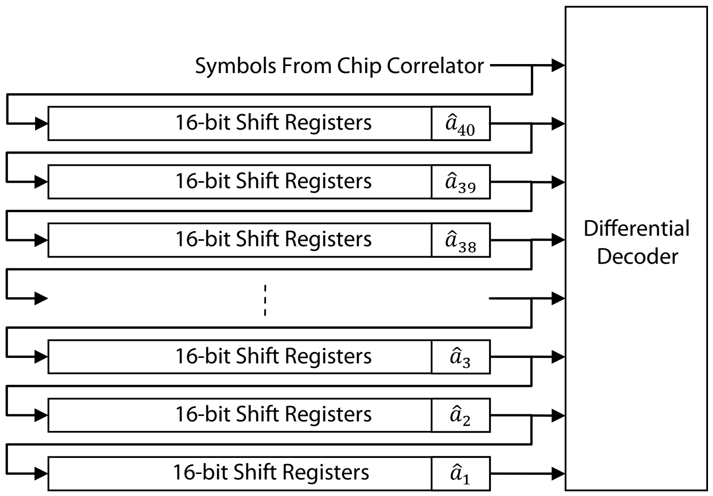


Figure 2.2: Preamble Detector Implementation.

2.2 Simulation Results

Conditioned on the presence of the preamble at moment i_0, two error events are defined as follows: First, the decision statistic of greater than the threshold G, at moment $k \neq i_0$, determines the erroneous presence of a preamble. Such an event is called a *false alarm* here and its probability is denoted by P_{false}. Second, if the correlation statistic at moment i_0 is less than the threshold G, a preamble is missed. Such an event is called a *miss* and its probability is denoted by P_{miss}. The analysis of the performance of the described preamble detector is out of the scope of this thesis, but a brief description of simulation results is given here.

The performance of the scheme was measured for the SNR values shown in

Table 2.1: SNR values for chip, bit and the preamble.

A_c	E_c	**cSNR**	**sSNR**	**pSNR**
0.125	0.015	−21.07	−9.07	6.92
0.1875	0.035	−17.55	−5.55	10.45
0.25	0.0625	−15.05	−3.05	12.94
0.3125	0.097	−13.1	−1.1	14.9
0.375	0.14	−11.55	0.45	16.45
0.4375	0.191	−10.19	1.81	17.81
0.5	0.25	−9.03	2.97	18.97
0.5625	0.316	−8	4	20
0.625	0.39	−7	5	21
0.6875	0.4726	−6.25	5.75	21.75
0.75	0.5625	−5.5	6.5	22.5
0.8125	0.66	−4.8	7.2	23.2
0.875	0.7656	−4.16	7.84	23.84
0.9375	0.879	−3.57	8.43	24.43

Table 2.1. This table includes SNR values at the preamble (pSNR), symbol (sSNR) and chip level (cSNR) level as well as the energy per chip (E_c) and the amplitude of a rectangular pulse (A_c) (as the shaping pulse) assuming $L_b = 16$ and $W = 40$. The channel is assumed to be AWGN where the real and imaginary part of the noise each are unit power Gaussian random variables. The values in the table are calculated using the following formulas:

$$E_c = A_c^2, \tag{2.6}$$

$$cSNR = 10log(E_c/2), \tag{2.7}$$

$$sSNR = cSNR + 10logL_b, \tag{2.8}$$

$$pSNR = sSNR + 10logW \tag{2.9}$$

Table 2.2: Parameters used in the MATLAB simulations.

c	$40'h2481F1539C$
b	$16'h8DC6$

Table 2.2 shows the value of the parameters used for the simulations.

Figure 2.3 shows the P_{miss} versus the $pSNR$ plot obtained from MATLAB simulations with no timing mismatch. The noise samples here are the captured samples

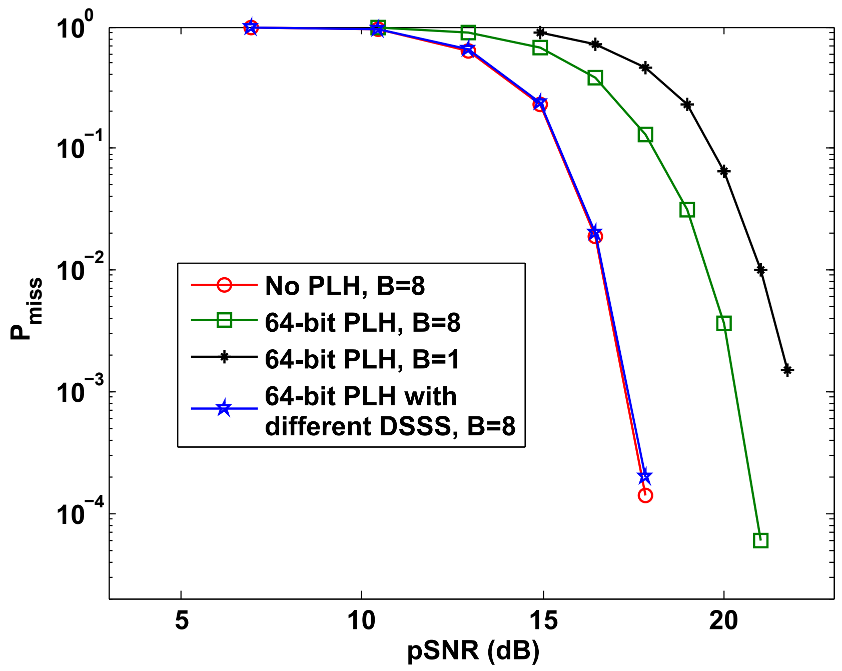


Figure 2.3: Simulated P_{miss} vs. pSNR for $P_{false} = 10^{-3}$.

(obtained using the ChipScope waveform viewing tool) from the FPGA implementation. This figure compares the performance of the preamble detector where the samples input to the detector are either 8 bits or 1 bit (sign bit only). The performance of the detector with one bit input symbols is attractive for the lowest power applications. The PLH contains a 16-bit packet ID with a repetition code of rate 4, thus the length of the PLH is 64 bits in this simulation. Note that the PLH and preamble use the same spreading sequence. This figure also includes the performance of the preamble detector (presented in [11]) where the packets contain no PLH and input samples to the preamble detector are 8-bit samples. As shown in the figure, assuming the same spreading sequence for both the preamble and the PLH, the performance of the preamble detector drops by about 3 dB in the presence of PLH. The reason is that, for a fixed threshold value, the presence of PLH increases the false alarm rate. Thus to keep the false alarm rate fixed, we need to increase the threshold in the presence of PLH, which results in a higher miss rate. The simulation results in this figure show the degradation in performance caused by the presence of the PLH can be avoided if a different DSSS is used for spreading the PLH. This figure also shows that the performance of the preamble detector with 1-bit input samples degrades insignificantly (about 1 dB) compared to the preamble with 8-bit input samples where the PLH length is the same.

Figure 2.4 shows the performance of the preamble detector in the presence of frequency offset where $pSNR = 19dB$ and $P_{false} = 0.001$. The figure compares the performance where the input symbols to the preamble detector contain either 8 bits or 1 bit (sign bit only). Simulations show that for $\Delta f T_c < 0.001$, the performance degradation is insignificant and thus the preamble detector is robust to the frequency offset provided that $\Delta f T_c < 0.001$ in either case (B=1 or B=8).

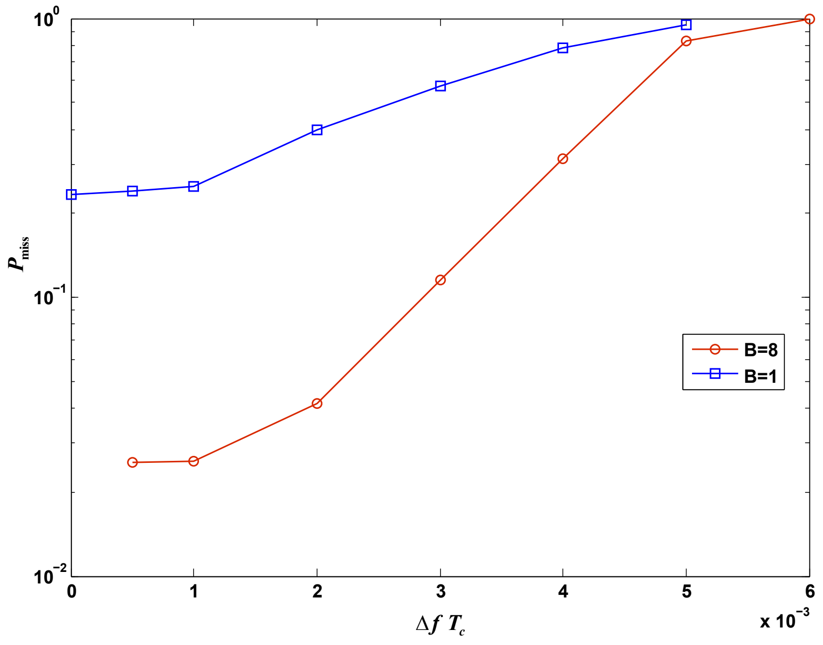


Figure 2.4: Simulated P_{miss} vs. $\triangle f T_c$ for $P_{false} = 10^{-3}$ and $pSNR = 19dB$.

Chapter 3

Transmitter Chip

This chapter details the function and specifications of the digital baseband communication circuitry of the wireless sensor (mote) transmitter implemented as a semi-custom (standard cell based) 130-nm digital CMOS VLSI chip based on the designs presented in [11, 12] for a specific application.

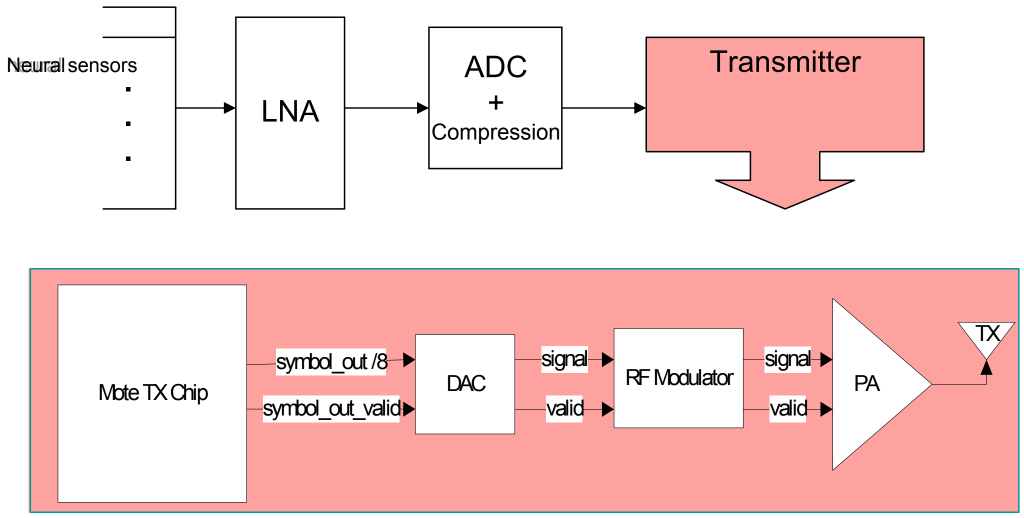


Figure 3.1: TX Node Front-end in the Medical Application.

The detailed specifications of the design (for example, packet specifications and data rate) are defined based on a potential medical application. Figure 3.1 illustrates the architecture of the transmitter node to be implemented for the potential medical application (explained in Section 1.2). In particular, the figure shows where our design (digital baseband asynchronous transmitter, called the “mote TX chip” in this chapter) appears in the transmitter subsystem that follows the analog-to-digital converter. The transmitter chip (TX chip in Figure 3.1) is supposed to perform necessary processing to set the packet-based digital asynchronous baseband communication. Our baseband digital transmitter is intended to be integrated eventually with DAC+RF circuitry to form a complete wireless node, as shown in Figure 3.1.

This chapter aims to explain the following details:

- Transmitter functionality and packet structure in Section 3.1.
- Chip architecture (sub-modules and their tasks) in Section 3.2.
- Chip pinout and the external interface of the chip in Section 3.3.
- Digital design flow for fabrication in Section 3.4.
- The chip’s test platform and results in Section 3.5.

3.1 Transmitter Functionality and Packet Structure

Since the communication in our design is packet-based, the transmitter generates a preamble and prefixes it to the payload whenever there is one payload to be sent out. The packet structure is illustrated in Figure 3.2. Note that each packet consists of a preamble, a physical layer header (PLH), and the payload.

Preamble
PLH
Payload

Figure 3.2: Packet Structure.

Figure 3.3 shows a block diagram of the baseband digital transmitter unit (TX unit) that was designed for this thesis. In this figure, the symbol rate of the output of each component is labeled on their connection lines. According to [11], bits in the packet are differentially encoded to provide immunity to unknown carrier phase drift, and then they are modulated by a direct-sequence spread spectrum (DSSS) *m*-sequence (spreading sequence) of length $\mathtt{L}_b$ chips per bit at the transmitter. The transmitter unit also includes a chip pulse shaping filter .

There could be two separate *m*-sequences, with potentially different lengths, for the preamble and payload. The payload *m*-sequence utilizes a unique, packet-specific spreading sequence to enable multi-packet reception. Note that the design for this thesis doesn't include the payload processing; instead it assumes a previously spread payload chip sequence is available to the TX unit.

A differential correlation type preamble detector is employed along with a threshold-based decision algorithm at the receiver to locate in time the exact start of the payload.

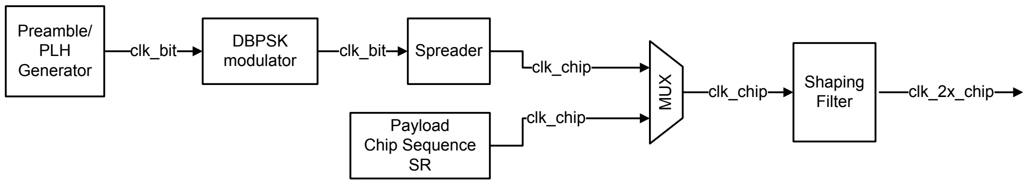


Figure 3.3: TX Unit Block Diagram.

Table 3.1: Implementation parameters of the design.

Parameter	**Value**	**Notes**
Preamble length	5 Bytes (40 bits)	IEEE 802.15.4 Standard
PLH length	8 Bytes (64 bits)	
Preamble modulation	DBPSK	
$\mathtt{L}_b$	16	
Payload length	512 chips	
Packet chip rate	Maximum 19.496 Mcps	
Bandwidth	25 MHz	3dB bandwidth
Shaping filter	12-tap RRC FIR filter	40dB image rejection

Table 3.1 shows the key implementation parameters for our design. The rate parameters are defined by the potential medical application. The length of the *m*-sequence for the preamble is 16 chips per bit. The preamble is a known sequence of 40 bits prepended in front of the PLH. The PLH is 64 bits which can include payload length, packet ID, payload *m*-sequence, etc. The goal of this design is to fabricate a simplified transmitter that only performs the required processing (modulation,

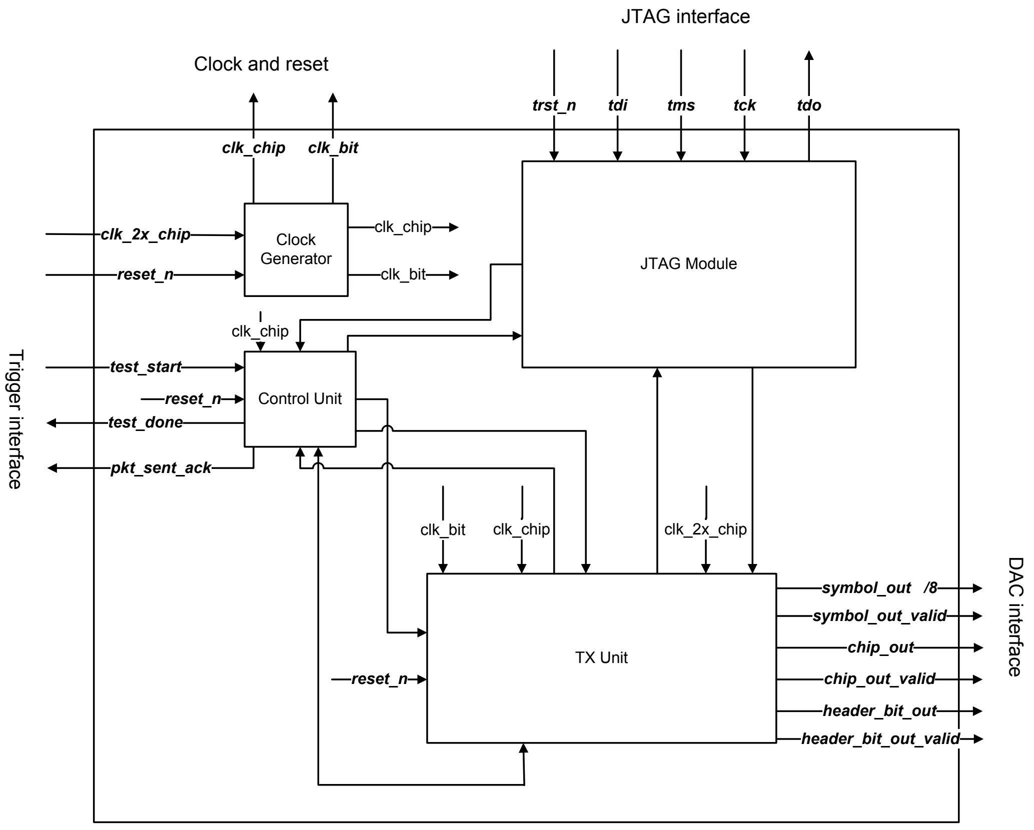


Figure 3.4: Chip Architecture.

spreading, ...) for the preamble and PLH, but not the payload. Note that the previously spread payload sequence (called the payload chip sequence) could be loaded into the TX unit. In that case, the payload chip sequence would be appended to the PLH and then shifted out. The payload chip sequence length is assumed to be 512 chips. Channel coding should be considered for the payload data in the pre-processing; otherwise, the BER could be unacceptably high even assuming a reasonable range of SNR (the reasonable range of cSNR is -20dB to -4dB which corresponds to sSNR values in the range -8dB to 8dB, where with no coding would result in a higher BER [7]).

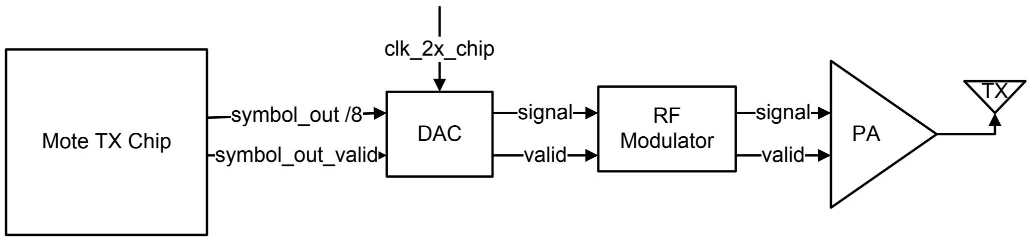


Figure 3.5: Chip Interface with Analog and RF Components.

3.2 Chip Architecture

Figure 3.5 shows the interfaces between the mote transmitter chip and the analog and RF components. As explained above, the payload is generated internally in this test chip. Once there is a packet to be transmitted, its sequence of symbols appears on the `symbol_out` bus and the `symbol_out_valid` signal stays high for this period. The `symbol_out_valid` signal provides the possibility of turning off the DAC and the RF circuitry when the transmitter is silent (no packet to communicate) and could be turned off to save energy.

Figure 3.4 shows the modules within the mote transmitter chip and their interface with each other and the input/output pins. The input/output signals could be grouped into four major external interface signal groups:

- JTAG interface
- DAC interface
- Clock and reset
- Trigger interface

A brief description of each module's input/output and functionality is given below.

3.2.1 JTAG System

Figure 3.6 shows the JTAG system with its input/output signals (`tdi`, `tdo`, `tms`, `tck`, `trst_n`). JTAG registers are used in the design for holding key operating parameters.

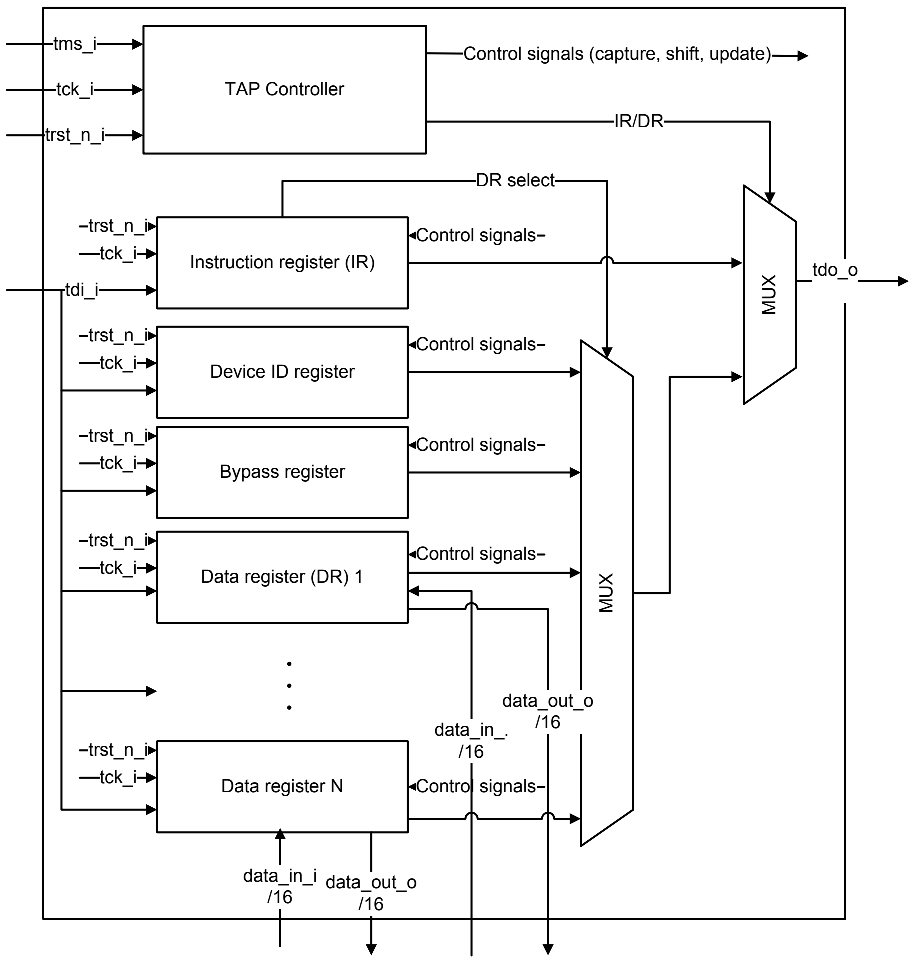


Figure 3.6: JTAG System Block Diagram.

