An FPGA implementation of a sleep enabled PON system

Zheyu Liu
New Jersey Institute of Technology

Follow this and additional works at: https://digitalcommons.njit.edu/theses

Part of the Electrical and Electronics Commons

Recommended Citation
https://digitalcommons.njit.edu/theses/257

This Thesis is brought to you for free and open access by the Electronic Theses and Dissertations at Digital Commons @ NJIT. It has been accepted for inclusion in Theses by an authorized administrator of Digital Commons @ NJIT. For more information, please contact digitalcommons@njit.edu.
Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be “used for any purpose other than private study, scholarship, or research.” If a user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of “fair use” that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select “Pages from: first page # to: last page #” on the print dialog screen
The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.
ABSTRACT

An FPGA Implementation of a Sleep Enabled PON System

by

Zheyu Liu

Owing to the growing demand for bandwidth-hungry video-on-demand applications, Passive Optical Network (PON) has been widely considered as one of the most promising solutions for broadband access. Environmental concerns motivated network designers to lower energy consumption of optical access networks. A well-known approach to reduce energy consumption is to allow network elements to switch to the sleep mode.

In this framework, an improved Optical network Unit (ONU) architecture in TDM-PON is proposed to reduce the handover time of status switching. Energy-saving performances of current and improved architectures are compared in different scenarios. The simulation results show that by applying a proper sleep mode mechanism, the improved architecture can effectively reduce the ONU energy consumption. We further implement the cycle sleep scheme on a multi-ONU testbed based on the improved ONU architecture. The experimental results have substantiated the viability of the improved ONU architecture.
AN FPGA IMPLEMENTATION OF A SLEEP ENABLED PON SYSTEM

by
Zheyu Liu

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Helen and John C. Hartmann Department of
Electrical and Computer Engineering

January 2016
AN FPGA IMPLEMENTATION OF A SLEEP ENABLED PON SYSTEM

Zheyu Liu

Nirwan Ansari, Thesis Advisor
Distinguished Professor of Electrical and Computer Engineering, NJIT

John Carpinelli, Committee member
Professor of Electrical and Computer Engineering, NJIT

Edwin Hou, Committee member
Professor of Electrical and Computer Engineering, NJIT
BIOGRAPHICAL SKETCH

Author: Zheyu Liu
Degree: Master of Science
Date: January 2016

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering, New Jersey Institute of Technology, Newark, NJ, 2016
- Master of Science in Electrical Engineering, Xidian University, Shaanxi, China, 2011
- Bachelor of Science in Electrical Engineering, Xidian University, Shaanxi, China, 2008

Major: Electrical Engineering
DEDICATION PAGE

This thesis work is dedicated to my husband, Yi Xu, who has been a constant source of support and encouragement during the challenges of graduate school and life. I am truly thankful for having you in my life.

This work is also dedicated to my parents, who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.
ACKNOWLEDGMENT PAGE

Firstly, I would like to express my sincere gratitude to my advisor Prof. Nirwan Ansari for the continuous support of my study, and for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. John Carpinelli, and Prof. Edwin Hou, for their encouragement, and insightful comments.

I must also acknowledge Mina Taheri hosseinabadi, for her suggestions, and provision of the materials evaluated in this study.

This thesis is based upon work supported by the National Science Foundation under Grant No. CNS-1218181.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>3</td>
</tr>
<tr>
<td>1.2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>2.1</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>15</td>
</tr>
<tr>
<td>3.2.1</td>
<td>15</td>
</tr>
<tr>
<td>3.2.2</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>23</td>
</tr>
<tr>
<td>4.2</td>
<td>26</td>
</tr>
<tr>
<td>4.3</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1</td>
<td>28</td>
</tr>
<tr>
<td>4.3.2</td>
<td>29</td>
</tr>
<tr>
<td>4.3.3</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>APPENDIX A</td>
<td>36</td>
</tr>
<tr>
<td>APPENDIX B</td>
<td>62</td>
</tr>
<tr>
<td>APPENDIX C</td>
<td>69</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>75</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Power Consumption Data of Each PON Generation</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>Power Consumption Comparison for Different ONU Architectures</td>
<td>17</td>
</tr>
<tr>
<td>3.2</td>
<td>Wake-up Overhead Comparison for Different ONU Architectures</td>
<td>17</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Structure of PON</td>
<td>1</td>
</tr>
<tr>
<td>2.1</td>
<td>Classification for Energy-Saving Solutions in PON</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Cyclic Sleep Handshake Protocol</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Receiver Architecture of Current ONUs</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Sleep Control Circuit for ONU</td>
<td>18</td>
</tr>
<tr>
<td>3.3</td>
<td>Performance Comparison of Sleep Modes Against Sleep Time</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Performance Comparison of Sleep Modes Against the Number of ONUs</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>FPGA Development Process</td>
<td>24</td>
</tr>
<tr>
<td>4.2</td>
<td>The SFP HSMC Card</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Testbed Picture</td>
<td>28</td>
</tr>
<tr>
<td>4.4</td>
<td>The Layered Architecture of Cycle Sleep PON System</td>
<td>29</td>
</tr>
<tr>
<td>4.5</td>
<td>Format of the Control Message</td>
<td>30</td>
</tr>
<tr>
<td>4.6</td>
<td>Output Data of OLT (a) Overall Output and (b) Output Details</td>
<td>31</td>
</tr>
<tr>
<td>4.7</td>
<td>Data Received at ONU1, (a) Overall DS Data at ONU1, (b) Control Message for ONU1, and (c) Control Message for ONU2</td>
<td>32</td>
</tr>
</tbody>
</table>
A passive optical network (PON) is a telecommunications network that has a point to multipoint network architecture as depicted in Figure 1.1.