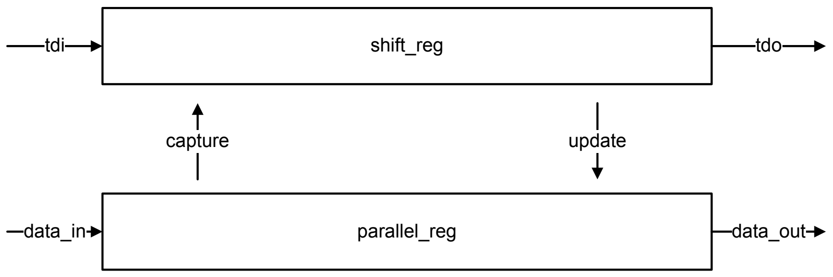


Figure 3.7: JTAG DR Structure.

Figure 3.7 shows the structure of one data register. Each JTAG data register contains a working register (bottom register in Figure 3.7) whose content can be captured and updated in parallel. A shift register (top register in Figure 3.7) is also provided alongside the working register so that the captured state of the working register can be shifted out serially without disturbing the working register. JTAG data registers (DR) have `data_in_i` and `data_out_o` buses with parallel access. Each register inside the chip that needs to be monitored or configured is implemented as a JTAG data register.

If we want to configure an internal register inside the chip, where data register $\mathtt{DR}_x$ is associated with it, we must complete the following steps:

- Serially load the JTAG instruction register `IR` so that it connects $\mathtt{DR}_x$ across `tdi_i` and `tdo_o` as the currently addressed JTAG data register.
- Serially load the configuration data via `tdi_i` into $\mathtt{DR}_x$.
- Update the `data_out_o` of $\mathtt{DR}_x$ into the target internal register.

If we want to monitor an internal register inside the chip, which is the currently-addressed data register $\mathtt{DR}_y$, we should

- Capture the target internal register onto the `data_in_i` of $\mathtt{DR}_y$.
- Shift the data serially out from `tdo_o`.

Figure 3.8 shows the TAP controller state machine. The `tms_i` control input is used to determine the state transition that occurs on each rising edge of `tck_i`. By

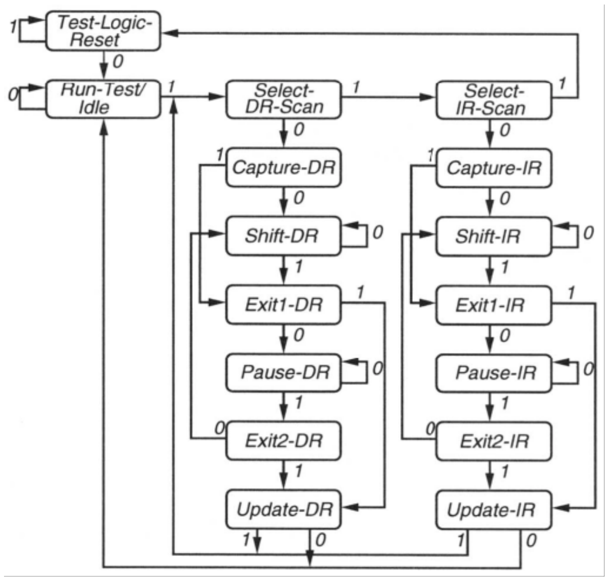


Figure 3.8: TAP Controller State Machine.

choosing sequences of `tms_i` values appropriately, the state machine can be caused to produce useful test actions. For example, the state sequence in the center is used to capture, shift and update the currently-addressed data register. The state sequence at the right is used to capture, shift and update the instruction register (IR). A bit field in the IR determines which of the data registers is currently addressed. The input `tms_i` and `tms_i` are sampled at the rising edge of `tck_i`. The output `tdo_o` is latched at the falling edge of the `tck_i`.

The list of implemented JTAG data registers in our design is given in Table 3.2. For each data register, the register width and initial value are given. The table also shows whether the data register is read-only or both readable and writable. Read-only data registers are used to monitor internal registers which don't need to be updated through the JTAG port.

A brief description of each data register is given below. For each data register, a default value (not necessarily all-zero value) is automatically loaded when `trst_n` is asserted low. If the register is writable then the default value can be overwritten serially through the JTAG port before the chip is used.

`preamble_sequence` This data register configures the preamble of the packets. The

Table 3.2: TX JTAG Data Registers.

Register Name	**Width**	**Initial (default) value**	**R/W**
`preamble_sequence`	40	40'h2481F1539C	RW
`preamble_spreading_sequence`	16	16'h066B	RW
`plh_sequence`	64	64'h00FF00F0000F0000	RW
`inter_packet_spacing`	16	16'h00	RW
`total_packet_number`	32	32'h00F00000	RW
`tx_filter_coeff`	96	{8'h00,8'hFF,8'h04,8'hF9, 8'hF6,8'h34,8'h66,8'h34, 8'hF6,8'hF9,8'h04,8'hFF}	RW
`payload_chip_sequence`	512	{8{64'h222222DDDD 22DD22}}	RW
`mux_sel`	4	4'hC	RW
`plh_sequence_counter`	16	16'h0	R
`enable_status`	3	3'h0	R

initial value of this register is defined by the 39-bit (note that the packet bits are differentially encoded, thus the LSB can be ignored) low aperiodic correlation binary sequence given in [38].

`preamble_spreading_sequence` This data register stores the spreading sequence used for both the preamble and PLH.

`plh_sequence` The default contents of the PLH of a packet is the value of an automatically incrementing 16-bit counter whose output is coded with a repetition code of rate 4 (note that this counter wraps around to zero after it reaches its maximum value). However, if the `constant_plh` flag (in `mux_sel` register) is set to 1 (the default value of this flag is 0), the PLH of the packet is filled with the non-incrementing value of this data register.

`inter_packet_spacing` This data register configures the number of silent periods (with the same length as a packet with no payload!) between two consecutive packets.

`total_packet_number` This data register configures the total number of packets to be generated and sent as the termination condition (more detail in Section 3.2.2).

`tx_filter_coeff` This data register contains the coefficients of the 12-tap root raised cosine pulse shaping filter in the format of {`coeff_12`, `coeff_11`, `coeff_10`,...,`coeff_1`}.

`payload_chip_sequence` This register configures the payload chip sequence.

`mux_sel` This register configures the select line of other blocks as follow:

`mux_sel[0]` = `constant_plh`: If this bit is 0, the counter value is loaded as the PLH; otherwise the `plh_sequence` is loaded as the PLH.

`mux_sel[1]`=`silent_level`: This bit configures the silent signal level (high or low) between packet transmission.

`mux_sel[2]`=`txfilter_en`: If this bit is 1 the TX filter is enabled; otherwise it is disabled.

`mux_sel[3]`=`payload_sr_en`: If this bit is 1 the payload chip sequence shift register is enabled; otherwise the shift register is disabled.

`plh_sequence_counter` This data register monitors the value of the 16-bit PLH sequence counter. Note that this data register is read-only, which means the internal PLH sequence counter can't be updated through this data register.

`enable_status` This data register is used to monitor the enable of payload shift register, PLH shift register and preamble shift register ({`payload_sr_en`, `plh_sr_en`, `preamble_sr_en`}). Note that this data register is read-only, which means the enables of these three shift registers can't be updated through this data register.

The JTAG data registers are loaded with their default values after a reset (`trst_n`). Thus, they can always be used immediately for a test measurement after a hardware reset. However, individual `DR`s can have their default values overwritten via the JTAG port as required.

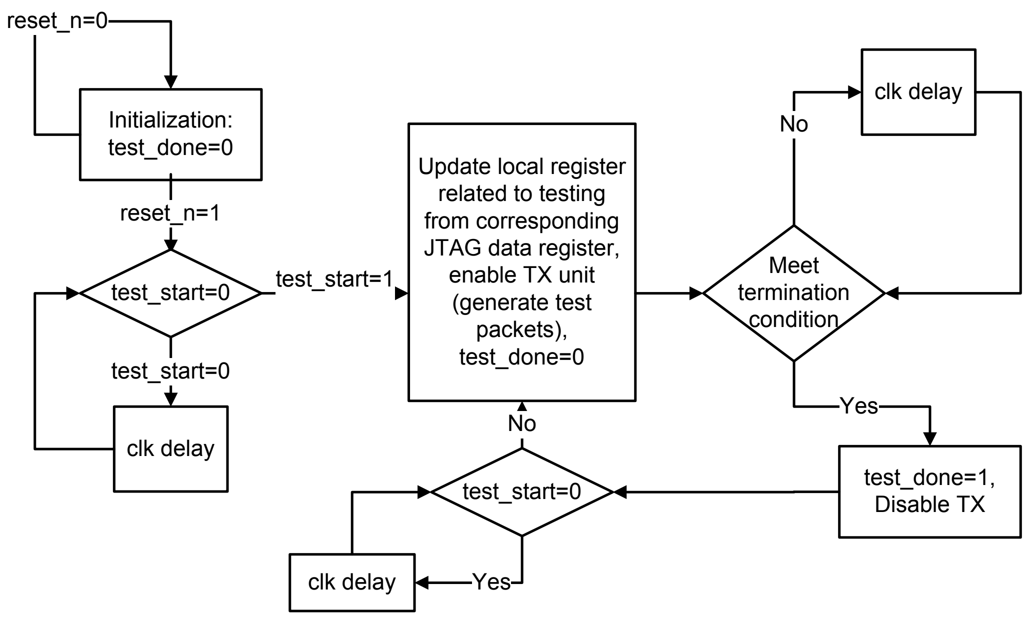

reset_n=0
Initialization: test_done=0
reset_n=1
test_start=0
test_start=1
test_start=0
clk delay
Update local register related to testing from corresponding JTAG data register, enable TX unit (generate test packets), test_done=0
Meet termination condition
No
clk delay
Yes
test_done=1, Disable TX
No
test_start=0
Yes
clk delay

Figure 3.9: Control Unit State Machine.

3.2.2 Control Unit

This module sequences the high-level test procedure (start generating packets, continue generating packets, stop generating packets) as explained here: Looking at the `test_start` and `reset_n` input signals, the control unit decides when the test is to start. Then, based on the value of the `total_packet_number` data register, it knows how many packets should be generated. Once a test is started, the control unit keeps the test running (generating packets) until the specified number of packets are generated. The `test_done` output signal is asserted to acknowledge the termination of the test. The control unit generates the enable signal for the preamble, PLH and payload shift registers.

The content of the PLH of test packets can either be a fixed pre-known sequence of bits (loaded into the `plh_sequence` JTAG data register) or it can be the present value of the counter. Using the counter value permits a simple way of measuring the miss rate and false alarm rate at the receiver. As was mentioned before, channel coding should be considered for the PLH and payload; otherwise the BER could be high even within a reasonable range of SNRs (the reasonable range of cSNR is -20dB to -4dB; which corresponds to sSNR range of -8dB to 8dB). Our design assumes that the contents of the `plh_sequence` data register are already coded. Also, when PLH is loaded with a sequence counter, a repetition code of rate 4 is being used. That means that assuming that the PLH's length is 64 bits, the sequence counter is a 16-bit counter.

Figure 3.9 shows the the control unit state machine. As it is shown in this figure, the control unit updates the local test related registers once the `test_start` signal is asserted high. Note that updates to the JTAG data registers do not change the value of local registers before the `test_start` signal goes high.

The payload chip sequence (programable through the `payload_chip_sequence` JTAG data register) may or may not be included as part of the packet. The JTAG data register `mux_sel[3]` determines if the payload chip sequence should be shifted out or if the packet should be transmitted with no payload. The bit sequence loaded in the JTAG data register `payload_chip_sequence` is assumed to be already coded and spread out to the chip rate.

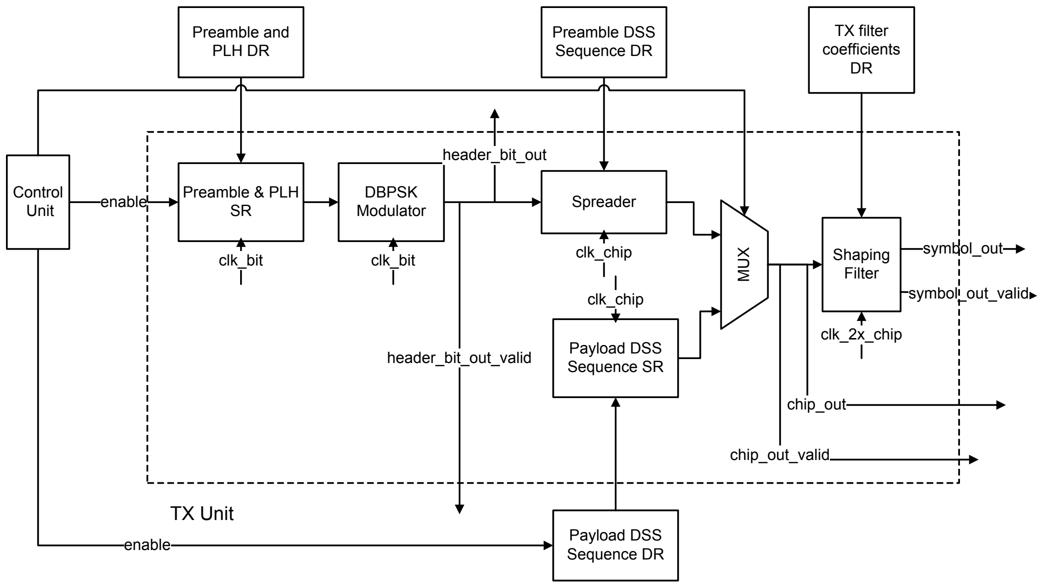


Figure 3.10: TX Unit Block Diagram.

3.2.3 TX Unit

Figure 3.10 shows the TX unit sub-modules with its input/output signals and also its interface with other modules.

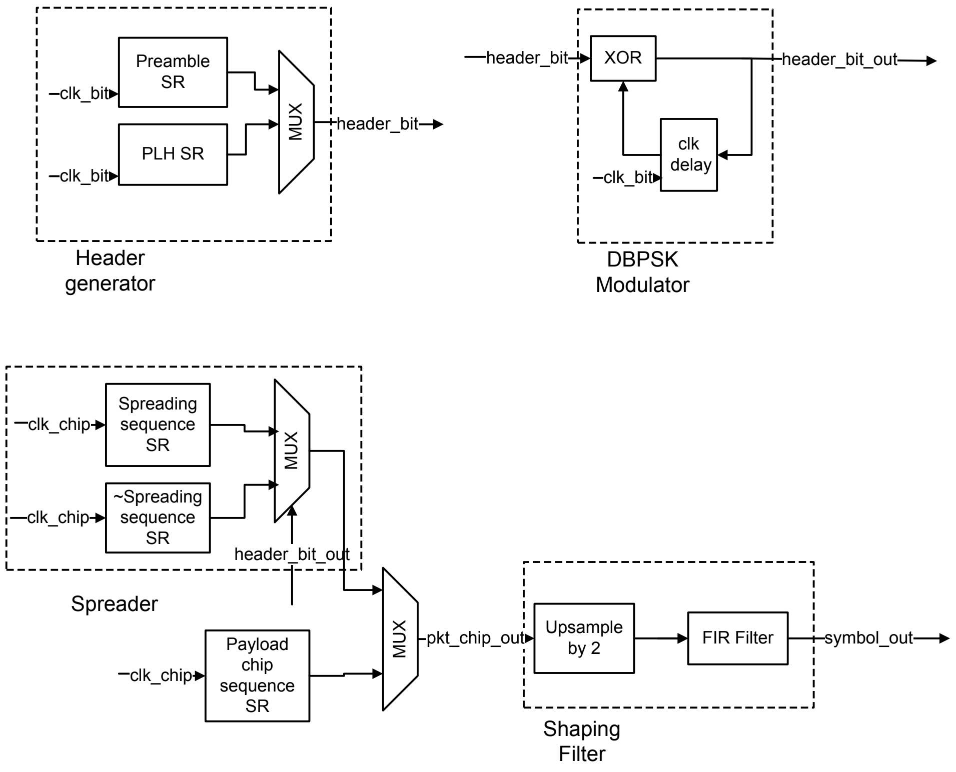


Figure 3.11: Modulator, Spreader and Shaping Filter Block Diagram.

A brief description of each sub-module is given here.

- *Preamble and PLH Shift Register*: When enabled, this shift register serially sends out the preamble and PLH binary sequence at the bit rate. This shift register is updated from the `preamble_sequence` and `plh_sequence` data registers once `test_start` is asserted high. Note that this module works with the clock signal `clk_bit`.

- *DBPSK Modulator*: This sub-module performs the differential BPSK modulation at the bit rate. The functional block diagram of this module is given in

Figure 3.11. Note that this module is clocked by `clk_bit`.

- *Spreader*: This sub-module performs the spreading at the chip rate. The spreading sequence is updated from the `preamble_spreading_sequence` data register once `test_start` is asserted high. The functional block diagram of this module is given in Figure 3.11. Note that this module is clocked by `clk_chip`.

- *Shaping Filter*: This sub-module performs upsampling and FIR filtering. The filter coefficients are updated from the `tx_filter_coeff` data register once `test_start` is asserted high. The functional block diagram of this module is given in Figure 3.11. Note that this module is clocked by `clk_2x_chip`.

3.2.4 Clock Generator

The highest frequency clock required in our design is the clock for our shaping filter (`clk_2x_chip`). The clock generator module receives `clk_2x_chip` (which is applied to an input pin of the chip) as the reference clock and derives all of the other required on-chip clocks. Here is a list of the required internal clocks and their purpose:

- `clk_bit`: To sequence the preamble and PLH bits.

- `clk_chip`: To spread the preamble and PLH bits. Since the spreading sequence in our design is 16 chips per bits, the frequency of `clk_chip` is 16 times the frequency of `clk_bit`.

- `clk_2x_chip`: To pulse shape the packet chips. Since the shaping filter upsamples its input samples by a factor of two, the frequency of `clk_2x_chip` is twice the frequency of `clk_chip`.

`reset_n` is the active low asynchronous input to this module. Figure 3.12 shows the phase relationship among the externally supplied reference clock `clk_2x_chip` and the generated clocks.

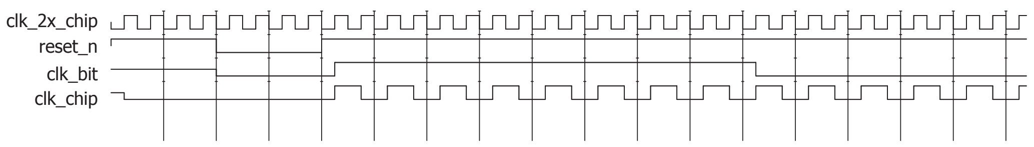


Figure 3.12: Phase Relation among the Input and Generated Clocks.

3.3 Chip Pinout

Table 3.3 describes the chip pinout. For each pin the direction, width and toggling frequency are provided.

Table 3.3: Chip Pinout.

Pin Name	I/O	Width	Toggle Freq.
`reset_n`	I	1	async
`clk_2x_chip`	I	1	$F^{(\max)}_{\mathsf{clk\_2x\_chip}}$
`tck`	I	1	$F^{(\max)}_{\mathsf{tck}}$
`trst_n`	I	1	async
`tdi`	I	1	$F^{(\max)}_{\mathsf{tck}}$
`tdo`	O	1	$F^{(\max)}_{\mathsf{tck}}$
`tms`	I	1	$F^{(\max)}_{\mathsf{tck}}$
`test_start`	I	1	$F^{(\max)}_{\mathsf{clk\_2x\_chip}}$
`test_done`	O	1	$F^{(\max)}_{\mathsf{clk\_2x\_chip}}$
`chip_out`	O	1	$F^{(\max)}_{\mathsf{clk\_chip}}$
`chip_out_valid`	O	1	$F^{(\max)}_{\mathsf{clk\_chip}}$
`symbol_out`	O	8	$F^{(\max)}_{\mathsf{clk\_2x\_chip}}$
`symbol_out_valid`	O	1	$F^{(\max)}_{\mathsf{clk\_2x\_chip}}$
`header_bit_out`	O	1	$F^{(\max)}_{\mathsf{clk\_bit}}$
`header_bit_out_valid`	O	1	$F^{(\max)}_{\mathsf{clk\_bit}}$
`pkt_sent_ack`	O	1	$F^{(\max)}_{\mathsf{clk\_2x\_chip}}$
`clk_chip`	O	1	$F^{(\max)}_{\mathsf{clk\_chip}}$
`clk_bit`	O	1	$F^{(\max)}_{\mathsf{clk\_bit}}$

A brief description of each pin is listed below. The maximum clock frequency (achieved in our prototype implementation of the design) and typical clock frequency values (appropriate for the potential medical application) of the frequencies are also reported here.

`reset_n` System reset (async, active low). Initializes the mote registers to their default values.

`clk_2x_chip` System clock. All the clocks for the different clock domains in the transmitter are derived from this clock. $F^{(\max)}_{\mathsf{clk\_2x\_chip}} = 100\,\mathrm{MHz}$, $F^{(\mathrm{typ})}_{\mathsf{clk\_2x\_chip}} = 38.4\,\mathrm{MHz}$.

`tck` Independent clock for the JTAG test registers. $F^{(\max)}_{\mathsf{tck}} = 10\,\mathrm{MHz}$, $F^{(\mathrm{typ})}_{\mathsf{tck}} = 1\,\mathrm{MHz}$.

`trst_n` JTAG reset (async, active low). Initializes the JTAG registers to their default values.

`tdi` JTAG serial test data input. The bit streams destined for the JTAG instruction register (IR) and all JTAG data registers (DRs) are fed serially into the chip through this pin.

`tdo` JTAG serial test data output. The internal scanned register data comes out of the chip serially on this pin.

`tms` JTAG test mode select to control/address JTAG functions. This serial input signal guides the JTAG state machine through its states to capture, shift and update the JTAG IR or the currently addressed JTAG DR.

`test_start` Test start signal. This signal should go high for at least one cycle of `clk_2x_chip` to trigger the control unit to start the test mode. It assumes that a valid test configuration has been loaded previously.