PON does not require outside plant electronics. It uses a passive optical splitter instead of placing an Ethernet switch at the outside plant. In the downstream, the splitter divides the light sending from the Central Office (CO) and then broadcasts it to all Optical Network Units (ONUs). In the upstream, the splitter combines the light coming from ONUs, and or other active devices in the passive optical network [1].

**Figure 1.1** Structure of PON.
PON consists of three main parts [1]:

- Optical Line Terminal: The OLT at the service provider’s central office provides the interface between PON and the backbone network.
- Optical Network Unit: The ONU provides the service interface to end users. Therefore, they are located close to the end users.
- Optical Distribution Network: The ODN in PON connects the OLT at the central office to the ONUs close to the end users by using optical fibers and splitters. The ODN usually forms a tree structure with the OLT as the root of the tree and the ONUs as the leaves of the tree.

A PON is characterized by a simple point-to-multipoint topology, low-cost implementation, and relative ease of deployment, thus making it the most flexible, scalable, and future-proof optical access technology. A major factor in the success of PON is the ability to share the underlying network resources, such as physical fiber plant, communications channel capacity, and frequency spectrum among its subscribers.

Depending on the data multiplexing scheme, PONs can be divided into three types [2]: Time division multiplexing (TDM) PON, Wavelength division multiplexing (WDM) PON, and orthogonal frequency division multiplexing (OFDM) PON. In TDM PON, data transmission is divided into time slots [3], and traffic from/to multiple ONUs are TDM multiplexed onto the upstream/downstream wavelength. WDM PON efficiently exploits the large capacity of optical fibers [4], and it increases capacity by utilizing optical devices with multi-wavelength provisioning capability compared to TDM PON such as Ethernet PON [5]. OFDM PON [6] employs a number of orthogonal subcarriers to transmit upstream/downstream traffic.
1.1 Evolution of PON

Nowadays, PON has emerged as the most successful and widely deployed broadband technology. Owing to its potential to meet the tremendous bandwidth requirements, several PON technologies have been standardized and widely deployed over the last decade [7].

A. 1G-EPON

1G-EPON is the first generation of EPON, specified by IEEE 802.3ah; it provides bidirectional 1-Gb/s links using 1490-nm wavelength for downstream and 1310-nm wavelength for upstream direction, with 1550 nm reserved for future extensions or additional services, such as analog video broadcast. EPON uses the same MAC found in any IEEE 802.3 (Ethernet) compliant devices. The point-to-multipoint connectivity is supported by the multipoint control protocol (MPCP), which uses standard Ethernet frames generated in the MAC layer. The adaptation of low-cost optics and flexible bandwidth allocation has greatly spearheaded the mass deployment of 1G-EPON systems.

B. 10G-EPON

In 2009, 10G-EPON, a successor of 1G-EPON, was standardized by the IEEE 802.3av task force. This technology is currently being tested by various network operators in preparation for commercial deployments. 10G-EPON supports symmetric 10-Gb/s downstream and upstream, and asymmetric 10-Gb/s downstream and 1-Gb/s upstream data rates [2]. 10G-EPON is compatible with legacy 1G-EPON and can coexist on the same fiber plant. To lower the cost of 10G-EPON implementations, a balance between performance of optical transceivers and complexity of electronics has been considered. To extend the power budget while keeping the optical transceiver parameters relaxed, the 10G-EPON specifications include a mandatory forward error correction (FEC) encoding,
which also helps reduce the deployment cost. Considering the high capacity and low-cost implementation possibilities of 10GEPON, it could be the de facto broadband solution in foreseeable future.

C. GPON

GPON was developed by the ITU-T as the G.984 series of Recommendations. The focus of this work was to develop a universal PON architecture which is able to deliver a mix of variable-size frames, ATM cells, and native TDM traffic. GPON supports asymmetric data rates of 2.488 Gb/s downstream and 1.244 Gb/s upstream. The GPON architecture supports a two-wavelength Coarse Wavelength Division Multiplexing (CWDM) scheme similar to EPON. An additional downstream wavelength is allocated for distribution of analog video services.

D. XG-PON

XG-PON architecture has been recently standardized in ITU-T (the G.987 series of Recommendations) [7, 8]. It supports coexistence with GPON on the same fiber plant and provides 10-Gb/s downstream and 2.5-Gb/s upstream data rates.

1.2 Power Consumption in Current PON

Energy efficiency in communications networks is currently drawing much attention. Access networks (both fixed and mobile) are the major contributors to current communications network energy consumption [9]. This is mainly attributed to the large number of involved elements (i.e., the customer premise equipment). As a replacement of digital subscriber line (DSL)-based wired access networks with optical access systems,
PONs have the potential of reducing the energy consumed by access networks. Nevertheless, it is still desirable to further reduce energy consumption of PONs.

Within PONs, ONUs are customer premise equipment, and they are the energy hungriest devices. Indeed, the ONUs are numerous and always on, and often lead to the low utilization of PON capacity. It has been shown that over 65% of the total PON power consumption is contributed by ONUs [10]. In other words, ONUs are the major target for energy saving in PONs. By designing a proper scheme to turn off the idle ONUs, the energy efficiency will be further improved.

Typically, an ONU consists of a transceiver and an electronic circuitry called System on Chip (SoC), which implements the medium access control (MAC) layer functions. The transceiver blocks consist of two components: optical and electronic components.

The optical components are:

- A transmitter, typically a Fabry-Perot (F-P) laser, to optically transmit data to the OLT in the upstream direction.
- An avalanche photo-diode (APD), to receive data from the OLT, and to convert optical signals to electrical signals.
- A WDM coupler to couple downstream and upstream wavelengths into a single optical fiber.

The electronic part of the transceiver includes:

- A burst mode laser driver (BM-LD) to drive the F-P laser for upstream transmission.
- A trans-impedance amplifier (TIA) to translate and amplify downstream photocurrent into voltage.
- A limiting amplifier (LA) to reshape the voltage coming from the TIA.
- A continuous mode clock and data recovery (CDR) system.
Several implementations of the ONU transceiver are available with different degrees of integration and thus different power consumption levels. The main functional block of the electronic circuitry is the serializer-deserializer (SERDES), which converts serial signals to parallel signals and vice versa. In this work, the remaining functions of the SoC are included in a back-end electronic circuit.

The ONU power consumption data can be obtained from component data sheets, research papers, and standards [11]. Power consumption data for EPON, GPON, 10GEPON, and XG-PON are summarized in Table 1.1. Data have been collected by analyzing the data sheets available online from tens of vendors. As shown in Table 1.1, the CDR and the SERDES consume the highest power in both EPON and GPON, i.e., more than 80 percent of the power of the entire ONU front-end. A similar behavior is expected for 10G-EPON and XG-PON. However, the ONU receiver and transmitter generally share the SERDES. Moreover, the collected data confirm that about 60–70 percent of the overall ONU power consumption is attributed to the ONU transceiver and back-end electronic circuit [12].
Table 1.1 Power Consumption Data of each PON Generation

<table>
<thead>
<tr>
<th>Receiver front-end component</th>
<th>EPON</th>
<th>GPON</th>
<th>10G-EPON</th>
<th>XG-PON</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Range</td>
<td>Avg</td>
<td>Range</td>
</tr>
<tr>
<td>APD</td>
<td>2.6</td>
<td>2-3.75</td>
<td>2.6</td>
<td>2-3.75</td>
</tr>
<tr>
<td>TIA</td>
<td>83.4</td>
<td>56-112</td>
<td>83.4</td>
<td>56-112</td>
</tr>
<tr>
<td>LA</td>
<td>121</td>
<td>89-140</td>
<td>126</td>
<td>100-165</td>
</tr>
<tr>
<td>CDR</td>
<td>545</td>
<td>540-580</td>
<td>520</td>
<td>260-790</td>
</tr>
<tr>
<td>SERDES</td>
<td>550</td>
<td>530-660</td>
<td>560</td>
<td>530-660</td>
</tr>
<tr>
<td>Total receiver front-end</td>
<td>1302</td>
<td></td>
<td>1292</td>
<td></td>
</tr>
<tr>
<td>Transceiver</td>
<td>1350</td>
<td>1100-2500</td>
<td>1500</td>
<td>1040-2250</td>
</tr>
<tr>
<td>Back-end circuit</td>
<td>2700</td>
<td></td>
<td>3150</td>
<td></td>
</tr>
<tr>
<td>Whole ONU (services)</td>
<td>6000 (Ethernet data port+IPTV)</td>
<td></td>
<td>7000 (triple play + multicast video)</td>
<td></td>
</tr>
</tbody>
</table>


The rest of this thesis is organized as follows. Chapter 2 discusses the current solutions for energy saving in PON and existing sleep-control algorithms. Chapter 3 details the improved scheme and describes how the ONU receiver architecture can affect the synchronization process. Furthermore, simulation results are conducted to evaluate the energy saving performance of the existing and improved ONU architectures. Chapter 4 provides an experimental implementation of the scheme described in Chapter 3 in a multi-ONU testbed. This chapter provisions the main source code of the implemented PON system. APPENDIX A is the top-level code, which defines the logic and entrance of the whole project, APPENDIX B provides the code used to generate data, and APPENDIX C is to extract control messages and valid data.
In order to decrease the energy consumption, many solutions have been proposed to put ONUs into the sleep mode when the ONU does not have downstream or upstream traffic. Ideally, an ONU is desired to stay in the sleep mode with low power consumption when the ONU does not have traffic, and to switch back into the active mode when traffic of an ONU arrives.

2.1 Solutions for Saving Energy in PON

This section provides a classification of the existing solutions, which have been proposed by standardization authorities and academia for implementing energy-efficient optical access networks. As shown in Figure 2.1, the current solutions can be mainly classified into three categories: physical layer, data link layer, and hybrid solution. It can be inferred from the figure that both the physical layer and datalink layer solutions can be further divided into two sub-categories.
Physical layer solutions [13] focus on reducing PON energy consumption by modifying the physical layer of PON architectures without changing the upper layer protocols. They can be further classified into device-oriented and service-oriented solutions. Device-oriented solutions target at lowering energy consumption of the physical devices. Service-oriented solutions try to improving the performance of the services provided by the physical layer in order to enable upper layer solutions. Physical layer solutions are often utilized in combination with data link layer solutions to implement hybrid solutions.

Data link layer solutions [13, 14] target the data link layer of the IEEE 802.3 architecture (i.e., the medium access control MAC layer) or the transmission convergence (TC) layer in GPON (ITU-T Recommendation G.984.3) and XG-PON (ITU-T Recommendation G.987.3). They are based on the possibility of switching network
elements to a low power mode (e.g., sleep mode). In GPON, similar functionalities are also
defined in the ONU management and control interface layer (OMCI) (ITU-T Recommendation 984.4 for GPON and ITU-T Recommendation G.988 for XG-PON).
They can be further divided into MAC control and traffic scheduling solutions. Most of the
schemes are proposed to reduce the energy consumption of PONs.

Hybrid solutions [13] are those that combine physical and data link layer solutions
to reduce energy consumption. This combination approach is commonly adopted in applications.