`test_done` Test done signal. This signal goes high after the test cycle meets the termination condition and stays high until either `reset_n` goes low, or `test_start` goes high again at the start of a new test sequence.

`chip_out` Stream of packet chips at the chip rate. This signal carries the serial stream of packet bits after spreading but before they have been shaped. $F^{(\max)}_{\mathsf{clk\_chip}} = F^{(\max)}_{\mathsf{clk\_2x\_chip}}/2$, $F^{(\mathrm{typ})}_{\mathsf{clk\_chip}} = F^{(\mathrm{typ})}_{\mathsf{clk\_2x\_chip}}/2$.

`chip_out_valid` Valid signal for stream of packet chips. This signal is high whenever the TX unit is sending chips out and is low when the TX unit is silent.

`symbol_out` Transmitter's digital baseband output at the chip rate. This signal carries the packet chips after they have been shaped.

`symbol_out_valid` TX baseband output valid signal. This signal is high whenever the TX unit is sending symbols out and is low when the TX unit is silent.

`header_bit_out` Stream of header bits at the bit rate. This signal carries the serial stream of header bits. $F^{(\max)}_{\mathsf{clk\_bit}} = F^{(\max)}_{\mathsf{clk\_chip}}/16$, $F^{(\mathrm{typ})}_{\mathsf{clk\_bit}} = F^{(\mathrm{typ})}_{\mathsf{clk\_chip}}/16$.

`header_bit_out_valid` Valid signal for stream of header bits. This signal is high whenever the TX unit is sending header bits out and is low when the TX unit is silent.

`pkt_sent_ack` Acknowledge signal for packet sent. It goes high for one cycle of `clk_2x_chip` when TX unit finishes sending one packet out.

`clk_chip` Chip clock derived from `clk_2x_chip`.

`clk_bit` Bit clock derived from `clk_2x_chip`.

3.4 Top-down Digital Design Flow

Here, we follow the digital design flows provided by University of Toronto VLSI research group [39] and also EPFL University [2]. We also used useful points provided in other design flows by other research groups [40] [41] [42]. Note that these design flows provide the general information and options for the softwares to be used in the process of fabricating a design. The custom design flow with proper options according to our transmitted design is provided in the appendix of this thesis. To complete the digital design flow we used ModelSim (version 2006.2d), Synopsys Design Vision (version `vZ-2007.03-SP5`) and Cadence First Encounter (version `v09.11-s084_1`) and Cadence (`version v2007.4_14.15`).

Fig 3.13 illustrates the typical top-down digital design flow. The following design flow provides details in four stages. The first stage is the simulation using ModelSim. The second is the synthesis stage using Synopsys Design Vision (DV). The third stage is the place and route using Cadence First Encounter (FE). Finally the last is the Design Rule Check (DRC) performed in Cadence.

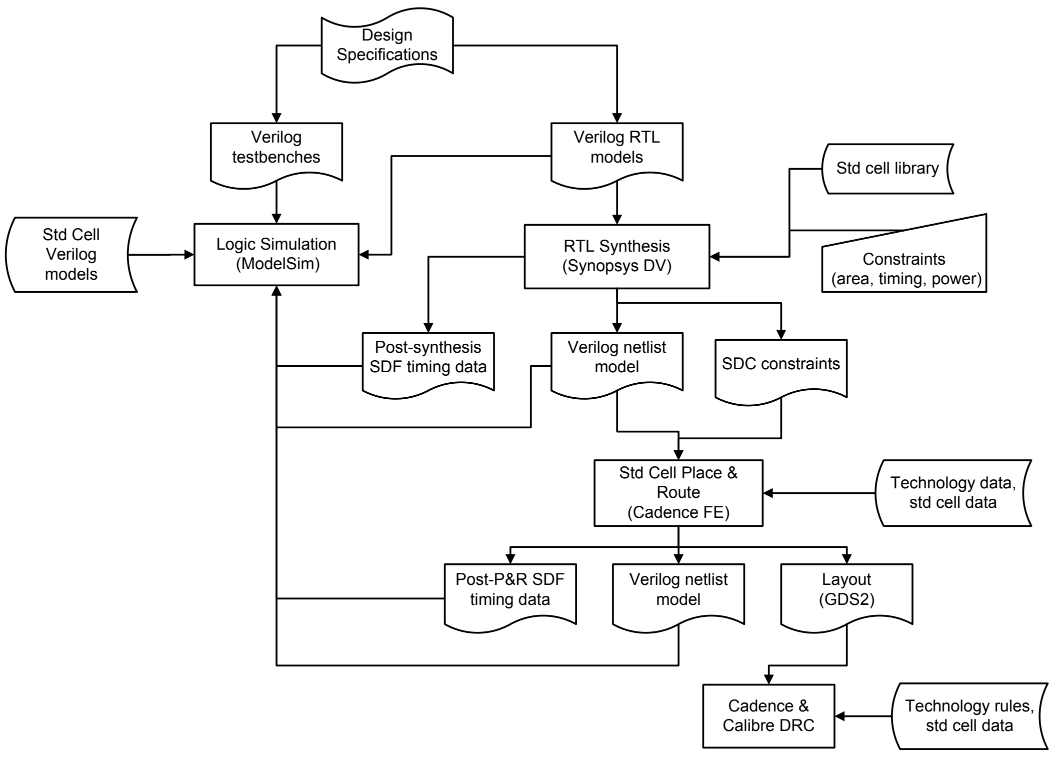

Design Specifications
Verilog testbenches
Verilog RTL models
Std cell library
Std Cell Verilog models
Logic Simulation (ModelSim)
RTL Synthesis (Synopsys DV)
Constraints (area, timing, power)
Post-synthesis SDF timing data
Verilog netlist model
SDC constraints
Std Cell Place & Route (Cadence FE)
Technology data, std cell data
Post-P&R SDF timing data
Verilog netlist model
Layout (GDS2)
Cadence & Calibre DRC
Technology rules, std cell data

Figure 3.13: Top-down Digital Design Flow [2].

The Verilog RTL models of the transmitter design are validated through simulation by means of a number of testbenches also written in Verilog using a simulator such as ModelSim. The details of the behavioral verification, post-synthesis verification and post-P&R verification are given in Appendix A.1 and A.2 and A.3 respectively.

The synthesis process infers a possible gate-level realization of the input RTL description of the design that meets user-defined constraints such as area, timings or power consumption [2]. The targeted logic gates for this design belong to a standard cell library that was provided by IBM Corp. to CMC Microsystems.

DV performs logic synthesis and design optimization. The synthesis step generates several outputs: a gate-level Verilog netlist, a Synopsys Design Constraints (SDC) file, and a Standard Delay Format (SDF) file which stores the timing data. The SDF file is typically used for post-synthesis simulation, while the first and second files are suited as inputs to the place-and-route step. The SDF file includes detailed delay information for simulation. Note that considered delays at this step are correct for the gates but they are only estimates for the interconnections [2]. The details of the steps followed by DV are given in Appendix B. The main challenge in the synthesis and implementation of our design is that it includes different clock domains (`clk_chip` and `clk_bit`) where they are generated from a master clock (`clk_2x_chip`). Since these clocks are not asynchronous, clock domain crossing is not an issue. The synthesis and implementation tools should be guided properly to ensure `clk_chip` and `clk_bit` are generated from the master clock (`clk_2x_chip` and thus the timing requirements of different clock domains are met.

Once the synthesis of the design is complete, the tool can generate timing, area and power reports for the design. Table 3.4 and 3.5 report the hierarchy power and area estimates of each submodule generated by DV, respectively. Note that the reported power doesn't include the IO. Also, the report assumes the frequency of `clk_2x_chip` is 100 MHz and the frequency of `tck` is 10 MHz.

Table 3.4: DV hierarchy power report.

Module	**Switching Power (mW)**	**Internal Power (mW)**	**Leakage Power (pW)**	**Total Power (mW)**	%
mote_tx_top	0.149	3.604	$2.45e6$	3.755	100
control_unit	$4.05e-2$	2.007	$5.82e5$	2.048	54.5
tx_filter	$2.63e-2$	0.326	$1.87e5$	0.353	9.4
tx_unit	$7.59e-3$	0.785	$4.25e5$	0.793	21.1
spreading	$3.5e-3$	$4.33e-2$	$1.91e4$	$4.68e-2$	1.2
differential_encoder	0.00	$1.44e-4$	942.658	$1.45e-4$	0.0
header_generator	$7.87e-5$	$2.36e-2$	$6.97e4$	$2.37e-2$	0.6
jtag_top	$2.19e-3$	0.452	$1.17e6$	0.456	12.1
clock_generator	$7.25e-2$	$3.27e-2$	$1.07e4$	0.105	2.8

The place-and-route (P&R) step generates a geometric realization of the gate-

Table 3.5: DV hierarchy area report.

Module	**Total area** (μm^2)	%
mote_tx_top	159646.7344	100
control_unit	41996.0469	26.3
tx_filter	9590.4141	6.0
tx_unit	26534.7246	16.6
spreading	1078.5598	0.7
differential_encoder	46.0800	0.0
header_generator	4429.4468	2.8
jtag_top	75219.4844	47.1
clock_generator	668.1599	0.4

level netlist so-called a layout [2]. FE performs silicon virtual prototyping, floor planning, power grid realization, clock tree synthesis and automated routing. It also is used for resistive-capacitive (RC) extraction, setup and hold checks, timing optimization, filler cell insertion and metal fill.

The P&R step generates several outputs: a geometric description (layout) in GDS format, a SDF description and a Verilog gate-level netlist. The SDF description at this stage includes interconnect delays. The generated Verilog netlist may be different from the one read as input to P&R because the P&R step may make further timing optimizations and buffer insertion during placement, clock tree generation and routing [2]. This gate-level Verilog netlist can be simulated by using the same Verilog testbenches (as for behavioral simulation) and the more accurate SDF data extracted from the layout. The details of the steps followed by FE are given in Appendix C.

Once the P&R of the design is complete, the tool can generate power reports for the design. Table 3.6 reports the power consumption of the chip. Note that the provided numbers include core+IO power consumption.

The details of the steps followed by Cadence for the layout preparation are given in Appendix D.

Figure 3.14 shows the silicon die photograph. Table 3.7 concludes the fabricated area specifications.

Table 3.6: FE min/max power report.

Condition	**Internal power (mW)**	**Switching power (mW)**	**Leakage power (mW)**	**Total power (mW)**
$VDD = 1.08V$, $Temp = -55C$, $DVDD = 2.3V$	27 : $IO = 16.5$, $core = 10.5$	3 : $IO = 0.14$, $core = 2.86$	1 : $IO = 0.7$, $core = 0.3$	31 : $IO = 17.5$, $core = 13.5$
$VDD = 1.65V$, $Temp = 125C$, $DVDD = 2.7V$	50 : $IO = 31$, $core = 19$	6.5 : $IO = 0.3$, $core = 6.2$	3.5 : $IO = 2.5$, $core = 1$	60 : $IO = 34$, $core = 26$

Table 3.7: Fabricated area report.

Content	**size** μm $\times$ μm
Corner pad	247×247
Core	438×1095
Core + IO pads	932×1589
Core + IO pads + boundary (submitted to CMC)	1232×1889

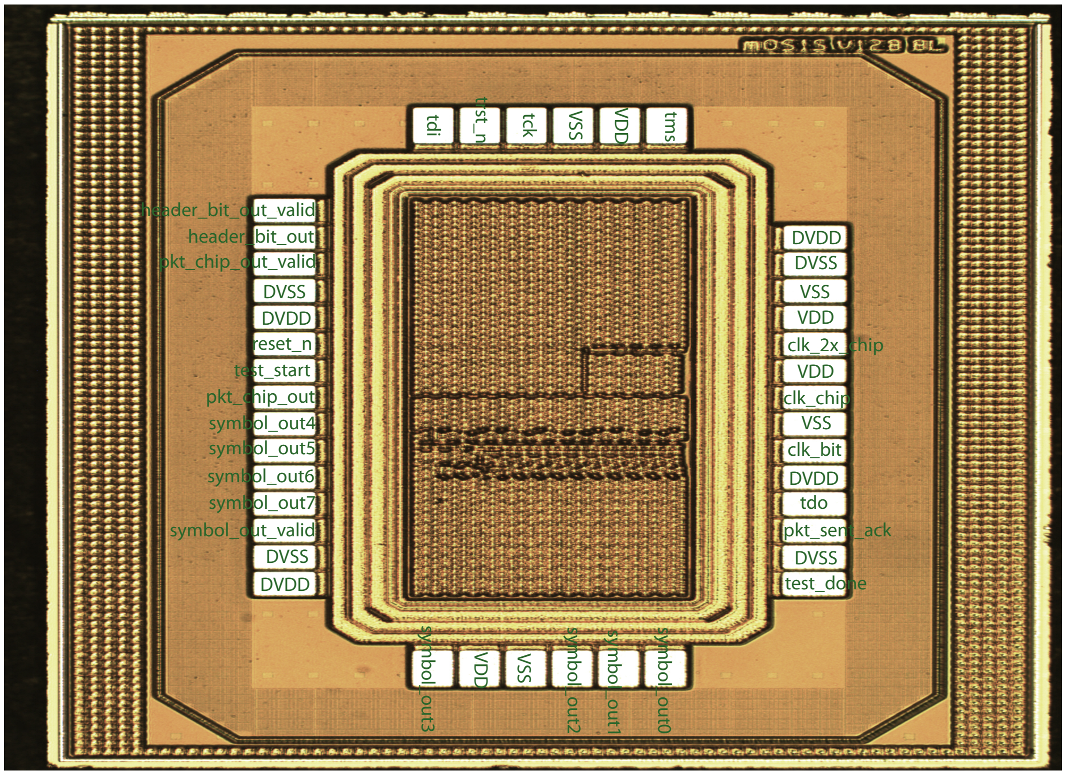


Figure 3.14: Die Photograph.

3.4.1 TX Power Efficiency

To measure the power efficiency of the core of TX (Preamble and PLH shift register + DBPSK modulator + Spreader), it was placed and routed where the target area is $420 \times 420\mu m^2$.

Table 3.8: FE min/max core power report.

Condition	**Internal+Leakage power (mW)**	**Switching power (mW)**	**Total power (mW)**
$VDD = 1.08V$, $Temp = -55C$	1.9	0.5	2.4
$VDD = 1.65V$, $Temp = 125C$	2.75	0.75	3.5

Table 3.8 shows the power estimates reported by FE. These reports were generated for `clk_chip` $= 50MHz$ (`clk_bit` $= 3.125MHz$), *input activity*=0.2, *switching activity*=0.5. The power efficiency of the TX is calculated as:

$$\begin{aligned} power\_efficiency &= \frac{power\_consumption}{rate} = \frac{3.5mJ/s}{50Mchip/s} = 70pJ/chip \\ &= \frac{3.5mJ/s}{3.125Mbit/s} = 1.12nJ/bit, \end{aligned}$$

3.5 Chip Testing

The fabricated chip was packaged using $PQFP120$ type packaging from CMC. Figure 3.15 shows the bonding diagram of the chip. The $PQFP120$ clamshell fixture from CMC was used for chip testing.

3.5.1 Test Platform

For testing of the chip, we used the XEM3010 USB-based FPGA integration board which features the Xilinx Spartan-3 FPGA, 32MB 16-bit wide SDRAM, high efficiency switching power supply, and two high-density 0.8-mm expansion connectors [3]. The USB interface of the board with PC provides fast configuration downloads and FPGA-PC communication. The on-board clock generator with three

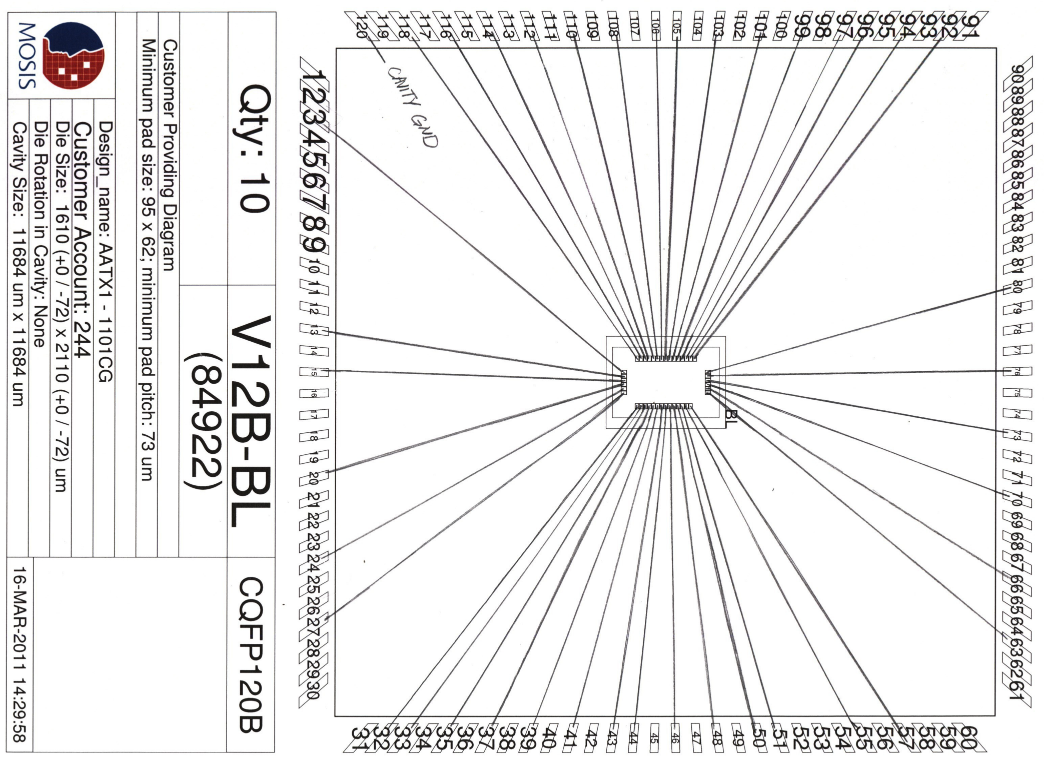

Qty: 10
V12B-BL
(84922)
CQFP120B
Customer Providing Diagram
Minimum pad size: 95 x 62; minimum pad pitch: 73 um
Design_name: AATX1 - 1101CG
Customer Account: 244
Die Size: 1610 (+0 / -72) x 2110 (+0 / -72) um
Die Rotation in Cavity: None
Cavity Size: 11684 um x 11684 um
16-MAR-2011 14:29:58
MOSIS
CAVITY GND
BL

Figure 3.15: Chip Bonding Diagram.

independent PLLs provide flexible outputs to the FPGA, SDRAM, and expansion connectors. A simple breakout board (BRK3010) is provided as an accessory to the XEM3010. This breakout board provides easy access to the high-density connectors on the XEM3010 by routing them to lower density holes. Figure 3.16 shows the picture of the XEM3010 + BRK3010 assembly on the right and the clam shell fixture from CMC on the left.

Figure 3.17 shows the functional block diagram of the XEM3010 board. From the 3.3V supply using small low-dropout(LDO) regulators, the board generates 3.3V, 2.5V and 1.2V outputs. 3.3V and 1.2V are provided to expansion devices as regulated, reliable supplies. The XEM3010 appears to the PC as a USB 2.0 plug and play peripheral. The board uses a Cypress $CY7C68013AFX2LP$ microcontroller to provide the USB interface.

A small serial EEPROM is attached to the USB microcontroller to store the boot code for the microcontroller as well as PLL configuration data and a device

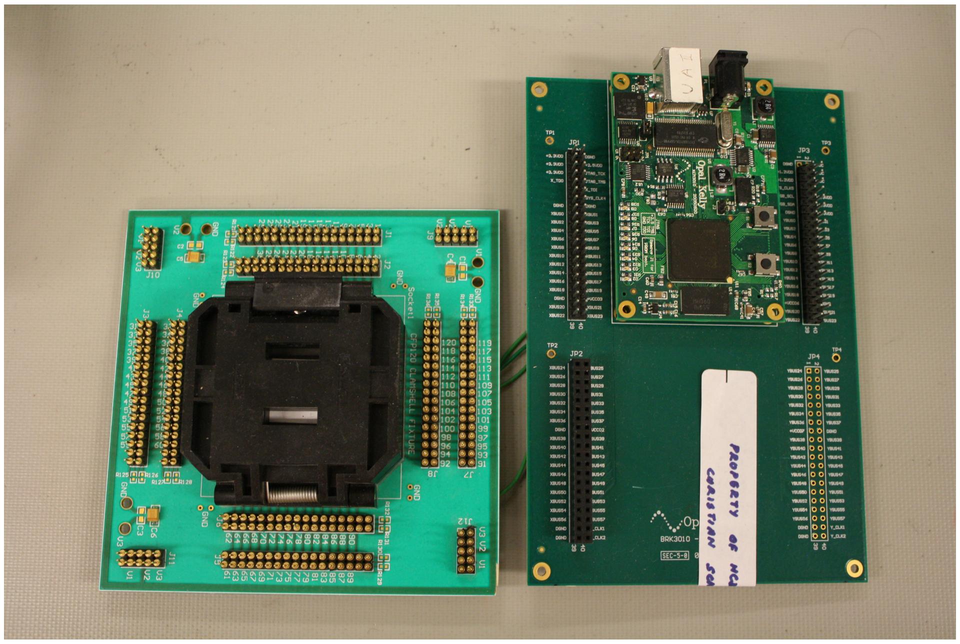


Figure 3.16: Clam Shell Fixture and the OpalKelly XEM3010 Board.