**2.2 Existing Schemes for Energy Saving in PON**

A number of schemes have been proposed to reduce the energy consumption of the ONUs.
These proposed energy saving schemes mainly belong to datalink layer solutions which are
further divided into MAC control and traffic scheduling solutions. The first class tries to
design a proper MAC control scheme to convey the downstream (DS) queue status to
ONUs that always involves a handshake process, while the second class focuses on
investigating energy-efficient traffic scheduling schemes [15].

The Service Interoperability in Ethernet Passive Optical Networks (SIEPON)
standard described in [16] illustrates mechanisms and protocols for reducing the ONU
power consumption. This standard supports two power saving modes. In the Tx mode, the
ONU disables only the transmit data path, whereas in the TRx mode, both transmit and
receive data paths are disabled. In the mechanism, sleep cycles are established based on
mutual consent of the OLT and ONU. It is quite similar to the scheme described in [17],
which details the negotiation process between the OLT and ONUs.
Sleep and Periodic Wake-up (SPW) mechanism is proposed in [17]. In the cyclic sleep mode, the ONU transceiver is switched to the sleep mode. During a sleep period, DS traffic is buffered at the OLT and upstream (US) traffic is buffered at the ONU. The most important tasks of a cyclic sleep control mechanism are to determine sleep related decisions including when an ONU is switched to sleep and for how long the ONU can sleep to minimize energy consumption without violating Quality of Service (QoS) constraints. The decisions can be made by either the OLT alone or both the OLT and the ONU based on information of one or both transmission directions. The sleep time can be constant or variable depending on the implemented sleep control. The cyclic sleep is depicted in Figure 2. A sleep request (Sleep req) is always originated by the OLT, based on a traffic condition. The sleep request contains a desired sleep time. When the sleep request is sent, the OLT stops transmitting DS frames, stores them into a DS buffer and waits for a response from the ONU. Upon reception of the Sleep req message, depending on the implemented sleep triggering method, the ONU either positively acknowledges with an ACK message and goes to sleep, or it negatively acknowledges with a NACK and remains active. In case of the ACK, the ONU stores all the incoming US traffic from users in an US buffer until the expiration of the sleep duration. Upon waking up, the ONU first send a bandwidth request to the OLT and wait for a response. If it receives the wake up message, it can send the buffered data based on the allocated bandwidth. Once the OLT receives the request, depending on the DS traffic condition, the OLT responds with either an Awake message to force the ONU to wake up to receive DS traffic or a Sleep message to allow the ONU stay asleep in a new sleep period.
Figure 2.2 Cyclic Sleep Handshake Protocol (adopted from Figure 1 of [6]).

Yan et al. [18] proposed the upstream centric schemes (UCS) and the downstream centric schemes (DCS), and evaluated two energy management mechanisms. In the UCS scheme, the status of the ONU highly depends on the US traffic. The ONU sleeps outside its assigned US bandwidth allocation and during the sleep time US and DS traffic transmission are stopped. In the DCS scheme, the ONU must be awake during its US bandwidth allocation and when the OLT schedules DS transmission for it. The DS transmission scheduling depends on DS traffic only, and thus it does not need to be synchronized with US scheduling.

Zhang et al. [15] proposed a simple and efficient sleep control scheme to tackle the downstream challenge. First, they set certain DS traffic scheduling rules at the OLT and let the rules be known to the ONUs. Since an ONU possesses the information of the DS traffic
scheduling rules, it can infer its current DS queue status based on historical arrival DS traffic. Second, according to the inferred queue status, ONUs make their own sleep decisions based on some sleep control rules. These sleep control rules implemented at the ONU side is also known to the OLT. Third, based on the sleep control rules, the OLT is aware the status of the ONUs, and buffers the incoming DS traffic of asleep ONUs accordingly. Essentially, there are four key components of the sleep control scheme: the DS traffic scheduling rules, the DS queue inference at ONUs, the sleep control rules, and the ONU sleep status inference at the OLT.
CHAPTER 3  
CYCLE SLEEP TRIGGERING ALGORITHM AND A NEW SYNCHRONIZATION ARCHITECTURE

In this chapter, the Cycle Sleep scheme is described and evaluated. The performance of the current ONU receiver architecture and the improved ONU receiver architecture are compared under the Cycle Sleep scheme.

3.1 Cycle Sleep Operation

Before discussing the operating mechanism of cyclic sleep, we need a clear definition of Transmission Cycle. In this thesis, a transmission cycle means a specific time period during which every ONU is assigned some time slots to complete its transmission.

\[ T_{cycle} = \sum_{k=1}^{N} T_{slot}^{k} \]  

(3.1)

where \( T_{slot}^{k} \) indicates the time for the k-th ONU to receive data.

The cycle sleep considered in this paper can be described as follows. At beginning of each Transmission Cycle, the OLT schedules the DS traffic to every ONU based on the incoming traffic and application requirements. Then, the scheduled information is packaged as a control message and broadcasted to all ONUs. The format of the control message is predefined in the PON system and known to the OLT and all ONUs, which will be discussed in Section 4.3.2. Upon receiving the control message, ONUs extract the scheduling information and so they know when they need to wake up in this Transmission Cycle. Every ONU easily knows when to sleep and how long it can sleep, and acts in
accordance with the schedule. In the sleep period, the OLT stops transmitting DS frames to this ONU, and will store these frames into a DS buffer for the next Transmission Cycle.

The sleep time of each ONU could be constant or variable. The constant $T_{\text{sleep}}$ is predefined regardless of DS traffic conditions while the variable $T_{\text{sleep}}$ is defined by the OLT according to the DS estimated transmitting time.