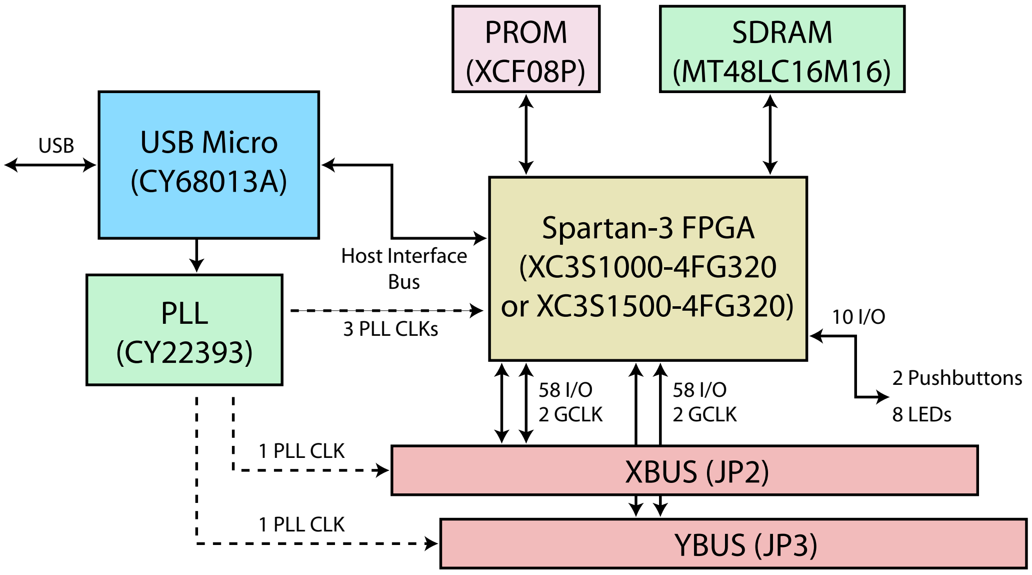


Figure 3.17: XEM3010 Block Diagram [3].

identifier string. The triple-PLL clock generator can provide up to five clocks, three to the FPGA and two to the expansion connectors JP2 and JP3. The PLL is driven by a 48-MHz signal from the USB microcontroller and can generate clocks up to 150-MHz. It is configurable through the FrontPanel API.

Eight LEDs and two pushbuttons are available for general use as debug inputs and outputs. LED anodes are connected through a pull-up resistor to 3.3V and the cathodes are connected directly to FPGA outputs. Thus to turn on an LED, the driving FPGA output pin should be brought low and to turn it off, the FPGA pin should be brought high.

The five PLL outputs are labeled `SYS_CLK1` through `SYS_CLK5`. `SYS_CLK4` connects to JP2 and `SYS_CLK5` connects to JP3. The other three pins are connected to the FPGA.

JP2 is an 80-pin high-density connector providing access to FPGA banks 2 and 3. Pin 11 on this connector is `SYS_CLK4` and the clock signal present on this pin can be configured through the PLL. JP3 is an 80-pin high-density connector providing access to FPGA bank 6 and 7. Pin 8 on this connector is `SYS_CLK5` and the clock signal present on this pin can be configured through the PLL.

The Xilinx Spartan 3 FPGA allows users to set I/O bank voltages to support several different I/O signalling standards. This functionality is supported by the XEM3010 where the user is allowed to connect independent supplies to the FPGA VCCO pins on four of the FPGA banks. By default, proper connections have been installed which attach each VCCO bank to the 3.3V supply, to change the power supply for a particular I/O bank, the appropriate connection should be removed. Since the I/O of our chip works with 2.5V (DVDD=2.5V), the VCCO of the FPGA banks connected to the chip should be connected to 2.5V.

XEM3010 is fully supported by the FrontPanel programmer's interface (API), a powerful C++ class library available to Windows programmer to interface their won software to the XEM.

The USB+FPGA+PLL+API on the XEM3010 is used to:

- Program the JTAG registers in the chip.
- Stimulate the trigger input signals and input clocks of the chip.
- Store the outputs of the chip (bit and chip sequence) into the FPGA and then transfer them to the PC.

Once the captured chip sequence is transferred to the PC through the USB and stored into a file, the correctness of the sequence is checked using a short MATLAB script.

In what follows, we briefly explain the developed FPGA code for the XEM3010 as well as the developed C++ code for the interface of the PC with the USB.

FrontPanel

FrontPanel (FP) is the software platform that provides the basic functionality required to configure and interface to the hardware including the FPGA and other peripherals on XEM board [3]. After FPGA configuration, the USB interface switches from a high-speed download port to an active communication port with FP, allowing the PC to interface and control FPGA design from within a single application. The application programmer's interface (API) of the FP provides these benefits for any custom application:

- device detection
- FPGA configuration
- FPGA communication using wires, triggers, pipes

The FP describes several components that make up the environment:

- FP HDL, which are the modules in the FPGA design that allow it to communicate with the PC.
- FP firmware, which runs on the microcontroller and provides the FPGA/PC communication.
- FP API, the programmer's interface that allowed us to design custom PC applications that communicate with the hardware.

FPGA Design

On the FPGA side of the interface, *endpoints* are used to connect FP components to the signals in our design. They work just like any external pin. We simply connect the signals that we want to control or observe to the endpoint ports. Then we connect the endpoint modules to a shared bus and place a *host interface* module on that same shared bus. The host interface; along with FP software and drivers, take care of the rest. Signals within the FPGA are immediately visible within the FP and the FP can control any input endpoints we have connected. These endpoints can be added to the FPGA design by simply instantiating them, and they consume a small amount of the available FPGA resources. The API communicates with the HDL endpoints and provides the ability to send and retrieve bulk data at the high-speed USB 2.0 data rate while the specific implementation of the USB interface disappears so that they don't get in the way of our FPGA design. In many cases, an endpoint is created from an existing signal in the FPGA design which should be observed in FP. In other cases an endpoint is created to perform a specific data transfer. The API methods are the corresponding PC-side interface to an endpoint in the FPGA.

An endpoint is either a *wire*, *trigger* or *pipe* and is directed either *in* or *out* of the design. They are always labeled from the perspective of the FPGA, so an *in* endpoint moves data into the design while and *out* endpoint moves data out of the design. All of the endpoints share a common connection to the host interface, which provides the connection to the PC through the USB on the XEM board.

Each instance of an endpoint has an associated address so it may be accessed independent of the others. Three types of endpoints with their associated address range are summarized in Table 3.9.

Table 3.9: Endpoint types.

Endpoint Type	**Address range**	**Asyn/Syn**	**Description**
WireIn	$0x00 - 0x1F$	Asyn	Signal state transfer
WireOut	$0x20 - 0x3F$	Asyn	Signal state transfer
TriggerIn	$0x40 - 0x5F$	Syn	One shot transfer
TriggerOut	$0x60 - 0x7F$	Syn	One shot transfer
PipeIn	$0x80 - 0x9F$	Syn	Multi-byte transfer
PipeOut	$0xA0 - 0xBF$	Syn	Multi-byte transfer

A short explanation of different types of endpoints is given below:

Wires: A wire is an asynchronous connection between the PC and HDL endpoint which usually is used to convey/configure the current state of some internal signal. Wires are updated periodically using a polling mechanism. In case of more than one wire in design, they are all updated at the same time. Therefore all 64 wire ins (or wire outs) are transferred together.

Triggers: Triggers are synchronous connections between the PC and the HDL endpoint. They are used to initiate or signal a single event, such as the start or end of a state machine. A trigger in creates a signal that is asserted for a single clock cycle. The synchronization clock is determined by the user. A trigger out triggers the PC when a signal's rising edge is detected.

Pipes: Pipes are synchronous connections between the PC and the HDL endpoint. Pipes are designed to transmit a series of bytes to or from the endpoint. They are most commonly used to download or upload memory contents. The PC controls the transaction for both pipe ins and pipe outs. Pipe transfer rates vary depending on host hardware and the length of the transfer.

Figures 3.18 and 3.19 show the block diagram of the FPGA design. The goal here is to stimulate the input signals of the chip and capture the output signals and evaluate the functionality of our digital chip. The API GUI + FPGA design provides the flexibility of configuring JTAG data registers as well as storing the chip output into files. The FPGA design has four major components: DCM, FP endpoint, JTAG state machine and a 256Kb dual-port FIFO.

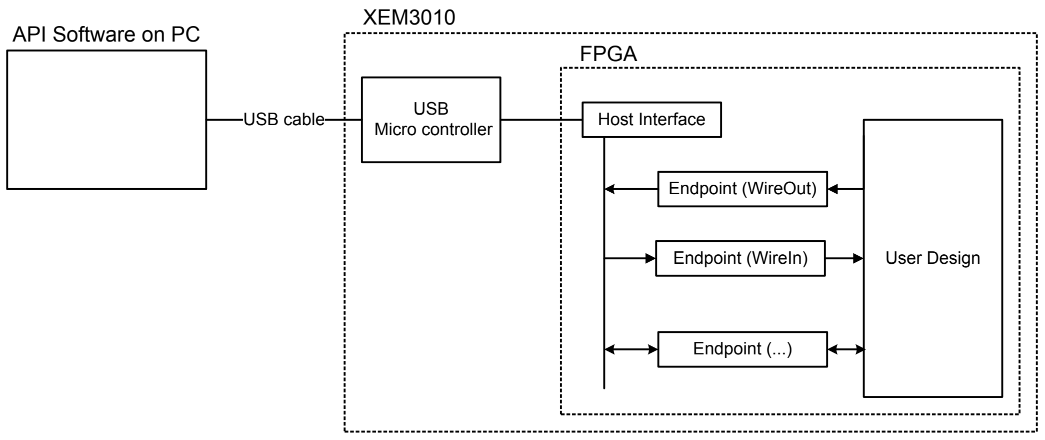


Figure 3.18: API-to-FPGA Communication.

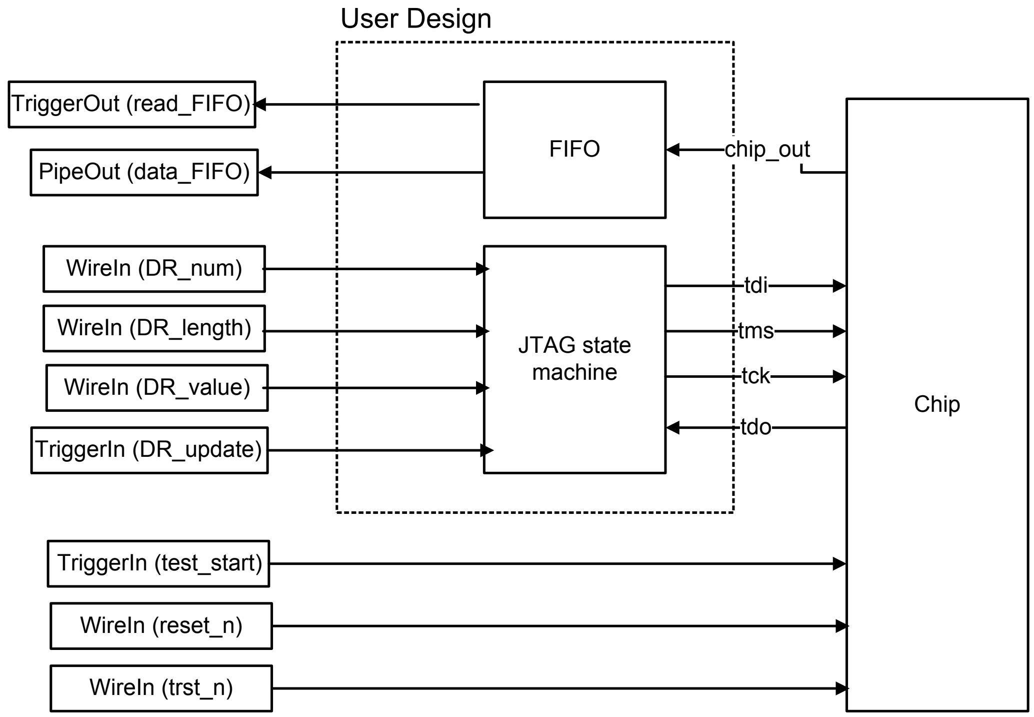


Figure 3.19: OpalKelly's FPGA Design for Chip Testing.

The dual-port FIFO is a 8-bit write, 16-bit read memory. The write clock and write data are `clk_chip` and chip sequence, respectively. The read clock is provided from the FP and the read data is connected to the PipeOut input data. The API is informed (using TriggerOut) once there are at least 2048 words written in the FIFO.

Once a JTAG data register (DR) configuration is requested (using ep41 TriggerIn), the state machine captures the DR index + DR length + DR value and generates the corresponding sequence of bits for the chip's JTAG input `tdi` and `tms` while storing the previous value of DR (while bits appear serially at chip JTAG output `tdo`) at the same time. Once the configuration is done, the previous values are reported to the API.

The list of the FP endpoints instantiated in our design and the purpose of each endpoint is given below:

- ep00 (WireIn): ep00[0] is dedicated to stimulate `reset_n`, ep00[1] is for `trst_n`, ep00[5:2] is to assign the index of the JTAG data register to be configured, ep00[15:6] is to assign the length of the JTAG data register to be configured.

- ep01 (WireIn): ep01[15:0] stores bits 15 to 0 of the value of the JTAG data register to be configured.
- ep02 (WireIn): ep02[15:0] stores bits 31 to 16 of the value of the JTAG data register to be configured.
- ep03 (WireIn): ep03[15:0] stores bits 47 to 32 of the value of the JTAG data register to be configured.
- ep04 (WireIn): ep04[15:0] stores bits 63 to 48 of the value of the JTAG data register to be configured.
- ep20 (WireOut): ep20[13:0] reports the number of the available words in the FIFO that store the chip/bit sequence.
- ep21 (WireOut): ep21[15:0] stores the previous value of the IR once a new value is configured.
- ep22 (WireOut): ep22[15:0] stores the MSB word of the previous value of a data register in JTAG once a new value is configured.
- ep40 (TriggerIn): ep40[0] is used to activate (assert high) the `test_start` chip input for one cycle of clock `clk_2x_chip`.
- ep41 (TriggerIn): ep41[0] is used to enable the state machine sequence we have implemented in our FPGA design to configure a JTAG data register in the chip.
- ep60 (TriggerOut): ep60[0] reports to the API that the `data_count` of the FIFO that stores the chip/bit sequence is passed a specific threshold. ep60[1] reports to the API that the configuration of the JTAG DR is completed.
- epa0 (PipeOut): the `read_data` of the FIFO that stores the chip/bit sequence is connected to the input data of this Pipeout endpoint.

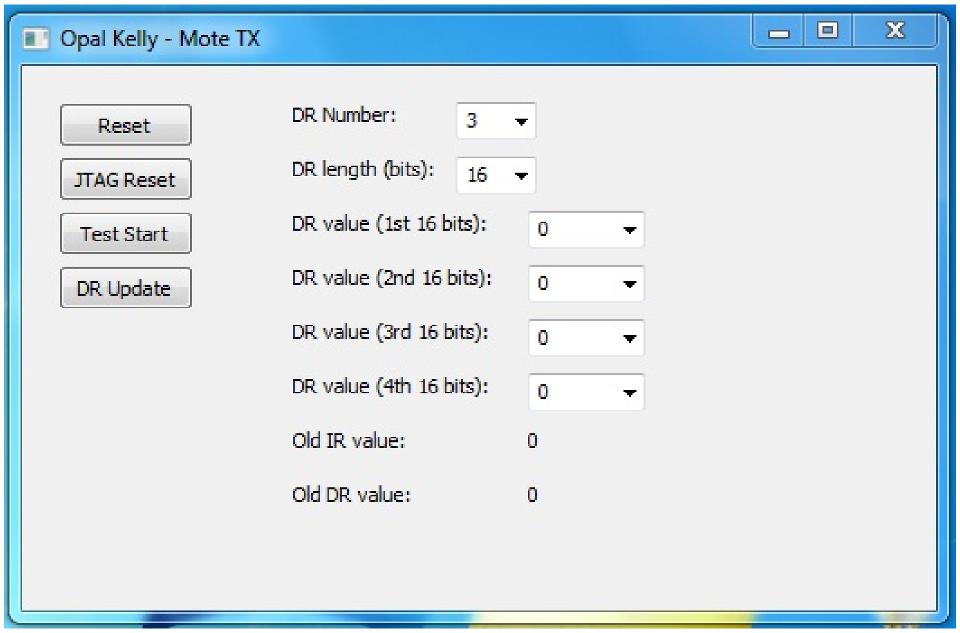
FrontPanel API

The FP API contains methods which communicate via the USB, but they have been specifically designed to interface with the FPGA. The API's library is written in C++ and is provided as a dynamically-linked library (DLL). The API reference guide can be found online [43]. A brief description of the methods that we used for our design is provided below:

- *OpenBySerial*: Open the first available XEM device.
- *LoadDefaultPLLConfiguration* : Configure the PLL using the stored EEPROM settings.
- *ConfigureFPGA*: Download a configuration bit file to the FPGA.
- *SetWireInValue (int epAddr, UINT32 val, UINT32 mask = 0xffffffff)*: Set the *val* value according to the *mask* value on the WireIn endpoint with address *epAddr*. Requires a subsequent call to UpdateWireIns.
- *UpdateWireIns*: Update all WireIn values (to FPGA) simultaneously with the values held internally to the API.
- *ActivateTriggerIn(int epAddr, int bit)*: Activate *bit* of TriggerIn endpoint with address *epAddr*.
- *UpdateTriggerOuts*: Used to retrieve all TriggerOut values (from the FPGA) and records which endpoints have triggered since the last query.
- *ReadFromPipeOut(int epAddr, long length, unsigned char *data)*: Transfer data (byte array) of length *length* from PipeOut endpoint with address *epAddr* to the *data*.
- *UpdateWireOuts*: Simultaneously retrieves all WireOut values (from FPGA) and stores the values internally.
- *GetWireOutValue(int epAddr)*: Read the result from the WireOut endpoint with address *epAddr*.

- *IsTriggered (int epAddr, UINT32 mask)*: Return true if the TriggerOut endpoint with address *epAddr* and bit according to the *mask* has been triggered since a previous call to UpdateTriggerOuts.

We used Visual C++ to develop our custom API for the testing of the chip. Figure 3.20 shows the GUI of the developed API. The developed API has three major jobs: configuring the FPGA, configuring the JTAG data registers, and storing the bit/chip sequence into a file.


Opal Kelly - Mote TX

DR Number: 3
DR length (bits): 16
DR value (1st 16 bits): 0
DR value (2nd 16 bits): 0
DR value (3rd 16 bits): 0
DR value (4th 16 bits): 0
Old IR value: 0
Old DR value: 0

Figure 3.20: API GUI.

Once we run the developed code, it first asks for the bit file to be used to configure the FPGA. The GUI then lets the user configure one JTAG DR by setting the `DR_Num`, `DR_Length`, `DR_Value`. Once the `test_start` chip input is asserted, and the FIFO has passed its threshold, the API reads the number of available words to be read, and appends them to a file using the appropriate function in C++. The `test_done` signal is connected to an LED on XEM to acknowledge the termination of the test.

Results of Chip Testing

As mentioned above, the different functions of the design were tested by capturing and verifying the chip's operating sequence. Here is the list of tested functions and the result of the testing:

- The chip sequence using the default values of the JTAG data registers was captured and verified. The captured chip sequence matched the expected sequence. For example PLH sequence was verified to count up from 0 to $2^{16} - 1$.

- The default value of the spreading sequence was overwritten using the JTAG and the captured chip sequence matched the expected sequence.

- The `mux_sel` was overwritten and the captured chip sequence matched the expected sequence.

- The default value of `payload_chip_sequence` was overwritten and the captured chip sequence matched the expected sequence.

- The default value of `total_packet_number` was overwritten and the captured chip sequence matched the expected sequence.

- The default value of `preamble_sequence` was overwritten and the captured chip sequence matched the expected sequence.

- The default value of `preamble_spreading_sequence` was overwritten and the captured chip sequence matched the expected sequence.

- The default value of `plh_sequence` was overwritten and the captured chip sequence matched the expected sequence.

- The default value of `inter_packet_spacing` was overwritten but the captured chip sequence didn't match the expected sequence. We found the source of the problem in the RTL code of the chip. Then we fixed the RTL code and verified the corrected design on the FPGA.

Power Efficiency of the Fabricated Design

The power consumption of the core (not including the I/O pads) of the fabricated chip was measured to be $600\mu W$ where $clkchip = 12.5MHz$. Thus its power efficiency is as follows:

$$\begin{aligned} power\_efficiency &= \frac{power\_consumption}{rate} = \frac{600\mu J/s}{12.5Mchip/s} = 48pJ/chip \\ &= 768pJ/bit. \end{aligned}$$

The power efficiency of the RF designs [25],[26] (presented in Section 1.3) for implantable medical devices is in the range of $20pJ/chip$ up to $500pJ/bit$. Thus the expected power efficiency of the baseband TX design is in the same order or lower than the RF transmitter. As shown in [44], the use of low-voltage technologies for fabrication (subthreshold silicon implementation) can provide a power saving of more than five times, thus subthreshold implementation of this design for low-power applications such as implantable devices is suggested.

Design Alternatives for Testing

The silicon area for fabrication awarded to us by CMC Microsystem was $1.5\mu m \times 2\mu m$. Thus the number of available pins (including power pins) was very limited. A parallel bus for testing (for example a parallel microprocessor interface) would provide a higher rate for interfacing with the core design, but it would also require more pins comparing to a serial bus. A serial bus has many advantages such as smaller physical interface, simplified design and thus lower power consumption comparing to a parallel bus. The disadvantage of the serial bus is the lower speed of the interface, which in our design is not a high priority. Therefore, a serial bus is more suitable for this design.

Among the different standards for serial buses such as I2C, SPI and JTAG, JTAG was selected for our design because it is a low-complexity, high data rate standard which is used in many devices for configuration. It also provides the possibility of using a daisy chain configuration to link up multiple devices.

The test platform (XEM3010 OpalKelly board) that was utilized for the chip testing was easy to use (plug and play), with a user-friendly GUI and a short learning curve. The transmitter chip is to eventually interface with analog and RF counterparts. Thus testing this chip with a stand-alone and small FPGA board (instead of a digital tester) was preferred. This same platform could also be used for stimulation of the transmitter chip when it is tested in the full transceiver link.

Note that critical internal signals such as `clk_chip`, `clk_bit`, `header_bit_out` and `header_bit_out_valid` were mapped to output pins to provide better testability for internal modules of the chip.

Chapter 4

Receiver Design

In Chapter 2, we provided the theory and the model of the transceiver design. Chapter 3 detailed the fabricated TX design. In this chapter, we aim to provide the details of the implementation of the RX design. Section 4.1 of this chapter explains the FPGA implementation of the design. The testing of the design is described in Section 4.2.

4.1 FPGA Implementation of the RX Design

As was explained in Chapter 2, the receiver is an asynchronous, correlation-based packet detector. For that reason, the receiver must continuously monitor the channel for the arrival of the preamble. Thus a low-power preamble detector, which shows good performance even with the low resolution of the input symbol stream, is desirable. Once the preamble is detected, the receiver could switch to a higher bandwidth modulation scheme for the payload symbols and could utilize enhanced schemes (e.g. powerful error correcting codes) during the extraction of payload.