### 3.2 ONU Receiver Architecture

Most of the studies conducted within both the research community [19] and working groups in the standard bodies [20] build upon the idea of allowing PON network elements, specifically ONUs, to switch to the sleep mode when it is idle for saving energy. However, to implement the ONU sleep mode, some issues must be addressed. In the current TDM PON, ONUs synchronize their local clock by recovering it from the OLT continuous downstream traffic through the CDR circuit [21, 22]. Therefore, if an ONU stops receiving DS transmission, the clock synchronization is lost and a period of time is needed to resume the transmission. Moreover, the absence of synchronization among different ONUs clocks might cause upstream data collision at the OLT [23].

This section first describes the synchronization process in the sleep-enabled PON system, and further introduces drawbacks of the current ONU receiver architecture. An improved ONU receiver architecture is then proposed that can significantly reduce the clock recovery time in the wake-up process.

#### 3.2.1 Current ONU Receiver Architecture and Sleep Mode

Figure 3.1 shows the receiver architecture of current ONUs. In this architecture, the received optical signal is first converted into the photocurrent signal through a signal
converter, typically an avalanche photodiode. The amplifier (TIA and LA) amplifies the electrical signal before sending it into the clock and data recovery circuit, where TIA translates and amplifies downstream photocurrent into voltage and LA reshapes the outgoing voltage from the TIA. The CDR forwards the recovered data and clock to the de-serializer, which then converts serial signals to parallel signals and sends them to the lower speed digital circuit for further processing. The digital circuit varies depending on different applications [24].

![Figure 3.1 Receiver Architecture of Current ONUs.](image)

In the current model, the sleep control is added before the signal converter, and thus the entire analog circuit is turned off during the sleep period. When the ONU wakes up from the sleep mode, the current ONU architecture uses CDR to recover the OLT clock. Significant time is required for CDR to recover the OLT clock, i.e., 2-5ms [25]. As shown in Table 3.1, the long recovery time ($T_{recovery}$) significantly increases the wake-up overhead that degrades the achievable energy saving performance.
In this architecture, when an ONU enters its sleep mode, the entire analog circuit and part of the digital circuit are turned off. Some parts of the digital circuit must be left ON, such as the clock and volatile memory. Thus, the ONU still consumes some power during the sleep status. Table 3.1 summarizes the expected power consumptions during the active and sleep mode for the current ONU. All power consumption data are extracted from a list of well-known PON component vendors [22, 23, 25].

### Table 3.1 Power Consumption Comparison for Different ONU Architectures

<table>
<thead>
<tr>
<th></th>
<th>Front-End Analog Circuit</th>
<th>Back-End Digital Circuit</th>
<th>Total Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>APD</td>
<td>TIA</td>
<td>LA</td>
</tr>
<tr>
<td><strong>Active</strong></td>
<td>N/A</td>
<td>100mW</td>
<td>100mW</td>
</tr>
<tr>
<td><strong>Current</strong></td>
<td>Off</td>
<td>Off</td>
<td>Off</td>
</tr>
<tr>
<td><strong>Improved</strong></td>
<td>On</td>
<td>On</td>
<td>On</td>
</tr>
</tbody>
</table>


### Table 3.2 Wake-up Overhead Comparison for Different ONU Architectures

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Clock Recovery Time</th>
<th>Max Synchronization Time</th>
<th>Total Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Current</strong></td>
<td>2-5ms</td>
<td>125us</td>
<td>5.125ms</td>
</tr>
<tr>
<td><strong>Improved</strong></td>
<td>none</td>
<td>125us</td>
<td>125us</td>
</tr>
</tbody>
</table>


#### 3.2.2 The Improved Architecture

Figure 3.2 shows the improved ONU architecture. In this architecture, an additional sleep mode control circuit is placed before de-serializer instead of before the signal converter. Other parts of the analog circuit remain the same as in the current ONU. Different from the current architecture, the CDR circuit is always ON to keep synchronizing with the OLT while the de-serializer is turned off when the ONU enters the sleep mode. The sleep control block calculates and sends the sleep time duration to the decision block. The decision block
chooses the counter path and sets the counter to time the sleep period. When the counter expires, it sends a TIMEOUT signal to the sleep control block. Then, the sleep control block forwards the signal to the decision block, which switches the recovered clock and data to the DMUX to resume the receiving mode.

**Figure 3.2** Sleep Control Circuit for ONU.

The advantage of the improved ONU is to eliminate clock recovery time using counter path. Although the CDR circuit is on during the sleep mode, it only requires small incremental power consumption as compared to the current architecture. This is a significant outcome, because reducing overhead time helps saving energy especially in the condition that needs frequent change of the status.

### 3.3 Energy Saving Performance Analysis

The energy saving performance of the two ONU receiver architectures are analytically computed and compared in this section. Energy consumed by an ONU can be determined as the sum of energy spent in the active mode plus that in the sleep mode, shown as Equation (3.2). Given Transmission Cycle $T_{\text{cycle}} = T_{\text{active}} + T_{\text{sleep}}$,

$$E_{\text{ONU}} = T_{\text{active}} * P_{\text{active}} + T_{\text{sleep}} * P_{\text{sleep}}$$ (3.2)
Where the values of $P_{\text{active}}$ and $P_{\text{sleep}}$ are retrieved from Table 3.1 in the analysis.

The following parts present the energy saving performance for two ONU architectures under two different traffic scenarios.

The first scenario considers traffic from the perspective of a single ONU, where the ONU periodically wakes up for a fixed period of time $T_{\text{active}}$. In this scenario, each ONU is assigned 2ms for receiving data. $T_{\text{sleep}}$ is parameterized and $T_{\text{active}} = 2\text{ms} + T_{\text{overhead}}$. $T_{\text{overhead}}$ is extracted from Table 3.1.

The second scenario considers traffic for all ONUs. The traffic is scheduled using fixed TDM, and the slot size $T_{\text{slot}}$ is assumed to be the same for all ONUs $T_{\text{slot}} = 2\text{ms}$. Each $ONU^k$ turns on to receive data for a fixed period of time, and therefore the active time for $k$-th ONU can be defined by Equation (3.3).

$$T_{\text{active}}^k = T_{\text{slot}}^k + T_{\text{overhead}}$$  \hspace{1cm} (3.3)

The definition for TDM cycle is defined by Equation (3.1).

In this scenario, the number of ONUs in system varies, and $N$ is parameterized in the analysis. The amount of time that an ONU spends in the sleep mode in one cycle can be determined according to Equation (3.4).

$$T_{\text{sleep}} = T_{\text{cycle}} - T_{\text{active}} = (N-1)T_{\text{slot}} - T_{\text{overhead}}$$  \hspace{1cm} (3.4)
In the subsequent analysis, the energy saving performance is evaluated by using the percentage of energy savings $\eta_{\text{savings}}$, which is calculated as follows:

$$\eta_{\text{savings}} = (1 - \frac{E_{\text{ONU}}}{T_{\text{cycle}} \cdot P_{\text{active}}}) \cdot 100\%$$  \hspace{1cm} (3.5)

Figure 7 demonstrates the results of the analysis for a single ONU. The current ONU architecture saves more energy when the sleep time is much longer than the active time. This is expected because the current architecture has less power consumption in the sleep mode than the improved one. On the other hand, the improved ONU architecture has higher performance than the current one when the sleep time is less than 25ms. This is because when the sleep time is reduced, the overhead time for the current ONU becomes a significant factor in ONU energy consumption.

The finding indicates that the current ONU design provides significant power saving when the ONU goes to sleep for a long period of time. However, in some applications, an ONU does not know about its future traffic demand and it needs to wake up more often to be aware of the traffic. In these situations, our improved architecture is more desirable. In Figure 3.3, for example, the improved ONU architecture provides more than 10% energy saving when the sleep time is between 2ms to 15ms.
Figure 3.3 Performance Comparison of Sleep Modes Against Sleep Time.

Under a TDM scheduled traffic scenario, an ONU receives traffic periodically and waits for a full TDM cycle in which all the other ONUs receive traffic. In this analysis, the improved ONU architecture is compared with the current one. The analysis considers energy saving performance when the number of ONUs ranges from 1 to 32. The TDM slot is set to 2ms. Figure 3.4 shows that the improved ONU architecture has better performance when PON has a less number of ONUs. For 17 ONUs, the improved architecture has the same performance as the current one. Especially when the number of ONUs is less than 13, the energy saving difference is more than 10%. Based on the above discussion, the improved architecture has excellent performance if the system contains less than 17 ONUs when time slot is 2ms. Note, however, that a PON system is normally implemented with the number of ONUs in the 2 to the $n$th power, where $n$ is a positive integer.

More significantly, the results also show that the current ONU architecture fails to save any energy at some specific time (i.e., in the simulation, when the number of ONUs is less than 5). This is because the overhead time exceeds the length of the TDM cycle, which prevents the ONU from switching into the sleep mode.
The sleep time for an ONU in TDM-PON is not expected to be too long when TDM traffic is involved. Essentially, an ONU might need to wake up shortly after it switches to the sleep mode. Furthermore, the inter-ONU scheduling cycle in TDM-PON typically lasts only a few milliseconds to satisfy the quality of service (QoS) requirement incurred by delay sensitive applications. As a result, the improved ONU could effectively reduce the ONU energy consumption by using the sleep mode mechanism under real-time traffic.
CHAPTER 4
HARDWARE IMPLEMENTATION

This section first lists the required steps in the FPGA development process. Then, the FPGA boards used in the implementation are introduced, and the specific interfaces and modules are explained. Section 4.3 describes the system architecture in the form of a layered stack with major functioning blocks, details the format of control messages, and implements the cycle sleep scheme in a multi-ONUs testbed.

4.1 FPGA Development Process

Figure 4.1 shows the flow-chart of an FPGA development process.

- Plan phase

An FPGA development process starts with system requirements, which are decomposed into lower levels of the design, i.e. sub-systems.

Legacy requirements can often be leveraged for the new design to minimize the time spent in this phase. Requirements are vital to the success of any project. This is often a balancing act to find what is “good enough” to move the design forward. It is a mistake to start sub-system level designs without requirements and generally leads to a conflict between functional teams and longer overall design cycles.

Language selection is a key of this phase. VHDL or Verilog for design is a typical choice. Verification is also decided at this phase. During this phase, selecting a third party Intellectual Property (IP) can help complete the design faster. This will also open the door for using standard verification IP to streamline the verification activity.
Figure 4.1 FPGA Development Process.
The FPGA requirements provide the foundation on how the design is partitioned and how it is tested. Also, an appropriate hardware architecture or proper partitioning of the design can optimize design re-use, complexity, power, quality and reliability.

Generally, the test plan provides a list of requirements and a plan to test them.

- **Execute phase**

This is the coding phase of the design. Both the design and the testing environment are created. This should be the shortest phase if the PLAN phase is as complete as possible.

VHDL and Verilog are the two main languages used to develop Register-Transfer Level (RTL). VHDL is more structured but more involved in programming and Verilog is less structured and can cause unforeseen behavior in the code.

Constraint verification is needed to create meaningful random simulation stimulus for a design that minimizes the need to stress the design through directed test patterns. It also minimizes the effort to create test patterns and reduces the size and effort of managing the test suite.