Figure 4.1 shows the block diagram of the implemented baseband RX design. As shown in the figure, the design also includes a channel emulator to mimic the effects of the channel (frequency offset and Gaussian white noise) so that the performance of the baseband design can be evaluated under controlled values of noise and frequency offset. It is shown in [11] that the non-coherent detector with two samples per chip

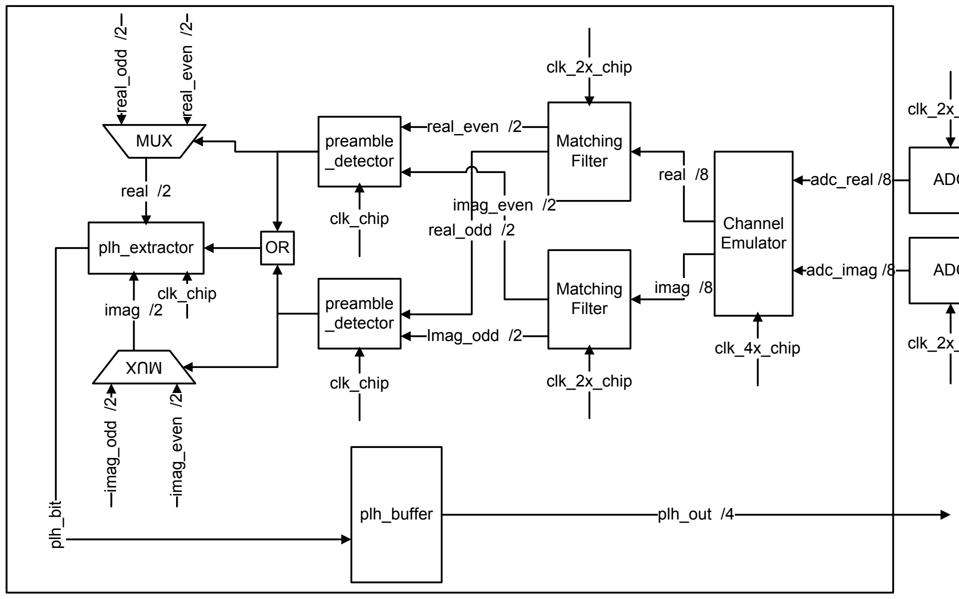


Figure 4.1: RX Architecture.

performs essentially identically to a coherent detector. Thus the ADC's sampling rate is twice the chip rate. These samples are sent over two parallel sections that perform preamble detection.

Figure 4.2 shows the modules within the mote RX and their interface with each other and the input/output pins. A brief description of each module's input/output and functionality is given below.

4.1.1 JTAG Module

The design and functionality of this module is essentially the same as what is given in Section 4.1.1. The list of implemented data registers in our RX design is given in Table 4.1. For each data register, the width and initial default register contents are given.

A brief description of each data register is given below. For each data register, the default value is used unless it is overwritten through the JTAG port.

`preamble_sequence` This register configures the pre-known preamble sequence.

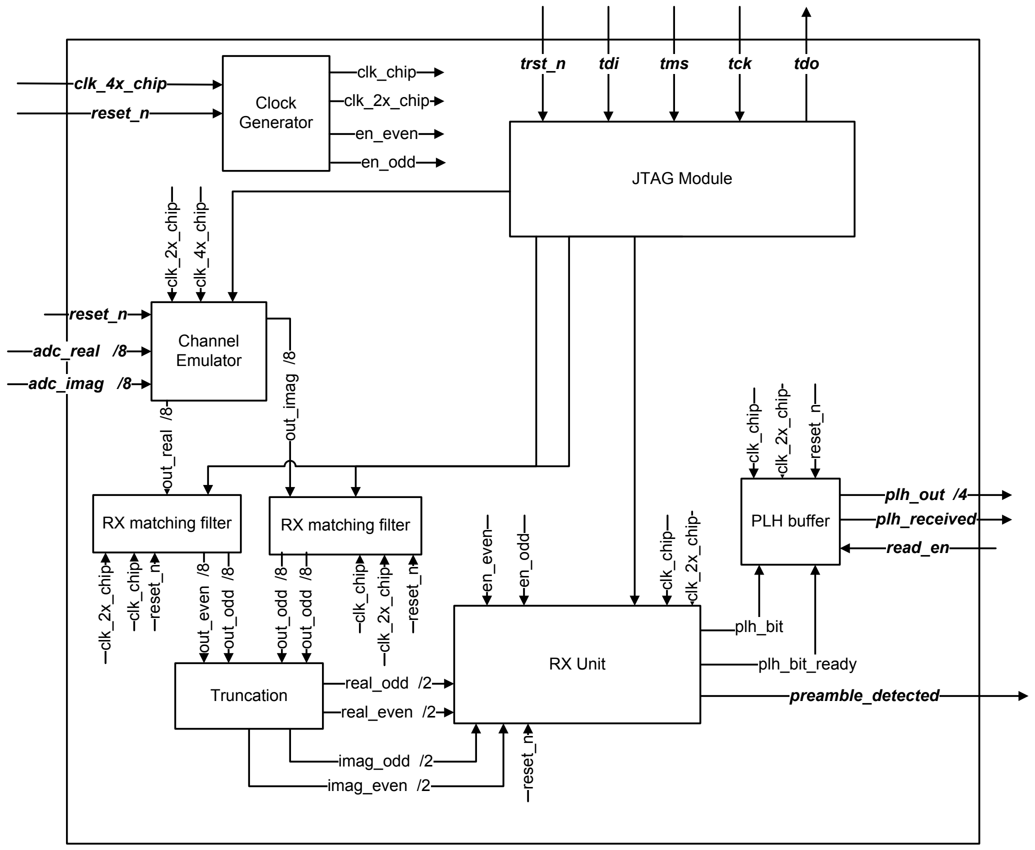


Figure 4.2: RX Modules.

Table 4.1: RX JTAG Data Register List.

Register Name	**Width**	**Initial (default) value**	**R/W**
`preamble_sequence`	40	40'h2481F1539C	RW
`preamble_spreading_sequence`	16	16'h066B	RW
`correlator_threshold`	8	8'h28	RW
`frequency_offset`	12	12'h000	RW
`rx_filter_coeff`	96	{8'h00,8'hFF,8'h04,8'hF9, 8'hF6,8'h34,8'h66,8'h34, 8'hF6,8'hF9,8'h04,8'hFF}	RW
`mux_sel`	3	3'h0	RW
`false_alarm_counter`	16	16'h0	R
`miss_alarm_counter`	16	16'h0	R

`preamble_spreading_sequence` This register configures the spreading sequence used for preamble and PLH.

`rx_filter_coeff` Coefficients of the 12-tap root-raised-cosine pulse shaping filter in the format of {`coeff_12`, `coeff_11`, `coeff_10`,...,`coeff_1`}.

`mux_sel` This register configures the select line of some muxes.

`mux_sel[0]`: `rx_filter_en`, if this bit is 1, RX matching filter is enabled; otherwise it is bypassed.

`mux_sel[1]`: `awgn_en`, if this bit is 1, noise generator is enabled; otherwise it is disabled.

`mux_sel[2]`: `freq_offset_en`, if this bit is 1, frequency offset generator is enabled; otherwise it is disabled.

`correlator_threshold` This register stores the correlator threshold.

`frequency_offset` This register stores the frequency offset.

`missed_packet_counter` This register stores the number of missed packets.

`false_alarm_counter` This register stores the number of false alarms.

The JTAG data registers have the default values after a reset (`trst_n`). Thus, they can always be used directly for the test measurement. Individual `DR`s can have their default values overwritten through the JTAG port as required.

4.1.2 Clock Generator (DCM)

In Figure 4.2, the clock source for each sub-module is shown. Since the noise generator module generates both real and imaginary noise symbols, each to be generated at (`clk_2x_chip`) rate, thus this module is clocked with `clk_4x_chip`. Clock generator module receives `clock_4x_chip` as the reference clock and derives `clk_chip` and `clk_2x_chip`. `en_even` and `en_odd` outputs are the indicators of odd and even edges of the `clk_2x_chip` regarding to `clk_chip`.

`reset_n` is the active low asynchronous input to this module. Clocks generated by this module are in-phase with the `clock_4x_chip` input clock. In case of an implementation on an FPGA, this module should be replaced with a DCM.

4.1.3 Channel Emulator

Figure 4.3 shows the block diagram of the implemented channel emulator module. The performance of the baseband receiver (preamble detector) should be evaluated under a controlled amount of noise and frequency offset. The channel is assumed to be corrupted with AWGN and thus the channel emulator is used to introduce the effects of the Gaussian noise and frequency offset to the design. It consists of two main submodules: `noise_generator` and `symbol_rotator`. Each of these submodule could be disabled or enabled according to `mux_sel[1]` and `mux_sel[2]`, respectively. Once disabled, they are bypassed and have no effect on their input symbol stream. The input symbols to the channel emulator module are the 8-bit two's complement (4-bit integer and 4-bit fraction) symbols from the ADC sampled by `clk_2x_chip`.

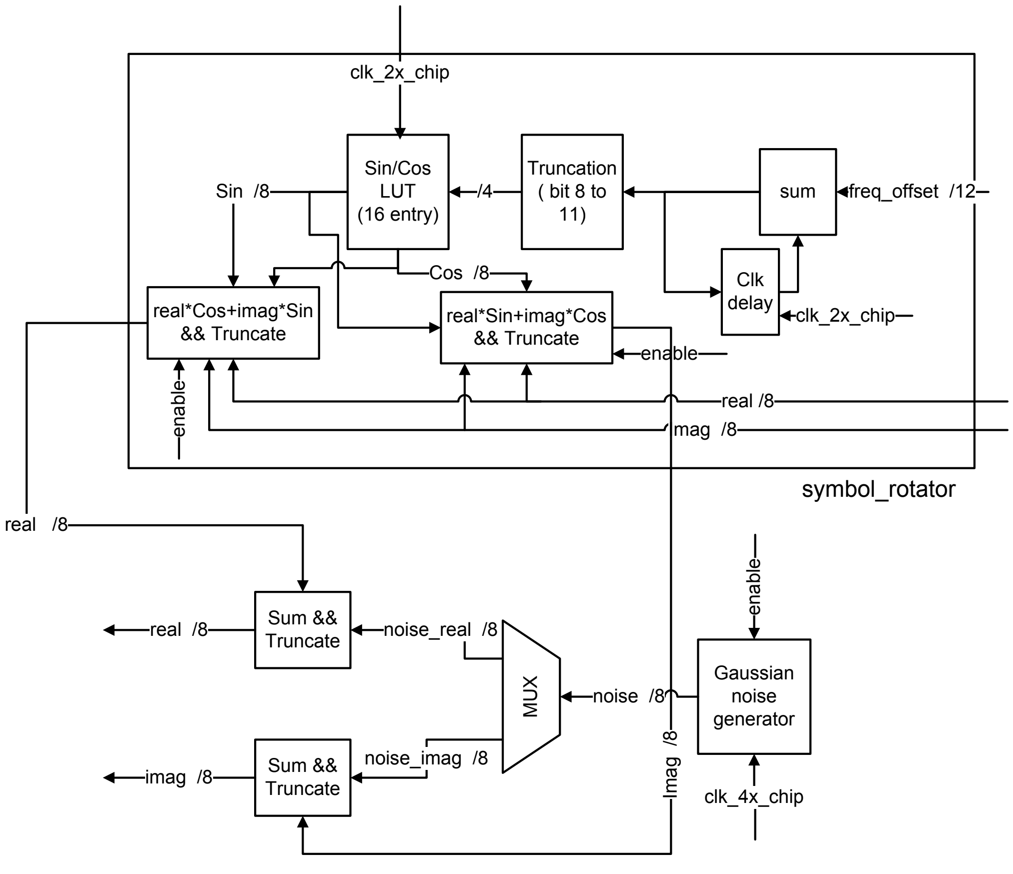


Figure 4.3: Channel Emulator.

4.1.4 Noise Generator

The noise generator module was provided to us from Ukalta Engineering (Edmonton, AB). The key features of this IP core are:

- It generates Gaussian-distributed samples that match the probability density function of the standard normal distribution (i.e., zero mean and unit variance) within ± 1 percent up to $\pm 3.1\sigma$.
- It produces uncorrelated Gaussian samples with a flat (white) power spectral density.
- Ultra-long repetition period of the output samples.

The $UGNG - 31$ core generates one 8-bit sample every clock period when the clock enable pin is held high. The generated samples are represented in two's-complement fixed-point format with a 4-bit integer and 4-bit fraction. Asserting the reset signal clears internal registers of the UGNG-31 and returns the core to its initial state. Figure 4.4 shows the distribution of the generated samples captured (using ChipScope) once this core was implemented in the FPGA.

4.1.5 Symbol Rotator

According to the equation 2.3, the symbol rotator should generate $e^{j(2\pi\triangle fkT_c+\phi)}$ samples and multiply the complex input symbols with them. The value of $\triangle fT_c$ is a 12-bit input to this module which is used to generate the corresponding phase. The sin/cos samples of the signals are stored in a 16-entry LUT in a two's-complement fixed-point format with 4-bit integer and 4-bit fraction. Table 4.2 shows the stored values in the LUT.

The complex input symbols to this module (in two's-complement format with a 4-bit integer and 4-bit fraction) are complex-multiplied with the sin/cos symbols and the output is registered out in two's-complement format with a 4-bit integer and 4-bit fraction.

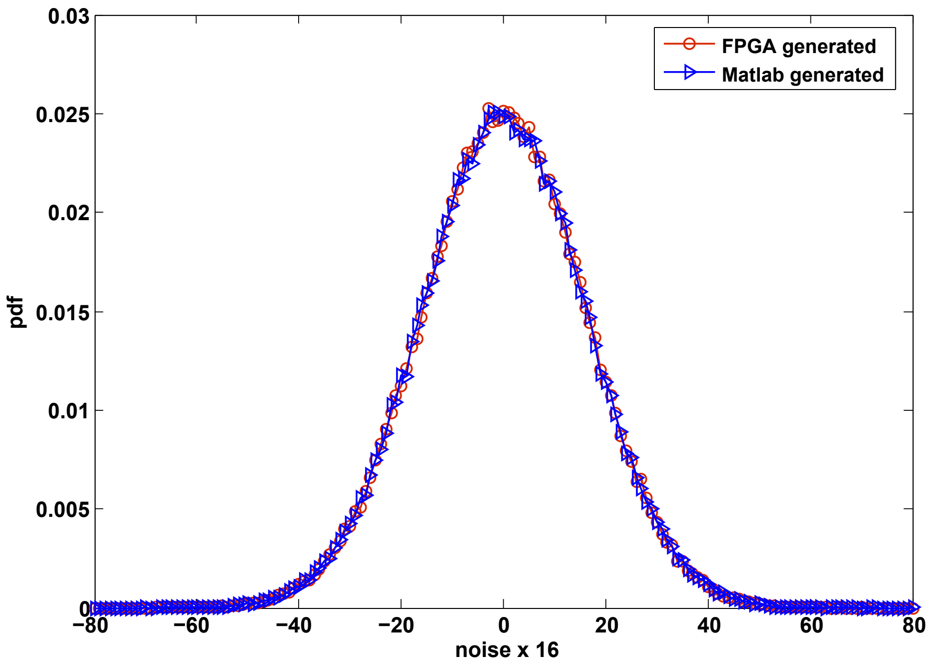


Figure 4.4: PDF of the Hardware and Software-based Gaussian Noise Generators.

Table 4.2: Sin/Cos LUT.

Phase	**Sin**	**Cos**
$2\pi(0)$	$8'h00$	$8'h10$
$2\pi(\frac{1}{16})$	$8'h06$	$8'h0F$
$2\pi(\frac{2}{16})$	$8'h0B$	$8'h0B$
$2\pi(\frac{3}{16})$	$8'h0F$	$8'h06$
$2\pi(\frac{4}{16})$	$8'h10$	$8'h00$
$2\pi(\frac{5}{16})$	$8'h0F$	$8'hFA$
$2\pi(\frac{6}{16})$	$8'h0B$	$8'hF5$
$2\pi(\frac{7}{16})$	$8'h06$	$8'hF1$
$2\pi(\frac{8}{16})$	$8'h00$	$8'hF0$
$2\pi(\frac{9}{16})$	$8'hFA$	$8'hF1$
$2\pi(\frac{10}{16})$	$8'hF5$	$8'hF5$
$2\pi(\frac{11}{16})$	$8'hF1$	$8'hFA$
$2\pi(\frac{12}{16})$	$8'hF0$	$8'h00$
$2\pi(\frac{13}{16})$	$8'hF1$	$8'h06$
$2\pi(\frac{14}{16})$	$8'hF5$	$8'h0B$
$2\pi(\frac{15}{16})$	$8'hFA$	$8'h0F$

4.1.6 Matching Filter

Once the effect of the channel is applied to the received symbols, the symbols are filtered by the matching filter. Thus, the 8-bit complex symbols out of the channel emulator are routed into the matching filter. The filter coefficients are retrieved from the `rx_filter_coeff` data register. This filter is instantiated twice, once for the real and once for the imaginary symbols. This module generates separate even and odd output symbols.

4.1.7 RX Unit

Figure 4.5 shows the block diagram of the RX unit submodule. As the figure shows, `preamble_detector` and `plh_extractor` are the two submodules of the RX unit. Since a low-resolution, low-complexity preamble detector is desirable, in this thesis we always implemented the RX unit assuming that its input symbols are only 2 bits wide [12]. Thus the output of the matching filter is first truncated to 2 bits (mapped to ± 1 based on the sign of the symbol). The detection algorithm runs in parallel both on even and odd symbols. If one of the two detection paths successfully detected a preamble, the PLH extraction module is activated and the corresponding stream of symbols is routed to the PLH extraction module. The following is the description of these two submodules:

Preamble Detector

Figure 4.6 shows the components of the preamble detection algorithm assuming that the spreading sequence is 16 chips per bit and the preamble sequence is 40 bits. Since this detection algorithm is for an asynchronous communication scheme, a new instance of the algorithm calculation should be started every chip interval. In other words, it should look at every input chip as potentially being the beginning of a preamble. The four major steps to detect a preamble are:

- Despreading of the complex chips to form the complex symbols ($\hat{a}_m(k)$): every 16 consecutive chips should be despreaded using the pre-known spreading sequence. As explained in [4], the despreading is equivalent to a series of 16 modified additions/subtractions. A new despreading calculation should start every `clk_chip`. The required series of modified additions/subtractions

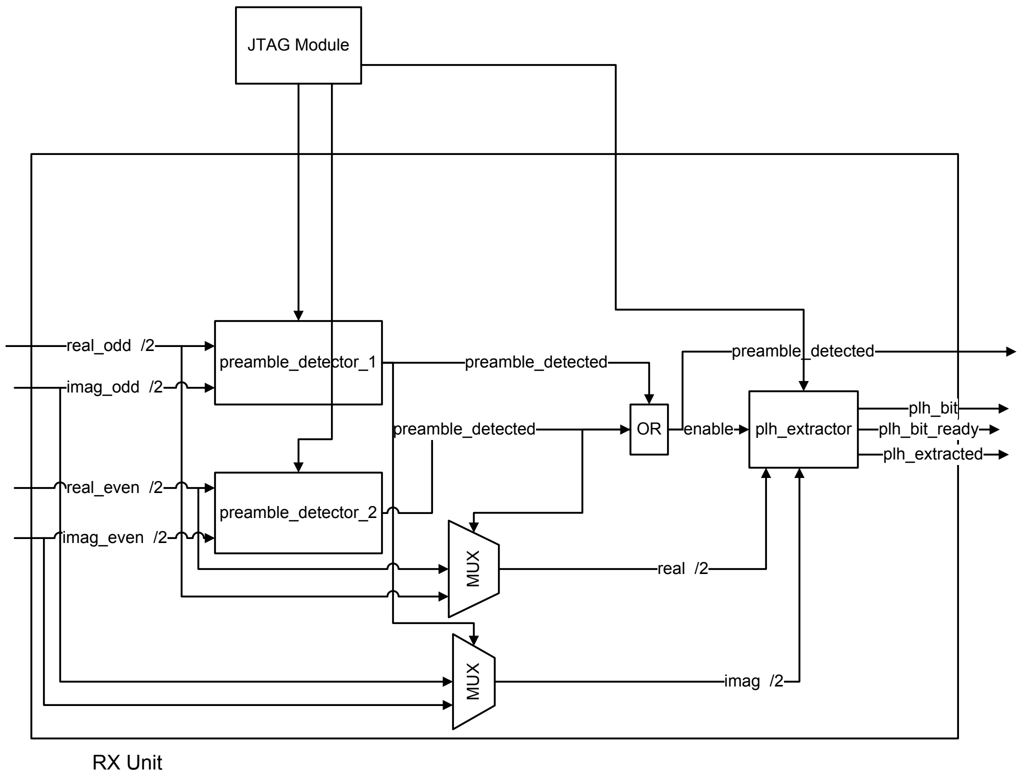


Figure 4.5: Architecture of the RX Unit.

is pipelined to increase the throughput of the detector. Figure 4.7 shows the adder tree implemented to perform the despreading. To decrease the delay of this computation, the adder tree is clocked with `clk_2x_chip`.

- Differentially decoding the complex symbols to form the bits: Since differential encoding of the bits is done at the transmitter, the receiver needs to perform differential decoding. Differential decoding of consecutive complex symbols requires performing the complex multiplication of $\hat{a}_m(k)\hat{a}^*_{m-1}(k)$. Since the resulting sequence should be cross-correlated to the c_m sequence, which only has real components, computing only the real part of $\hat{a}_m(k)\hat{a}^*_{m-1}(k)$ is enough.