- **Verification phase**

The Verification phase is divided into two distinct stages. The first phase is Functional Verification followed by the Lab Verification phase. This feature provides leverage in a couple of ways. First, using simulations as the primary form of debug provides a shorter loop to fix the inevitable shortcomings of simulation problems, and finding these problems in the simulation stage rather than the in-circuit verification stage greatly reduces the development time. Second, the two-stage verification allows the project to schedule staged hardware for the software organization.
The reason for simulation centric flow is two-folded. First, it is much easier to debug a design in simulation than the hardware. Second, it reduces the overall churn because software is testing on verified hardware and hardware can focus on hardware verification rather than responding to problems found during software testing.

In-circuit verification work is always necessary in the design process. However, putting the bulk of the effort into simulation makes the hardware work faster and more efficient. Industry has found that for every month spent in the simulation, you save about two to three months of in-circuit testing.

4.2 Introduction of EDA Tool and Testbed

The Electronic Design Automation (EDA) tool used for synthesizing and configuring the FPGAs is Quartus II 14.0.

The cycle sleep scheme is implemented in two Altera Transceiver Signal Integrity Development Kits equipped with Arria V GX FPGA (i.e., featured device 5AGXFB3H4F35C5ES or 5AGXFB3H4F40C5NES).

The FPGA board contains High Speed Mezzanine Card (HSMC) interface. This physical interface provides eight channels of 6.5536 Gbps-capable transceivers. The HSMC specification defines the electrical and mechanical properties of a high speed mezzanine card adapter interface for FPGA-based motherboards. This specification should allow for the design of interoperable motherboards and add-on cards by different manufacturers that can interoperate and utilize the high-performance I/O features found in today’s FPGA devices. In this work, we expand the functionality of the board through the addition of SFP HSMC (Small Form-Factor Pluggable) daughtercards.
The SFP HSMC card (Figure 4.2) is a hardware platform for evaluating the interoperation of Altera FPGA, specifically Stratix IV GX, Arria V GX, and Arria II GX, with generic SFP modules. The optical modules that are of particular importance are Gigabit Media-Independent Interface (SGMII) Ethernet, Fiber channel, Common Public Radio Interface/Open Base Station Architecture Initiative (CPRI/OBSAI), and Synchronous optical networking (SONET). Furthermore, the SFP HSMC card is intended to implement both telecommunications and data communications applications. The electrical and optical specifications of SFP should be compatible with those enumerated in the appropriate standards (i.e., the IEEE 802.3z Gigabit Ethernet standard and the ITU G.957 Synchronous Digital Hierarchy standard).

![Figure 4.2 The SFP HSMC Card.](http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=342)

Testbed in this work includes three parts: one OLT and two ONUs. Figure 4.3 shows a picture of the real testbed. The Optical Distribution Networks (ODN) is not implemented because the experiments aim at testing the cycle sleep mode function which is independent from the ODN and implemented in electronics. The output and input of the OLT and ONUs can be directly interconnected by means of Active Optical Direct Attach Cable.
4.3 Emulation of the PON Testbed

4.3.1 Layered Architecture

Figure 4.4 is the layered architecture of the cycle sleep PON system. The OLT and ONU functions partly use the IP cores, such as altera_xcvr_custom_phy. This section mainly describes how to implement the sleep scheme by some major blocks, whereas the MAC and physical layer are out of the scope of this work.

The OLT and ONU block diagrams both contain Sleep Control blocks. The OLT Sleep Control determines the active time interval of each ONU according to the DS traffic. Meanwhile, the ONU Sleep Control part calculates its estimated sleeping time.
Both data frame and control message are generated by the OLT Generator as shown in Figure 4.4, implemented within the FPGAs. The DS FIFO is a buffer used to store DS data frames during the ONU inactive time, i.e., when the ONU sojourns in the sleep state. The ONU Monitor receives data frames and collects the relevant statistics for the DS transmission. The ONU Extractor extracts the received frames to hunt for control messages and forwards the extracted headers, such as sleep time, to the corresponding Sleep Control.

![Figure 4.4 The Layered Architecture of the Cycle Sleep PON System.](image)

### 4.3.2 Control Message Format

The control message format depicted in Figure 4.5 includes three segments. *Transmission header* is a specific 8-byte opcode for the start of a transmission. In this work, we set the
header to be four AA55h, which can be predefined according to the application. The 16-bit opcode of the *Identification No.* field is used to identify the destination ONU of the following data. The value ranges from 0001h to FFFFh, which satisfies the requirements for various standards. The *Sleep period notification* segment contains a 16-bit field that indicates the desired sleep time specified by the OLT in terms of the number of data frames to be transmitted. Data segment is the valid data to be transmitted.

![Figure 4.5 Format of the Control Message.](image)

### 4.3.3 Data Analysis

Because of the hardware resource limitation, two ONUs are used for implementing the cycle sleep scheme. SignalTap is used to capture and display real time signals in the FPGA design.

Figure 4.6(a) shows the data transmitted by the OLT and Figure 4.6(b) shows the control message in magnification, that in one cycle. In Figure 4.6(a), we mark the Transmission cycle of our PON system, and we can observe the value of each bit in Figure 4.6(b).
Data captured on ONU1 are depicted in Figure 4.7 (a), (b) and (c).

Figure 4.7 (a) is an overall data view of ONU1. The DS data are a continuous signal emitting from the OLT, while the ONU1 shows discontinuous reception. In comparison between R_rx_data_stream in Figure 4.7 (b) and the tx_data in Figure 4.6 (b), it is easy to see that the two signals are essentially the same.
Figure 4.7 Data Received at ONU: (a) Overall Downstream Data at ONU1, (b) Control Message for ONU1, and (c) Control Message for ONU2.