- Calculating the cross correlation of the detected bit sequence with the pre-known preamble bit sequence to form the correlation statistic: The cross-correlation of the detected bit sequence and the preamble sequence, $\eta_k = \sum_{m=2}^{W} \hat{a}_m(k)\hat{a}^*_{m-1}(k)c_m$, is the decision metric for preamble detection. As

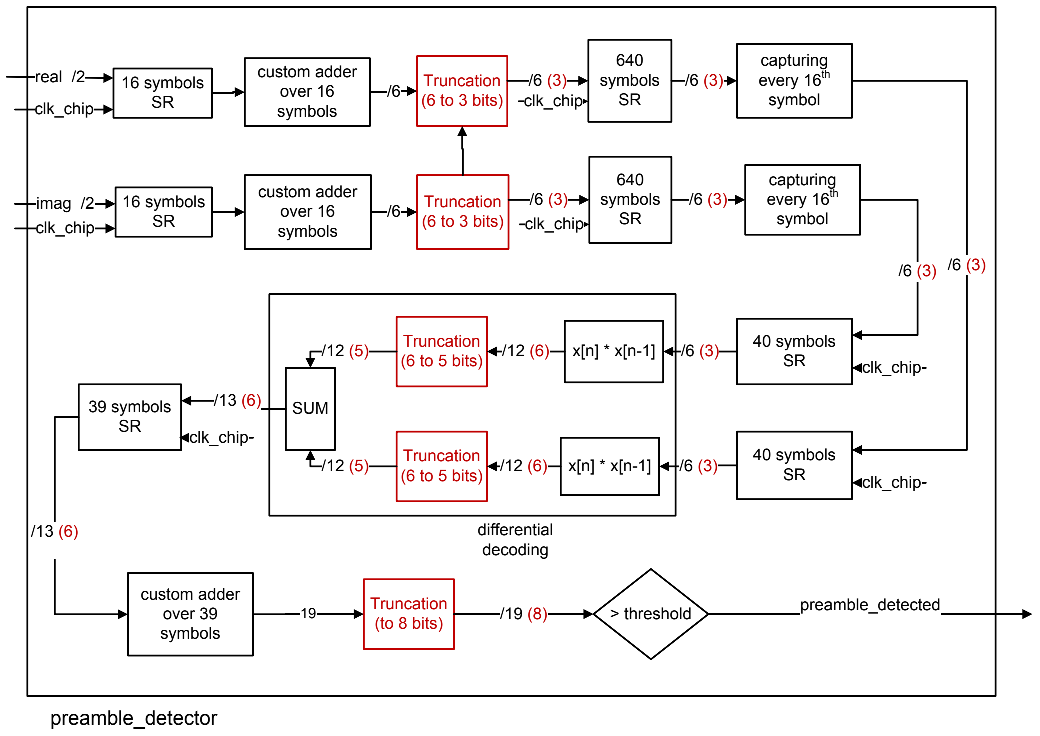


Figure 4.6: Preamble Detector.

explained in [4], this computation is equivalent to a series of 39 modified additions/subtractions (assuming an adder tree). A new calculation should start every `clk_chip` cycle, thus the series of modified additions/subtractions is pipelined to increase the throughput of the detector. To decrease the delay of this computation, the adder tree is clocked with `clk_2x_chip`.

- Comparing the correlation statistic with the pre-defined threshold to decide whether the preamble is detected: If the correlation statistic is greater than threshold G, (if $Re(\eta_k) > G$), the `preamble_detected` flag goes high and the PLH extraction module is enabled.

To enhance the preamble detector design in terms of area, speed and power, we consider truncation and rounding in different steps of the adder tree for despreading and also the adder tree to calculate the preamble cross-correlation. In Figure 4.6, the width of the input and output of the components of the preamble detector module is shown. The numbers in red show the width of the input and output of

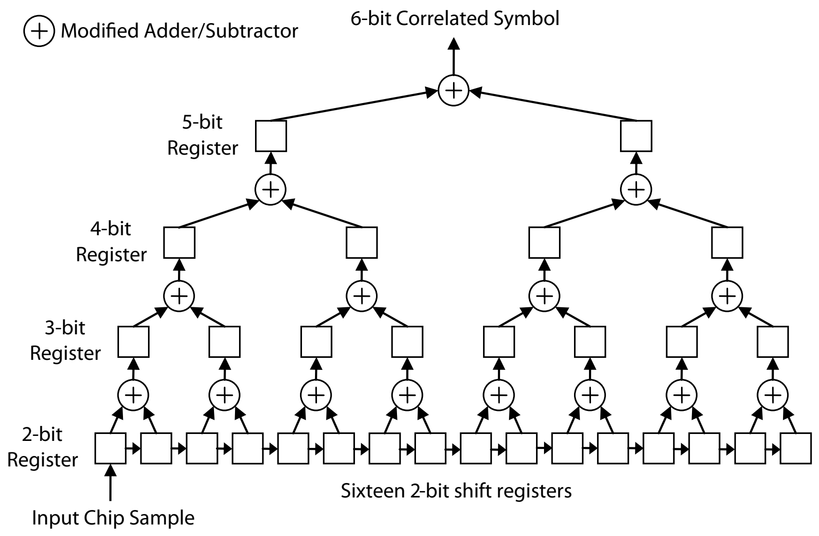


Figure 4.7: Adder Tree [4].

those components for the truncated detector. The most area consuming module is the multiplication module, thus the truncation of the input samples to this module saves significant amount of area. According to Table 2.1, for the pSNR in range of interest, cSNR is always negative. That means the power of the samples at chip level is lower than the power of noise. Thus a truncation after the despreading module (custom adder over 16 symbols) is appropriate. The amount of truncation was defined empirically based on the results of the simulation. The goal was to reduce the size of the logic without degrading the performance more than 1 dB.

The performance of this truncated detector is compared with that of the non-truncated detector in Section 4.2. Also the comparison of the speed and area of the two is given in Section 4.2.

PLH Extractor

Once a preamble is detected (`preamble_detected` flag goes high), the PLH extractor module gets activated. Figure 4.8 shows the components of the PLH extractor module assuming the spreading sequence is 16 chips per bit. Basically the PLH extractor performs the the despreading of the chips to form symbols, and then the differential decoding of the symbols to form bits. The details of these two functions are explained in Section 4.1.7. The result is truncated to one bit (the sign bit) and stored as the PLH bit. Once the 64 bits sequence of PLH have been extracted, the `plh_received` flag goes high.

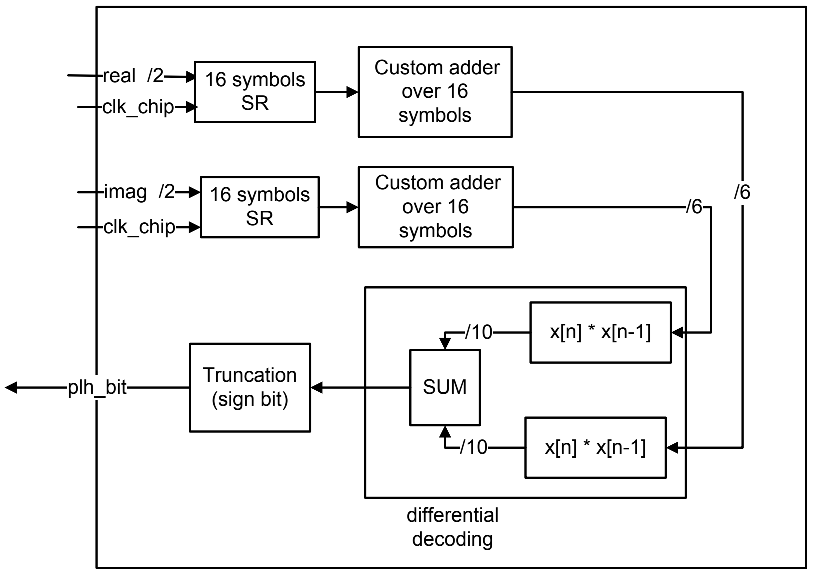


Figure 4.8: PLH Extractor.

4.1.8 PLH Buffer

This buffer stores the extracted PLH sequence. This sequence could be downloaded out in parallel through `plh_out` (4 bits) every time `read_en` goes high for one cycle of `clk_2x_chip`.

4.2 Design Testing

In order to verify the asynchronous transceiver design at hardware speeds, we used the FPGA boards designed by Micorolynx Systems Ltd., as shown in Figure 4.9.

The transmitter board includes Xilinx Spartan $3e$ FPGA ($XC3S1200e{-}4fg320$). The baseband digital design was implemented inside this FPGA. The board also includes two DACs for sampling the in-phase and quadrature symbols generated by the FPGA. The RF modulator for the ISM band as well as embedded antenna are available for the transmitter board. The crystal on the board generates a $16MHz$

source clock which is used by the on-board PLL to generate the clocks for the FPGA and the DACs. The PLL is programmable via the FPGA. The details of the Microlynx Systems boards are given in their datasheet [45].

The receiver board also includes Xilinx Spartan 3e FPGA ($XC3S1200e{-}4fg320$) as well as two ADCs for in-phase and quadrature symbols. The RF demodulator for the ISM band as well as embedded antenna are available on receiver board. The crystal on the board generates a 16MHz source clock which is used by the FPGA to generate the sampling clock for the ADCs. The details of the Micorolynx Systems boards are given in their datasheet [45].

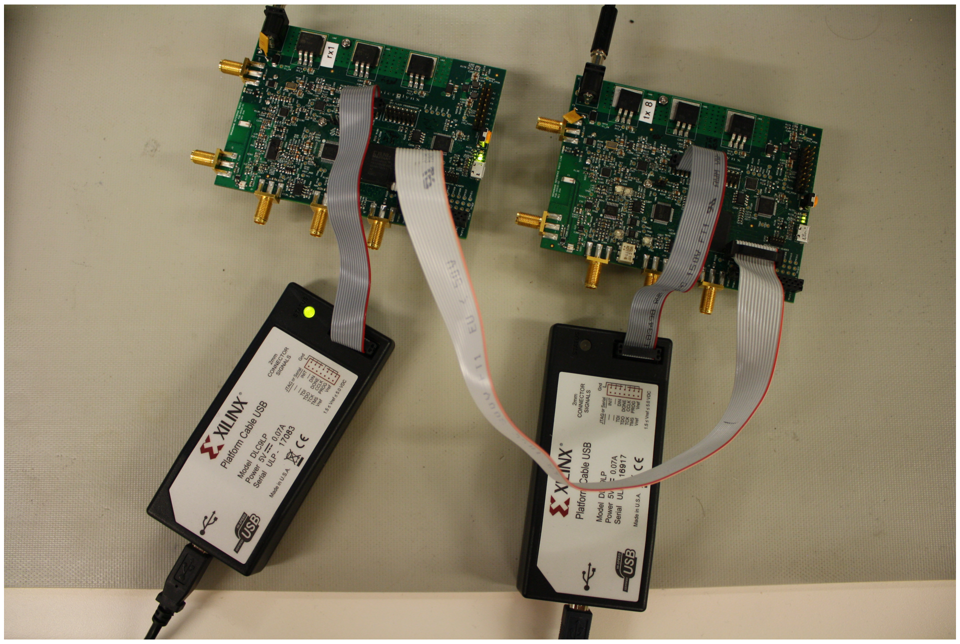


Figure 4.9: Microlynx TX and RX Custom FPGA Boards.

As a way of verifying the design, we used the transmitter and receiver designs to measure the miss rate and false alarm rate of the preamble detector for pSNR values varying from 15dB to 25dB. Conditioned on the presence of the preamble at moment i_0, two error events are defined as follows: First, the decision statistic of greater than the threshold G, at moment $k \neq i_0$, determines the erroneous presence of a preamble. Such an event is called a *false alarm* here and its probability is denoted by P_{false}. Second, if the correlation statistic at moment i_0 is less than the threshold G, a preamble is missed. Such an event is called a *miss* and its probability is denoted by P_{miss}.

The transmitter and receiver baseband designs were implemented on separate Microlynx Systems transmitter and receiver boards with clocks that were nominally the same but not synchronized. Thus this FPGA measurement evaluates the performance of the preamble detector in presence of actual time offset between transmitter and receiver. For the sake of these measurements, the RF link was bypassed and instead channel emulator module was included in the receiver design where it introduces controlled amounts of noise and frequency offset. Thus the digital baseband connection between the transmitter and receiver was made through the GPIO pins available on the boards, as shown in Figure 4.9.

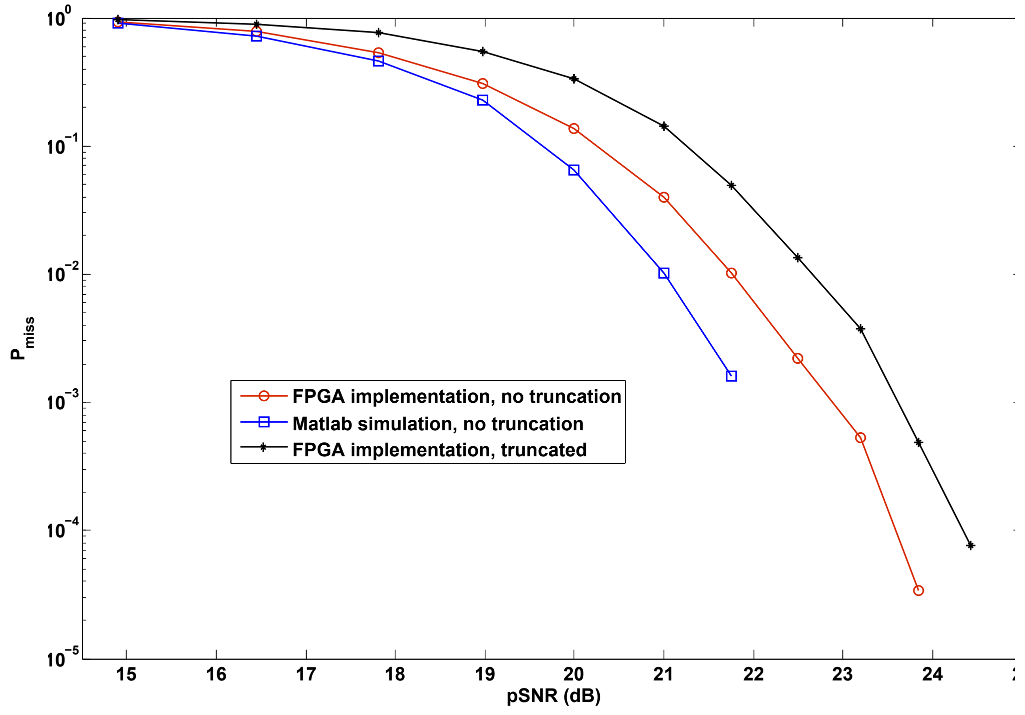


Figure 4.10: FPGA Implementation, P_{miss} vs. pSNR for $P_{false} = 10^{-3}$.

Figure 4.10 shows the probability of missing the preamble versus pSNR when $P_{false} = 0.001$. To keep the $P_{false} = 0.001$, the threshold value G has to change accordingly for different values of pSNR. The spreading sequence and preamble sequence are as given in Table 2.2 and no frequency offset is injected by the channel for this measurement. The width of the input symbols to the preamble detector

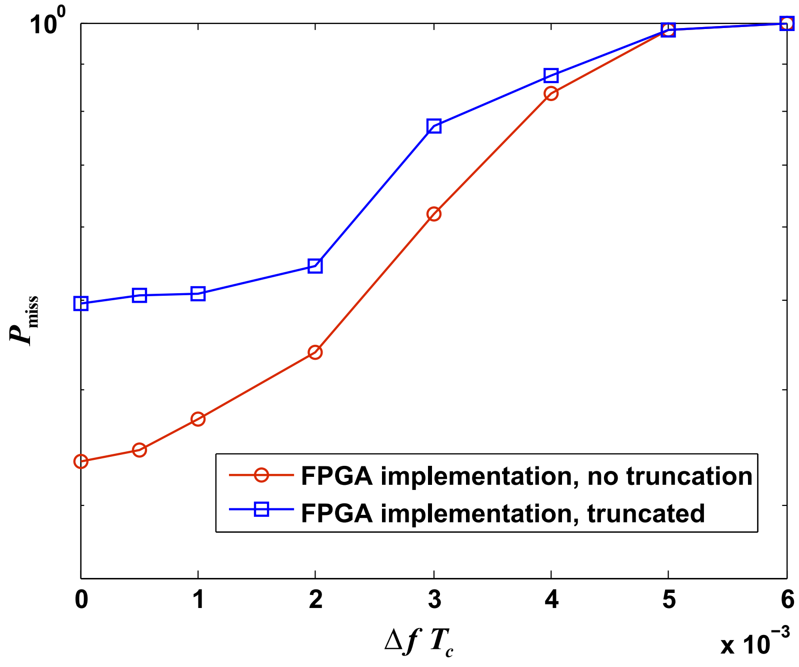


Figure 4.11: FPGA Implementation, P_{miss} vs. $\triangle f T_c$ for $P_{false} = 10^{-3}$ and $pSNR = 19dB$.

is $B = 2$ for this measurement. This figure compares the performance of the non-truncated preamble detector with the truncated preamble detector. It also includes the MATLAB simulation results (with the same design parameters) as the benchmark. For the MATLAB simulations, the noise symbols generated in the FPGA and then captured by the ChipScope were used. Note that MATLAB simulations show the better performance of the preamble detector using floating-point computations. As shown in this figure, the P_{miss} of the truncated preamble detector is at most 1dB worse than the non-truncated detector while in terms of area, the truncated detector introduces about 50% saving based on the report in Table 4.3. Thus, the truncated preamble detector is enhanced in terms of area, and also potentiality in speed and power consumption, while the degradation in its performance is modest (only about 1 dB) comparing to the non-truncated detector.

Figure 4.11 shows the effect of frequency offset on the probability of missing preamble, for both the non-truncated and truncated preamble detector, when $pSNR = 19dB$ and $P_{false} = 0.001$. To keep the $P_{false} = 0.001$, the threshold value

G has to change accordingly for different values of offset. The spreading sequence and preamble sequence are as given in Table 2.2. The width of the input symbols to the preamble detector is $B = 2$ for this measurement. The results presented in this figure show that for $\Delta f T_c < 0.001$, the performance degradation is insignificant in either the truncated preamble detector or the non-truncated preamble detector. Thus the implemented preamble detector is robust to the frequency offset as long as $\Delta f T_c < 0.001$.

Table 4.3: FPGA area utilization of RX unit.

Design	**DCM**	**MULT18X18**	**Slice**	**Speed (`clk_chip`)**
non-truncated detector	1 out of 8 (12%)	2 out of 28 (7%)	6926 out of 8672 (79%)	90 MHz
truncated detector	1 out of 8 (12%)	2 out of 28 (7%)	3589 out of 8672 (41%)	91.25 MHz

Table 4.4: DV power report for RX unit.

Module	**Switching Power (mW)**	**Internal Power (mW)**	**Leakage Power (pW)**	**Total Power (mW)**
original rx_unit	221.932	23.318	$2.58e7$	245.276
optimized rx_unit	75.807	12.327	$1.16e7$	88.146

Table 4.5: DV area report for RX unit.

Module	**Total area** (μm^2)
original rx_unit	1410219.625 (plh_extractor: 1%, preamble_detector: 49.5%)
optimized rx_unit	668759 (plh_extractor: 2%, preamble_detector: 49%)

Table 4.3 reports the area utilization of the RX unit module (two preamble detectors and one PLH extractor) on the Spartan $3e$ FPGA ($XC3S1200e-4fg320$). It compares the area utilization of the non-truncated design with the truncated one. The last column of the table reports the highest achievable frequency for `clk_chip` of the design. As seen in the table, the truncated design uses almost half the available slices inside the target FPGA compared to the non-truncated design while its performance, shown in Figure 4.10, doesn't degrade significantly.

In order to compare the improvement (in terms of area and power) comparing the original preamble detector and the optimized (truncated) preamble detector, both of these design are synthesized using DV. Table 4.4 and 4.5 report the power and area estimates (generated by DV) of the original and optimized RX unit, respectively. Note that the reported power doesn't include the IO. Also, the report assumes the frequency of `clk_2x_chip` is 100 MHz. As shown in the tables, the optimized RX unit consumes almost a third of the power and about half the area comparing to the original RX unit design.

Chapter 5

Conclusions

The wide range of potential applications of WSNs has attracted the attention of researchers. The power consumption and size are the most important challenges in the design of WSN. Recently emerged applications of WSN, such as neural prostheses and brain-computer interfaces, introduce other challenges such as wide-band robust communication link design.

Packet-based, asynchronous MAC schemes are matching candidates for WSN applications. The preamble detection is an important task in packet-based communications. A low-power, low-complexity preamble detector for wide-band WSN communication was proposed in [11]. The goal of this research was to fabricate the transmitter design presented in [11]. The theory of the transceiver design was briefly discussed in Chapter 2. The MATLAB simulations presented in this chapter compared the performance of the design in two scenarios: packets with no PLH, and packets with a 64-bit PLH, and showed that if the PLH utilizes the same m-sequence as the preamble it causes about 3dB degradation in performance of the preamble detector. To avoid this degradation, the PLH should use a common but different m-sequence.

Chapter 3 explained the detail of the architecture of transmitted chip that was fabricated using IBM Corporation's 130-nm digital CMOS process. The fabricated chip was tested successfully at a nominal clock frequency of 12.5 MHz and a power supply value of VDD=1.2V and DVDD=2.5V. The embedded JTAG logic in fabricated chip facilitated the testing of the design with different system parameters. The packet structure used in fabricated chip only included the preamble and PLH. Including the payload processing in the transmitter design is an important priority for future work of the project.

Chapter 4 explains the details of the preamble detector that was implemented and verified on an FPGA. Since the preamble detector that was originally implemented on the FPGA (presented in [12]) used a huge amount of logic, one goal was to reduce the area (logic utilization) of the preamble detector without compromising the performance significantly. An optimized preamble detector was presented in this chapter and its performance and area utilization were compared to the original preamble detector. Although the optimized preamble detector perform about 1 dB worse comparing to the original detector, it provides about 50% area saving. All of the test runs were performed on FPGA boards designed by Microlynx Systems Ltd. and their outcomes shown to be close to the results of MATLAB simulations. As with the prototype transmitter, the receiver design only includes the logic for preamble detection and PLH extraction. The design of payload extraction logic is an important priority for future work.

Bibliography

[1] C. Schelegel and D. Majumdar, *AHFMR Project: Smart Neural Prostheses*, HCDC Lab., Electrical and Computer Engineering Department, University of Alberta, 2011.

[2] A. Vachoux, *Top-down digital design flow, EDA tools: Mentor ModelSim, Synopsys Design Compiler, Cadence SoC Encounter (V3.4)*, Microelectronic Systems Lab, STI-IEL-LSM, 2008.

[3] (2009) XEM3010 User's Manual. Opal Kelly Incorporated, Portland, Oregon. [Online]. Available: http://www.opalkelly.com/library/XEM3010-UM.pdf

[4] E. T. T. Son, "Simulation of Quantization Noise Effects on the Performance of a Wireless Preamble Detector and Demonstration of a Functional FPGA Prototype," Master's thesis, University of Alberta, 2009.

[5] N. Abramson, "The ALOHA System–Another alternative for computer communications," in *Proc. Fall Joint Comput. Conf.*, Houston, TX, Nov. 1970.

[6] ——, "The Throughput of Packet Broadcasting Channels," *IEEE Trans. Comm.*, vol. COM-25, no. 1, pp. 117–128, Jan. 1977.

[7] J. Proakis, *Digital Communications.* McGraw-Hill, 2000.

[8] S. Haykin, *Digital Communications.* John Wiley & Sons, 1988.

[9] V. Shnayder, B. Chen, K. Lorincz, T. R. F. Fulford-Jones, and M. Welsh, *Sensor Networks for Medical Care*, Division of Engineering and Applied Science, Harvard University, Tech. Rep. TR-08-05, 2005.

[10] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton, "Codeblue: an ad hoc sensor network infrastructure for emergency medical care," in *Proc. Int. Workshop on Wearable and Implantable Body Sensor Networks*, London, UK, 2004.

[11] S. Nagaraj, S. Khan, C. Schlegel, and M. V. Burnashev, "Differential preamble detection in packet-based wireless networks," *IEEE Trans. Wireless Comm.*, vol. 8, no. 2, pp. 599–607, Feb. 2009.

[12] E. Son, B. Crowley, C. Schlegel, and V. Gaudet, "Architecture and FPGA Implementation of a Packet Detector for RF Motes," in *Proc. MILCOM'09*, Boston, MA, Oct. 2009.

[13] *CMOS8RF (CMRF8SF) Design Manual*, Mixed Signal Technology Development, IBM Microelectronics Division, 2010.

[14] *IBM CMRF8SF Process 1.2V Core, 2.5V I/O, 3.3V-Tolerant General Purpose In-Line (MA metal stack) I/O Library Databook*, ARM Inc., 2007.

[15] M. L. Bushnell and V. D. Agrawal, *Essentials of Electronic Testing.* Kluwer/Springer-Verlag, 2000.

[16] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Proc. the 4th Int. Symposium on Information Processing in Sensor Networks*, 2005, p. 364369.

[17] K. J. Choi and J.-I. Song, "A miniaturized mote for wireless sensor networks," in *Proc. the 10th Int. Conference on Advanced Communication Technology (ICACT)*, Feb. 2008, p. 514516.

[18] K. S. Jint, J. C. McEachent, and G. Singhl, "RF characteristics of Mica-Z wireless sensor network motes," in *Proc. the 49th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug. 2006, pp. 100–104.

[19] P. Chen, P. Ahammad, C. Boyer, S.-I. Huang, L. Lin, E. Lobaton, M. Meingast, S. Oh, S. Wang, P. Yan, A. Y. Yang, C. Yeo, L.-C. Chang, D. Tygar, and S. S.

Sastry, "CITRIC: A low-bandwidth wireless camera network platform," in *Proc. the 2nd ACM/IEEE Int. Conference on Distributed Smart Cameras (ICDSC)*, Sep. 2008, pp. 1–10.

[20] TELOSB platform datasheet (Document Part Number: 6020-0094-01 Rev B). Crossbow Technology Inc., San Jose, California. [Online]. Available: http://www.willow.co.uk/TelosB_Datasheet.pdf

[21] (2007) 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. Texas Instruments Inc., Dallas, Texas. [Online]. Available: http://www.ti.com/lit/ds/symlink/cc2420.pdf

[22] A. V. Nurmikko, J. P. Donoghue, L. R. Hochberg, W. R. Patterson, Y.-K. Song, C. W. Bull, D. A. Borton, F. Laiwalla, S. Park, YinMing, and J. Aceros, "Listening to brain microcircuits for interfacing with external world-progress in wireless implantable microelectronic neuroengineering devices," in *Proc. IEEE*, Mar. 2010.

[23] S. B. Lee, H.-M. Lee, M. Kiani, U.-M. Jow, and M. Ghovanloo, "An inductively-powered scalable 32-channel wireless neural recording system-on-a-chip for neuroscience applications," *IEEE Trans. Biomedical Circuits and Systems*, vol. 4, no. 6, pp. 360–371, Dec. 2010.

[24] A. M. Sodagar, G. E. Perlin, Y. Yao, K. Najafi, and K. D. Wise, "An implantable 64-channel wireless microsystem for single-unit neural recording," *IEEE J. Solid-State Circuits*, vol. 44, no. 9, pp. 2591–2604, Sep. 2009.

[25] J. Jung, S. Zhu, P. Liu, Y. J. E. Chen, and D. Heo, "22-pJ/bit Energy-Efficient 2.4-GHz Implantable OOK Transmitter for Wireless Biotelemetry Systems: In Vitro Experiments Using Rat Skin-Mimic," *IEEE Trans. on Microwave Theory and Techniques*, vol. 58, no. 12, pp. 4102–4111, Dec. 2010.

[26] F. Inanlou, M. Kiani, and M. Ghovanloo, "A 10.2 Mbps Pulse Harmonic Modulation Based Transceiver for Implantable Medical Devices," *IEEE J. Solid-State Circuits*, vol. 46, no. 6, pp. 1296–1306, 2011.

[27] M. Chae, W. Liu, Z. Yang, T. Chen, J. Kim, M. Sivaprakasam, and M. Yuce, "A 128-Channel 6mW Wireless Neural Recording IC with On-the-Fly Spike Sorting and UWB Tansmitter," in *IEEE Int. Solid-State Circuits Conf. Tech. Dig.*, San Francisco, CA, Feb. 2008.

[28] R. H. Barker, "Group Synchronizing of Binary Digital Systems," *Communication Theory, W. Jackson, Ed. Butterworth, London, England*, pp. 273–287, 1953.

[29] E. C. Posner, "Optimal search procedures," *IEEE Trans. Inform. Theory*, vol. 9, no. 1, pp. 157–160, 1963.

[30] J. Massey, "Optimum frame synchronization," *IEEE Trans. Commun.*, vol. 20, no. 2, pp. 115–119, 1972.

[31] G. Lui and H. Tan, "Frame synchronization for gaussian channels," *IEEE Trans. Commun.*, vol. 35, no. 8, pp. 818–829, 1987.

[32] L. B. Milstein, H. Gevargiz, and P. K. Das, "Rapid acquisition for direct sequence spread-spectrum communications using parallel SAW convolvers," *IEEE Trans. Commun.*, vol. 33, no. 7, pp. 593–600, 1985.

[33] E. A. Sourour and S. C. Gupta, "Direct-sequence spread spectrum parallel acquisition in a fading mobile channel," *IEEE Trans. Commun.*, vol. 38, no. 7, pp. 992–998, 1990.

[34] A. Polydoros and M. K. Simon, "Generalized serial search code acquisition: the equivalent circular state diagram approach," *IEEE Trans. Commun.*, vol. 32, no. 12, pp. 1260–1268, 1984.

[35] A. Polydoros and C. L. Weber, “A unified approach to serial search spread-spectrum code acquisitionpart I: general theory,” *IEEE Trans. Commun.*, vol. 32, no. 5, pp. 542–549, 1984.

[36] C. D. Chung, “Differentially coherent detection technique for directsequence code acquisition in a rayleigh fading mobile channel,” *IEEE Trans. Commun.*, vol. 43, no. 234, pp. 1116–1126, 1995.

[37] *IEEE std. 802.15.4-2006 part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs)*, 2006.

[38] H. D. Schotten and H. D. Luke, “On the search for low correlated binary sequences,” *AEU Int’l Jnl. Electronics and Comm.*, vol. 59, no. 2, pp. 67–78, May 2005.

[39] *Backend Digital Design Flow, IBM* 0.13μm *CMOS Technology with Artisan Standard Cell Libraries*, VLSI Research Group, University of Toronto, 2010.

[40] *Tutorial on CMCs Digital IC Design Flow*, Canadian Microelectronics Corporation, Kingston, Canada, 2001.

[41] *ASIC Design Flow Tutorial Using Synopsys Tools*, Nano-Electronics and Computing Research Lab, School of Engineering, San Francisco State University, San Francisco, CA, US, 2009.

[42] *Digital ASIC Design, A Tutorial on the Design Flow*, Digital ASIC Group, Lund University, Lund, Sweden, 2005.

[43] (2011) FrontPanel Programmer’s Interface. Opal Kelly Incorporated, Portland, Oregon. [Online]. Available: http://www.opalkelly.com/library/FrontPanelAPI/index.html

[44] F. Botman, D. Bol, C. Hocquet, and J.-D. Legat, “Exploring the Opportunity of Operating a COTS FPGA at 0.5V,” in *Proc. IEEE Subthreshold Microelectronics Conf.*, Sep. 2011.

[45] *Work Instructions Broadband Reference Radio Link*, Microlynx Systems Ltd., 2011.

[46] *Cell-Based IC Physical Design and Verification- SOC Encounter*, Delft University of Technology.

Appendix A

ModelSim: Simulation and Verification

A.1 RTL Simulation

To start ModelSim type *vsim* on the command line. To run the simulation, the following should be performed:

1. Create the working directory `work`, if not already present.
2. Compile all of the submodules, as well as the top module and the testbench.
3. Simulate the testbench and run the simulation for defined period of time.
4. Selected signals can be displayed in the *wave* window afterwards.

The testbench prepared for the RTL simulation configures the required JTAG data registers and then triggers the `test_start`. It also mimics the functionality of the TX where it generates the expected bit and chip sequence using non-synthesisable blocks of codes such as *for* and *while* loops. Every time `pkt_sent_ack` goes high, the testbench captures the sequences appearing on `header_bit_out` and `chip_out` and compares it with the expected sequence. If the result of the comparison (per packet stream sequence) is a match, then it prints the match message;

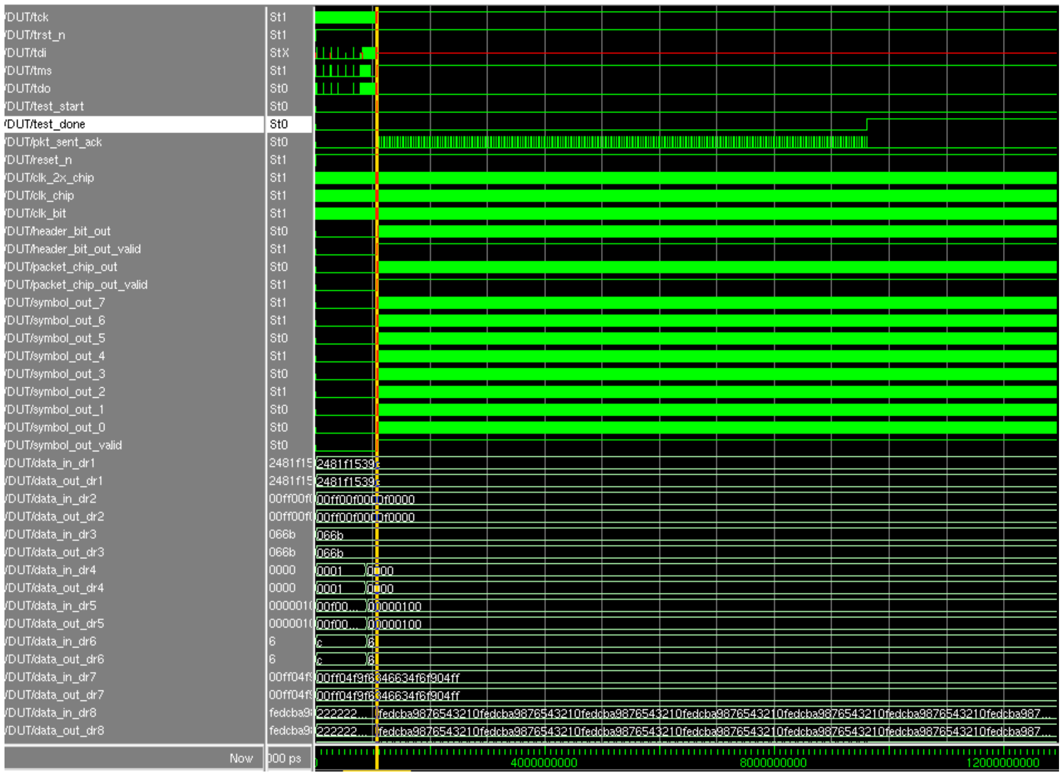


Figure A.1: RTL Simulation.

otherwise, it prints a meaningful error message and stops the simulation. Figure A.1 shows an example of the *wave* window in our simulations.

A.2 Post-synthesis Gate-level Simulation

The testbenches used for RTL model validation can be reused (with some modifications, for example to use the post-synthesis Verilog gate-level netlists). The gate-level simulation uses Verilog models for the logic gates that are provided in the design kit. These Verilog models follow the modeling standard to ensure proper back-annotation of delays using the SDF files generated by the synthesis or the place-and-route step [2].

The ModelSim simulator is started by typing *vsim* at the command line. To run the simulation, the following should be performed:

1. Create a working directory `work`.

2. Compile the Stdcells Verilog models, as well as the Verilog netlist generated by Synopsys DV and the testbench.

3. Simulate the testbench while including the SDF file to back-annotate the delays (the delays are inserted automatically by ModelSim once the SDF file is included).

4. Run the simulation for the defined period of time.

5. Selected signals can be displayed in *wave* window afterwards.

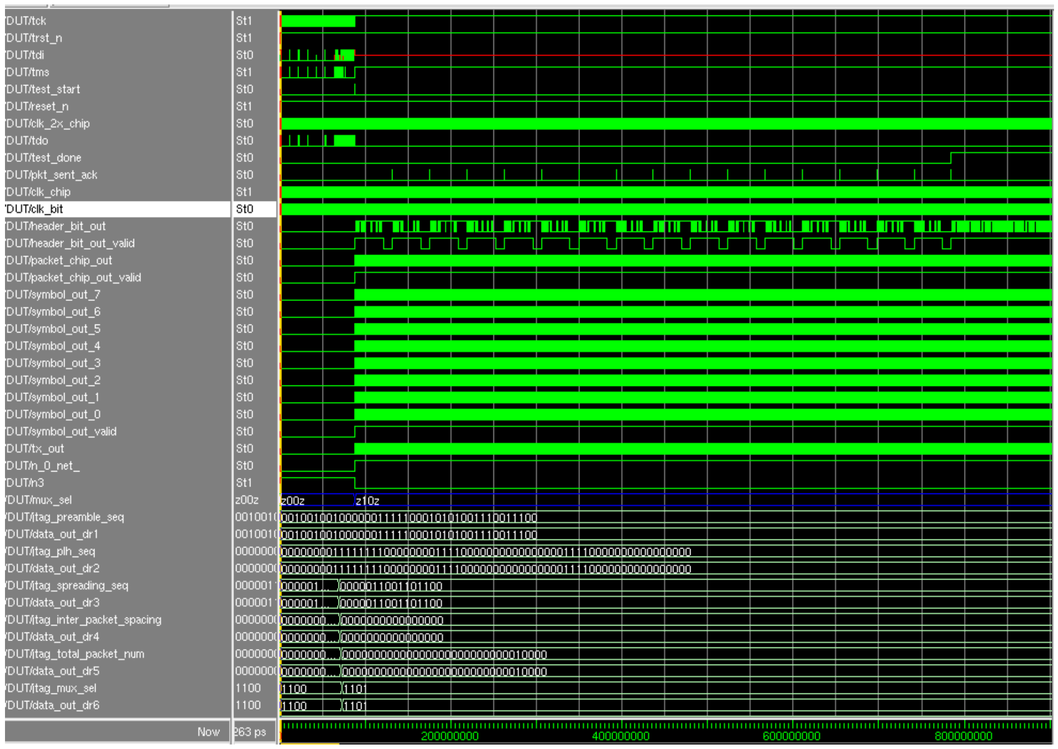

DUT/tck St1
DUT/trst_n St1
DUT/tdi St0
DUT/tms St1
DUT/test_start St0
DUT/reset_n St1
DUT/clk_2x_chip St0
DUT/tdo St0
DUT/test_done St0
DUT/pkt_sent_ack St0
DUT/clk_chip St1
DUT/clk_bit St0
DUT/header_bit_out St0
DUT/header_bit_out_valid St0
DUT/packet_chip_out St0
DUT/packet_chip_out_valid St0
DUT/symbol_out_7 St0
DUT/symbol_out_6 St0
DUT/symbol_out_5 St0
DUT/symbol_out_4 St0
DUT/symbol_out_3 St0
DUT/symbol_out_2 St0
DUT/symbol_out_1 St0
DUT/symbol_out_0 St0
DUT/symbol_out_valid St0
DUT/tx_out St0
DUT/n_0_net_ St0
DUT/n3 St1
DUT/mux_sel z00z z00z z10z
DUT/tag_preamble_seq 0010010 0010010010000001111100010101001110011100
DUT/data_out_dr1 0010010 0010010010000001111100010101001110011100
DUT/tag_plh_seq 0000000 0000000011111111000000001111000000000000000011110000000000000000
DUT/data_out_dr2 0000000 0000000011111111000000001111000000000000000011110000000000000000
DUT/tag_spreading_seq 000001 000001... 0000011001101100
DUT/data_out_dr3 000001 000001... 0000011001101100
DUT/tag_inter_packet_spacing 0000000 0000000... 0000000000000000
DUT/data_out_dr4 0000000 0000000... 0000000000000000
DUT/tag_total_packet_num 0000000 0000000... 00000000000000000000000000010000
DUT/data_out_dr5 0000000 0000000... 00000000000000000000000000010000
DUT/tag_mux_sel 1100 1100 1101
DUT/data_out_dr6 1100 1100 1101
Now 263 ps
200000000
400000000
600000000
800000000

Figure A.2: Post-synthesis Gate-level Simulation.

The testbench prepared for the post-synthesis simulation is the same as the one used in RTL simulation. Again whenever the `pkt_sent_ack` signal goes high, the testbench captures the sequences appearing on `header_bit_out` and `chip_out` and compares it with the expected sequence. If the result of the comparison is a match, it prints the match message; otherwise, it prints the error message and stops the simulation. Figure A.2 shows an example of the *wave* window in our

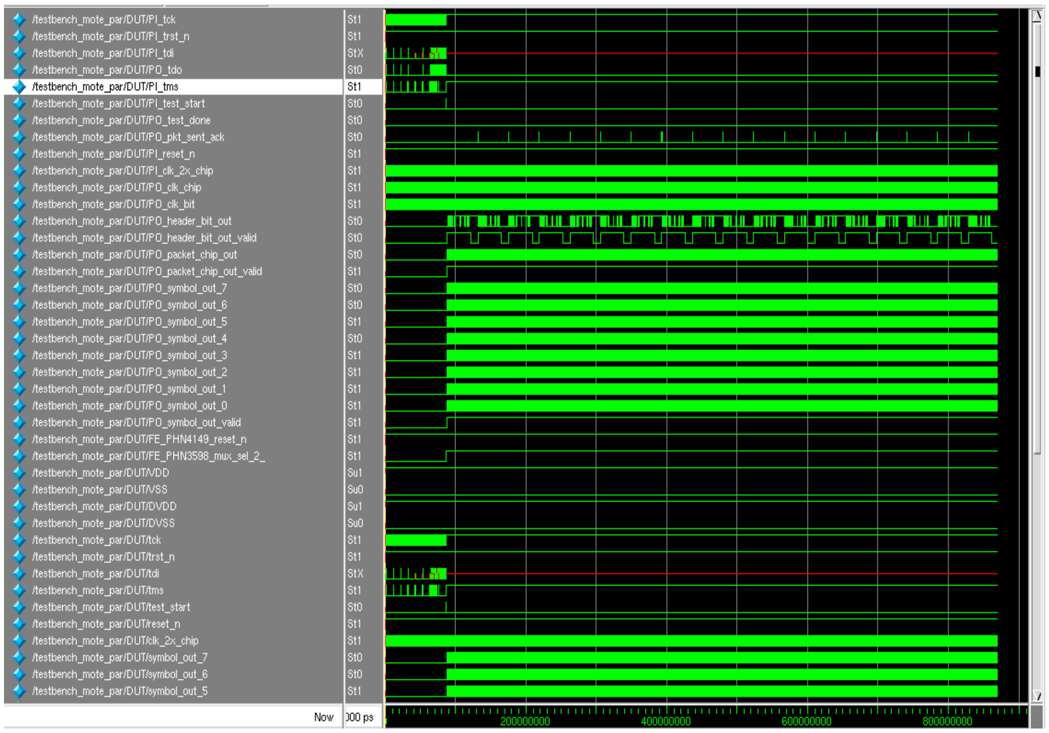

/testbench_mote_par/DUT/PI_tck
/testbench_mote_par/DUT/PI_trst_n
/testbench_mote_par/DUT/PI_tdi
/testbench_mote_par/DUT/PO_tdo
/testbench_mote_par/DUT/PI_tms
/testbench_mote_par/DUT/PI_test_start
/testbench_mote_par/DUT/PO_test_done
/testbench_mote_par/DUT/PO_pkt_sent_ack
/testbench_mote_par/DUT/PI_reset_n
/testbench_mote_par/DUT/PI_clk_2x_chip
/testbench_mote_par/DUT/PO_clk_chip
/testbench_mote_par/DUT/PO_clk_bit
/testbench_mote_par/DUT/PO_header_bit_out
/testbench_mote_par/DUT/PO_header_bit_out_valid
/testbench_mote_par/DUT/PO_packet_chip_out
/testbench_mote_par/DUT/PO_packet_chip_out_valid
/testbench_mote_par/DUT/PO_symbol_out_7
/testbench_mote_par/DUT/PO_symbol_out_6
/testbench_mote_par/DUT/PO_symbol_out_5
/testbench_mote_par/DUT/PO_symbol_out_4
/testbench_mote_par/DUT/PO_symbol_out_3
/testbench_mote_par/DUT/PO_symbol_out_2
/testbench_mote_par/DUT/PO_symbol_out_1
/testbench_mote_par/DUT/PO_symbol_out_0
/testbench_mote_par/DUT/PO_symbol_out_valid
/testbench_mote_par/DUT/FE_PHN4149_reset_n
/testbench_mote_par/DUT/FE_PHN3598_mux_sel_2_
/testbench_mote_par/DUT/VDD
/testbench_mote_par/DUT/VSS
/testbench_mote_par/DUT/DVDD
/testbench_mote_par/DUT/DVSS
/testbench_mote_par/DUT/tck
/testbench_mote_par/DUT/trst_n
/testbench_mote_par/DUT/tdi
/testbench_mote_par/DUT/tms
/testbench_mote_par/DUT/test_start
/testbench_mote_par/DUT/reset_n
/testbench_mote_par/DUT/clk_2x_chip
/testbench_mote_par/DUT/symbol_out_7
/testbench_mote_par/DUT/symbol_out_6
/testbench_mote_par/DUT/symbol_out_5
St1
St1
StX
St0
St1
St0
St0
St0
St1
St1
St1
St1
St0
St0
St0
St1
St0
St0
St1
St0
St1
St1
St1
St1
St1
St1
St1
Su1
Su0
Su1
Su0
St1
St1
StX
St1
St0
St1
St1
St0
St0
St1
Now
000 ps
200000000
400000000
600000000
800000000

Figure A.3: Post-P&R Gate-level Simulation.

simulations. As shown in the figure, the design functionality is just the same as the RTL model (with no setup/hold violation error) as long as we keep the frequency of the `clk_2x_chip` and `tck` less than the targeted frequency in synthesis (which are 100 MHz and 10 MHz, respectively). The only difference that was seen between the post-synthesis and RTL (pre-synthesis) simulations is that the generated `clk_chip` doesn't have a duty cycle of 50%. We run the simulations both for the minimum and maximum delay values (*-sdfmin* and *-sdfmax*) provided in the SDF file.

A.3 Post-P&R Gate-level Simulation

The testbench prepared for the post-P&R simulation is the same as the one used in RTL simulation. Again whenever `pkt_sent_ack` goes high, the testbench captures the sequences appearing on `header_bit_out` and `chip_out` and compares it with the expected sequence. If the result of comparison is a match, it prints the match message; otherwise, it stops the simulation. Figure A.3 shows an example of the *wave* window in our simulations. As it is seen in the figure, the design functionality is just the same as the RTL model (with no setup/hold violation error) as long as we keep the frequency of the `clk_2x_chip` and `tck` less than 80 MHz and 10 MHz, respectively. We ran the simulation for both the minimum and maximum delay values (-*sdfmin* and -*sdfmax*) provided in the SDF file.

Appendix B

Synopsys: Design Vision

An automated script is used to synthesis the design, generate gate-level netlist and report timing information. The script also has the necessary commands to generate the SDC constraints file required by FE and the SDF description file to be used for post-synthesis simulation. This section summarizes the steps that we followed to generate these files using DV.

Table B.1 shows the required StdCell and IO libraries for the $0.13\,\mu$m CMOS process.

Table B.1: Libraries Required in Design Vision.

Library	**Description**
scx3_cmos8rf_rvt_tt_1p2v_25c.db	StdCell library
iogpil_cmrf8sf_rvt_tt_1p2v_2p5v_25c.db	IO library

The required list of libraries and files used in synthesis can be added into a single setup file, .synopsys_dc.setup. This file should be placed in the DV `work` directory. To start the DV interface type *design_vision*. An automated script, including the following steps can be used:

1. Create design library `WORK` and analyze RTL sub modules and then the top module (it is important to analysis the modules in the right order going from submodules to top modules).

2. Elaborate the top module (mote_tx_top.v).

3. Check the design. The return value of 1 means there were no errors in the design. DV provides both symbol and schematic views of the design that can be used as a check.

4. Define the clocks and their properties such as frequency, uncertainty, etc. In our design `clk_2x_chip` is the master input clock (with requested maximum frequency of 100 MHz). `clk_chip` and `clk_bit` are clocks derived from `clk_2x_chip` (with requested maximum frequency of 50 MHz and 3.125 MHz, respectively) that should be introduced as `generated_clock` [46]. Also `tck` is defined as a clock (with requested maximum frequency of 10 MHz). `don't_touch_network` property is set for all of these clocks to prevent DC from inserting clock tree buffers (to be done at a later stage of the flow by FE) [39]. `report_clock` command can be used to check the list of master and generated clocks and their defined properties.

5. Specify capacitive loads in `pF` on input and output. The values we used for the I/O loads are defined by the pin capacitance provided in [14].

6. Set a chip area constraint. Setting the maximum area to $0\,\mu\mathrm{m}^2$ will force the DV to optimize for the smallest silicon area. In case of conflicting optimization goals, such as maximum clock frequency and minimum area, by default DV optimizes for timing first and then area second.

7. Specify `map_effort_level` and `area_effort_level`, and compile the design.

8. Check the design and generate desired reports (such as timing, hierarchy power and area reports). There was no negative slack in the logic delay of any clock groups in the report that the tool generated for our design. The amount of required area (only for the logic) is $159647.035789\,\mu\mathrm{m}^2$. The hold time violation was solved at a later stage of the flow by FE.

9. Before generating the Verilog netlist, some naming rules must be applied to the design.

10. Generate the synthesized netlist in Verilog format (mote_tx_top_syn.v). Also timing constraint file is generated (mote_tx_top_constraints.sdc). These two files are required to be imported into the FE tool in later stages. Also the SDF file is generated so that it can be used for back-annotation of the delays onto the Verilog netlists for post-synthesis simulation.

Automatic pad insertion is not supported by DV for the IBM 0.13 μm physical design kit. Therefore, netlist generated by DV should be modified and pads must be assigned to top-level ports. The sequence of steps for including the pads is listed below:

1. Rename the top-level netlist generated by DV (mote_tx_top_syn_wPads.v).

2. Change the name of the top-level output and input ports by prefixing the names with `PO` and `PI`, respectively.

3. Instantiate the desired pads according to the datasheet [14]. `PVDD` and `PVSS` are the pads used for Core VDD and VSS, respectively. `PDVDD` and `PDVSS` are the pads used for the IO ring power terminals VDD and VSS, respectively. `PIC`, `POC2A` and `PCORNER` are the pads used for input pins, output pins and corner pins respectively.

Note that having modified top-level port names (by prefixing the names with `PO/PI`); we must update the .sdc timing constraint file generated by DV before pad insertion since it relies on the names of top-level ports. Therefore, we should replace the old port names with new prefixed names created during pad insertion.

Appendix C

Cadence: First Encounter

This section summarizes the main steps followed to generate the placed-and-routed netlist using FE. To start with FE, the following five files must be available in the intended work directory:

1. Synthesized netlist generated by DV, modified after pads insertion (mote_tx_top_syn_wPads.v).
2. Timing constraint file generated by DV, modified after pads insertion (mote_tx_top_constraints.sdc).
3. Configuration file (.conf file).
4. IO placement file (.io file).
5. Clock tree synthesis technology file (.ctstch file).

To start FE, enter *encounter* at the command prompt. The following are the steps (using the FE GUI) which we followed.

C.1 Design Import

1. Select *File* ⇒ *Import Design.*
2. Click on *Load* and read in .conf file. Alternatively, we can manually specify the *Timing Libraries* (which include information on the cell timings such as

delays, setup/hold times) and *LEF Files* (LEF (Layout Exchange Format) files provides information such as metal and via layers and via generate rules which is used for routing tasks. They also provide the minimum information on cell layouts for placement and routing).

3. Add DV synthesized netlist (gate-level netlist generated by DV after modification for pads) and check off *Auto Assign* for *Top Cell* (to let the tool extract the top cell name from the file).

4. Specify the .sdc *Timing Constraint File* and .io *IO Assignment File*.

5. In the *Advanced* tab

   - Click on *IPO/CTS* and enter *BUFX2TF DLY1X1TF INVX1TF* for *CTS Cell List*.
   - Click on *Power* and enter *VDD* for the *Power Nets* field and *VSS* for the *Ground Nets* field (The names of power and ground nets must be the same as the ones used in the LEF file that describes the standard cells).

6. Save these settings by clicking *Save* at the bottom of the window. We can *Load* these settings if we need to re-import the design.

7. Click *OK* to start the design import process.

8. We can save the design using the command *Design* ⇒ *Save Design* under a new name (e.g. mote_tx_imported.enc) in each of the steps that follows so that the design can be restored at each step.

C.2 Floorplan

In the initial floor plan, the size of the die and the boundary of the core are assigned. In the IBM 0.13 μm CMOS process, the **I/O** pad size is 247 μm by 73 μm, the **Corner** pad size is 247 μm by 247 μm and the **Filler** pad size is 1 μm by 247 μm. In our design, the pad's area is more than the core area, thus the die size is defined based on the required pad's area. The total number of IO pads, including the core and IO power pads, are 41 which are arranged around the core as, 6 pads on top, 6 pads on bottom, 14 pads on right and 15 pads on left. In order to balance the right side with left, we need to insert filler pads.

1. Select *Floorplan* ⇒ *Specify Floorplan.*

2. Specify the die size by selecting *Specify By* ⇒ *Die Size By* ⇒ *Width*$= 932\,\mu$m, *Height*$= 1589\,\mu$m.

3. Specify spacing for the *Core to IO Boundary* (e.g $35\,\mu$m) all around the core to create room for power rings. Click *Apply* and *OK.*

C.3 Power Planning

The following steps describes how to add power rings and stripes. These rings and stripes are required to distribute VDD and VSS throughout the core evenly:

1. Choose *Power* ⇒ *Power planning* ⇒ *Add Rings.*

2. Enter the *Nets* for which you want to have a ring created: *VDD, VSS.*

3. Under *Ring Type*, select *Core ring contouring* and *Around core boundary.*

4. Specify *Ring Configuration* (we used the default values). Choose *Center in channel* for the ring *Offset* and click *OK.*

To place power stripes:

1. Choose *Power* ⇒ *Power planning* ⇒ *Add Stripes.*

2. Enter *VDD, VSS* in the *Nets* field.

3. Set the position of *First/Last Stripe Relative from core or selected area.* Set *X from left*(= 30 is used in our design), then click *OK.*

C.4 Placement

To run full-scale placement:

1. Click on *Place* ⇒ *Place Standard Cell.*
2. Select *Run Full Placement.*
3. Select the desired *Optimization Options* (in our design we activated *Include Pre-Place Optimization*, and also *Include In-Place Optimization*).
4. Click on *Mode* and select *Run Timing Driven Placement*, then click *OK.*
5. Observe placed cells by selecting *Place* ⇒ *Display* ⇒ *Display Spare Cell.*

C.5 Clock Tree Insertion

As mentioned before, in our design `clk_2x_chip` is the master input clock, where `clk_chip` and `clk_bit` are the two generated clocks derived from it. `tck` is another input clock into our design which is used only by JTAG module. FE can generate specifications of the clocks (such as period, skew, etc) based on the constraint given in the `.sdc` file. The `generated_clock` property in the `.sdc` file is reflected in the `Through_Pin` property of the clock tree file. The list of buffer footprints and inverter footprints is given in the `Buffer` section of the clock tree file. To synthesize the clock tree:

1. Select *Clock* ⇒ *Synthesize Clock Tree*, then in *Basic* tab specify *Clock Specification Files*, then click *OK.*
2. To view the synthesized clock tree, choose *Clock* ⇒ *Display* ⇒ *Display Clock Tree.* Note: at this stage, the clock tree is not routed.

C.6 Power Routing

It is a good idea to perform power routing first, before starting the chip routing, to create direct power connections between power domains [39]. The following steps are to demonstrate how to perform power routing:

1. Select *Route* ⇒ *Special Route.*

2. Make sure both power nets *VDD, VSS* are specified in the *Nets* field.

3. Turn off *Block Pins* and *Pad Rings* in the *Route* field and click *OK.*

C.7 NanoRoute

The Nanoroute performs global and detailed routing for the design:

1. Select *Route* ⇒ *NanoRoute* ⇒ *Route.* In *Concurrent Routing Features* select *Insert Diodes* and input *ANTENNATF* for *Diode Cell Name.*

2. Click on *Timing Driven* and select the *Effort* level (10 is used in our design).

3. Click on *Attribute* and type *DVDD, DVSS* in *Net Name* filed, then set *Skip Routing* to *TRUE* and click *OK* in *Attribute* window and also *NanoRoute* window.

C.8 Timing Optimization and Report

1. Click on *Options* ⇒ *Set Mode* ⇒ *Specify Operating Condition/PVT ....* In the *max* tab, select the slow operation condition. In the *min* tab, select the fast operation condition. The max operating conditions is used to meet setup timing constraints, while the min operating conditions is used to meet hold timing constraints. Running the `getOpCond -v` command in the Encounter console gives the active operating conditions.

2. Click on *Optimize* ⇒ *Optimize Design.* Select *postRoute* in *Design Stage* field.

3. Check off *Hold* and *Design Rule Violations*, click *OK.*

4. Inspect the log filed and iterate if any violations.

5. Click on *Optimize* ⇒ *Optimize Design.* Select *postRoute* in the *Design Stage* field.

6. Check off *Hold* and *Setup* and *Design Rule Violations*, click *OK.*

7. Inspect the log files and iterate if any violations.

8. Click on *Timing* ⇒ *Report Timing Analysis.*

9. Select *Post-Route* in *Design Stage field* and choose *Analysis Type* to be *Hold* once and *Setup* another time, click *Apply* and *OK.*

10. Inspect the *timingReports* folder.

Since we iterated the timing optimization until there are no violations, there is no slack in the reports. Note that any timing slack in post-route timing analysis must be corrected through *Design Vision* by modifying the timing constraints and re-iterating the design flow.

C.9 Filler Cell Placement

Filler cells are to make continuity between segmented standard cells by filling in the gaps and avoiding design rule violations. To run filler cell placement:

1. Open *Place* ⇒ *Physical Cell* ⇒ *Add Filler* menu and click *Select* to specify *Cell Name(s).*

2. Highlight all filler cells under *Cells List* and click *Add.* Close the dialog window.

3. Note the default *Prefix* name. This name is appended to all instantiated filler cells.

4. Keep remaining options unchanged. Press *OK* to begin filler cell placement.

Now there is no blank space between the standard cells and all row of the design are completely filled.

C.10 Verify Geometry

To run DRC geometry check:

1. Open *Verify* ⇒ *Verify Geometry* menu.
2. Keep default verification options and click *OK*.
3. Inspect verify geometry log generated at the command prompt and fix DRC violations (no violation in our design).
4. To examine DRC violations: select *Tools* ⇒ *Violation Browser*.

Geometry verification must be performed in all steps that modify the layout.

C.11 Metal Fill

Metal fill evens out the metal density across each layer and thus it enables the uniform application of insulating material. IBM performs auto fill on the following layers: *M1, M2, M3, MQ, MG* so designers should not attempt to add fill to these layers. Designers are responsible to adding fill to meet the density requirements (range of 23% to 70%) on the following layers: *LY, E1, MA*. Metal Fill is carried out in two steps: set-up and fill. To set-up metal fill:

1. Select *Route* ⇒ *Metal Fill* ⇒ *Setup.*
2. Input technology specific parameters under *Size and Spacing*, *Window and Density* tabs.
3. Click *OK* and inspect the generated log.

To add metal fill:

1. Select *Route* ⇒ *Metal Fill* ⇒ *Add.*
2. Make sure *Tie High/Low to Net(s)* option is on.
3. In the *Incremental Control* field, enable *Delete Metal Fill before Creating New Metal Fill* option.

4. In the *Layer Selection field*, choose the layers *LY, E1, MA*.

5. Press *OK* to start metal fill process.

Verify geometry of the metal filled design as described in Section C.10.

C.12 Verify Metal Density

1. Select *Verify* ⇒ *Verify Metal Density.*

2. In the *Layers* filed, specify the metal filled layers separated by space (e.g. *M6 M7 M8*).

3. Press *OK* to verify metal density. Inspect the generated log for DRC violations.

4. To examine DRC violations: select *Tools* ⇒ *Violation Browser.*

Other verification under *Verify* tab that should be done are: *Verify Connectivity*, *Verify Process Antenna*, *Verify Power Via.*

C.13 Post-route Timing Data Extraction

This step generates the post-route SDF file that includes both the actual interconnect and cell timing delays.

1. The parasitics must first be extracted. Select *Timing* ⇒ *Extract RC....*

2. Check off *Save Cap to* and *Save SPEF to* and assign the name of the two files. The generated Cap file includes the wired capacitance, pin capacitance, total capacitance, net length, wire cap per unit length and the fanout of each net in the design. The generated SPEF (Standard Parasitics Exchange Format) file includes RC values in a SPICE-like format.

3. Select *Timing* ⇒ *Write SDF* to generate the SDF file. Check off *Ideal Clock* (which means that flip-flops are assumed to have $0ps$ rising and falling transition times) and assign the file name.

C.14 Exporting Completed Design

This section describes how to export the design as a netlist and a GDS file. To export the design as a netlist:

1. Select *File* ⇒ *Save* ⇒ *Netlist*.
2. Save the netlist under a new name (mote_tx_top_par.v) and press *OK*.

The exported netlist has been used for post-place & route simulations.

To export your design as a GDS stream-out:

1. Select *File* ⇒ *Save* ⇒ *GDS/OASIS*.
2. Make sure *GDSII/Stream* is enabled for *Output Format*.
3. Enter *Output File* name (mote_tx_top_wPads.gds).
4. Make sure *Map File* field contains map file (gds_encounter.map).
5. Make sure *Structure Name* is enabled and contains the name of the top-level module.
6. Important: Enable *Write abstract information for LEF Macros* to generate LEF models for StdCells in the design.
7. Press *OK* to generate GDS stream-out.

C.15 Power Analysis

The power report generated by DV is not accurate since the clock tree buffers are not in place and timing optimization of the design is not done either. (Note that the tool could insert buffers into the design to make it meet the timing requirement. For that reason, the timing constraints of the design should be realistic in order to avoid unnecessary amount of buffering that introduces higher power consumption.) Once the place and route of the design is done, the tool (Cadence FE here) could report more realistic numbers. To perform the power analysis:

1. Select *Power* $\Rightarrow$ *Power Analysis* $\Rightarrow$ *Setup.*

2. Select *Corner:max* for the analysis of maximum power consumption or Select *Corner:min* for the analysis of minimum power consumption.

3. Select *Power* $\Rightarrow$ *Power Analysis* $\Rightarrow$ *Run.*

4. In *Basic* tab, pick the value for *Input Activity*, *Dominant Frequency* and *Flop Activity.* We picked 0.1, `clk_2x_chip=100` and 0.2 for these parameters respectively.

5. Press *OK* to generate the report.

Appendix D

Cadence: Layout

Before importing GDS stream generated by FE into Cadence, we need to add standard cell libraries to cds.lib:

```
DEFINE ibm13rfrvt ./ibm13rfrvt
DEFINE iogpil_cmrf8sf_rvt ./iogpil
```

We actually copied the standard cell libraries provided by the University of Toronto into the work directory since those libraries have the abstract view of the cells. We also need to set up the Cadence environment. To start Cadence enter *./startcmosp13* at the command prompt.

D.1 Import Design into Cadence

1. In *CIW*: *IBM_PDK* ⇒ *Library* ⇒ *Create*.
2. Specify the name and *attach it to an existing techfile*.
3. Link the library to *cmrf8sf*.
4. Specify 3 – 2 for number of levels of metal.

To import GDS stream into Cadence:

1. In *CIW*: *File* ⇒ *Import* ⇒ *Stream...*.
2. Indicate GDS name under *Input File*.

3. Indicate the stream-in library under *Library Name*.
4. Click *User Defined Data* and indicate *Layer Map Table*, then press *OK*.
5. Click *Options* and set *Retain Reference Library (No Merge)* to *ON* and *Reference Library Order* to *ibm13rfrvt iogpil_cmrf8sf_rvt*, then press *OK*.

D.2 Chip Boundary

Before inserting the chip boundary, configure Cadence bindkeys for IBM PDK:

1. Exit Cadence.
2. Change directory to the working directory.
3. Open .cdsinit and append the following to the end of the file:

```
load(strcat( ibmPdkPath "cmrf8sf/V1.7.0.2DM/cdslib51
/Skill/ibmPdkBindkeysCDS.il"))
envSetVal("asimenv.startup" "simulator" 'string "spectre")
ciwID=hiGetCIWindow()
hiSetWinStyle('default) ; else 'interactive
ciwID->useScrollbars=t ; else nil
ciwID->backingStore=nil ; else t
ciwID->hivNewWinOnHierTrav=nil ; else t
ciwID->infix=t ; else nil
ciwID->expertMode=nil ciwID->displayMouseBinding=t ; new user
ciwID->focusToCursor=t ; else nil
hiSetUndoLimit(3) ; maximum undo operations saved
ciwID->nestLimit = 20 ; maximum depth of nested commands
hiSetMultiClickTime(200) ; mouse action
```

4. Save .cdsinit and restart Cadence.

To insert the chip boundary:

1. In the *Library Manager*, open the *layout* view of the top-cell.
2. Insert the *image_bevel* cell found in *cmrf8sf* library.
3. Specify area dimensions of the boundary equal to the area of the die specified in FE plus the *image_bevel* corner area, thus in our design the dimension is $(932 + 300)\,\mu$m by $(1589 + 300)\,\mu$m.
4. Right click on the *image_bevel* cell and specify the origin coordinates: $(0, 0)$.
5. Center the chip inside the boundary.

To avoid DRC short violations caused by the bevel:

1. Select *image_bevel* in top-level layout view. Click *Edit* ⇒ *Hierarchy* ⇒ *Flatten...*.
2. Set *Displayed levels* and *Flatten PCells* to *ON*.
3. For each corner of the boundary, select and delete all layers of the L-shape inside the bevel and also the *logobnd*.

D.3 Calibre DRC

To run DRC check on the imported GDS file:

1. From the *layout* view: *IBM_PDK* ⇒ *Checking* ⇒ *Calibre* ⇒ *DRC*.
2. Set *BEOL_STACK* to $3 - 2 - 3$.
3. Set *CHECK_RECOMMENDED* to *ON*.
4. Set *DESIGN_TYPE* to *CELL*.
5. Set *LASTMETAL* to *MA*.
6. Set *NUMMETAL* to 8, then press *OK*.

7. Wait for the *Calibre Interactive* screen to open. Click on *Rules* and set the path in *DRC Rules File.* In our design we set the path to `/CMC/kits/cmosp13.V1.7.0.0DM/IBM_PDK/cmrf8sf/V1.7.0.2DM/Calibre /DRC/cmrf8sf.drc.cal`.

8. Click on *Inputs* and turn off the *Export from layout viewer*, then click *Run DRC.*

9. Examine all DRC errors and solve them.

D.4 Exporting the Design for Fabrication

To export the final layout from Cadence into GDS:

1. In *CIW*: *IBM_PDK* ⇒ *GDS2/GL1 Translation*:

   ```
   Library Name:  <name of the new library>
   Top Cell Name:  <name of the top-level cell>
   Technology:  cmrf8sf
   Translation:  Cadence into GDS2
   Convert PIN labels to text layer:  yes
   Scale UU: 0.01
   GDS2 name:  <stream-out name>.gds
   Specify cds <-> gds2 map
   Specify PIPO log:  PIPO_<stream-out name>.log
   ```