Description of parameters used in the emulation:

- **R_rx_data_stream**: the DS data stream from the OLT.
- **R_fetch_valid**: a flag to indicate the beginning of valid data.
- **R_rx_fetch_en**: enable signal; pulled high when the ONU begins to receive data.
- **R_rx_fetch_data**: data received from the OLT.
- Compare_data: reference data used to check whether errors exist.
- Error_flag_reg: pulled high when an error exists.
- Error_cnt: a counter to store the total number of errors.

Figure 4.7 (b) shows the detailed information of the captured data. It can be seen that the control message is AA55h AA55h AA55h AA55h 0001h 0064h. The four consecutive AA55h implies the sign for transmission, and the following 0001h indicates the destination ID is 1, which represents ONU1, and 0064h means that the following one hundred 16-bit data are going to be transmitted.

Figure 4.7 (c) demonstrates the end of transmission to ONU1 following the control message for the next transmission. R_fetch_valid is drawn to low to illustrate the end of valid data for the current destination. R_rx_fetch_data register keeps the last data in preparing the switching to the sleep mode. Error_flag_reg is driven low if an error occurs, and Error_cnt counts the number of errors. The values of these two parameters are 0 if this transmission is error free.

According to the definition of the transmission cycle and the observed results, we obtain

\[
T_{cycle} = \sum_{k=1}^{2} T_{slot}^k = T_{slot}^1 + T_{slot}^2 = T_{transmit(0064h+0032b)\text{data}} = T_{(150)}
\]

In this equation, \( k \) is the sequence number of ONU. In this implementation, the sleep time of each ONU is constant. Based on the emulation and assuming \( t \) is the time needed for transmitting one 16-bit data, we can calculate the percentage of energy savings as follows,
\[ \eta_{\text{savings}} = (1 - \frac{E_{\text{ONU}}}{T_{\text{cycle}} \times P_{\text{active}}}) \times 100\% = \left( 1 - \frac{100t \times 3.85 + 50t \times 1.28}{150t \times 3.85} \right) = 77.75\% \]

It can be concluded in this implementation that the cycle sleep scheme can save up to 77.75\% of energy consumption of one ONU.
CHAPTER 5
CONCLUSIONS

This thesis describes the wake-up process and compares the power consumption in the active mode and sleep mode for ONU with the current receiver architecture. An improved ONU architecture has been proposed to allow ONUs to switch to the sleep mode and quickly recover the OLT clock. The effects of the sleep mode on the energy saving performance are analytically computed and compared for the two architectures. The simulation results show that the improved architecture significantly reduces the clock recovery overhead of the current architecture as the ONU wakes up from the sleep mode.

This work further implements the cycle sleep scheme on a multi-ONU testbed based on the improved ONU architecture. The emulation results illustrate that the improved architecture can effectively reduce energy consumption by using the proper sleep mode approach under live traffic.
APPENDIX A

TOP-LEVEL CODE

Top-level code is the entrance of the whole project, which combines all the sub-functional modules by designing a proper logic sequence.
`include "definition.v"

module custom_PHY_test_top(
    input clkintop_125_p,
    input cpu_resett, //2.5V, CPU Reset Pushbutton, comment out if dev_clrn function is set

    //
    //HIGH-SPEED-MEZzanine-CARD interface -----------//198 pins
    //xcvr: 8
    //IO: 85
    output [3:0] hsma_d,
    //Bank 1 (transceivers)
    output [1:0] hsma_tx_p, //1.5V PCML, HSMA Transmit Data
    input [1:0] hsma_rx_p, //1.5V PCML, HSMA Receive Data-req's OCT
    //
    input refclk2_qr1_p, //1.5-V PCML, default 125MHz
    //input refclk3_qr1_p, //1.5-V PCML, HSMC CLKIN2 (differential)
    //user LED
    //IO: 6
    input [3:0] user_dipsw,
    output [3:0] user_led, //2.5V, Green User LEDs
    output hsma_tx_led, //2.5V, User LED - Labeled HSMA TX
    output hsma_rx_led //2.5V, User LED - Labeled HSMA RX
);

//==============================================================
//parameters definition
//==============================================================

ifndef SIM

```
parameter `REPEAT_ALIGN_CYCLE` = 5000;

`else

parameter `REPEAT_ALIGN_CYCLE` = 40000;

`endif

parameter `ALIGN_CYCLE` = 2000;

//--------------------------------------------------------------
wire [1:0] `pll_powerdown pll_powerdown
wire [1:0] `tx_analogreset tx_analogreset
wire [1:0] `tx_digitalreset tx_digitalreset
wire [0:0] `tx_pll_refclk tx_pll_refclk
wire [1:0] `tx_serial_data tx_serial_data
wire [1:0] `pll_locked pll_locked
wire [1:0] `rx_analogreset rx_analogreset
wire [1:0] `rx_digitalreset rx_digitalreset
wire [0:0] `rx_cdr_refclk rx_cdr_refclk
wire [1:0] `rx_serial_data rx_serial_data
wire [1:0] `rx_is_lockedtoref rx_is_lockedtoref
wire [1:0] `rx_is_lockedtodata rx_is_lockedtodata
wire [1:0] `rx_seriallpbken rx_seriallpbken
wire [87:0] `tx_parallel_data tx_parallel_data
wire [127:0] `rx_parallel_data rx_parallel_data
wire [1:0] `tx_std_coreclkin tx_std_coreclkin
wire [1:0] `rx_std_coreclkin rx_std_coreclkin
wire [1:0] `tx_std_clkout tx_std_clkout
wire [1:0] `rx_std_clkout rx_std_clkout
wire [1:0] tx_std_pcfifo_full ;//tx_std_pcfifo_full.tx_std_pcfifo_full
wire [1:0] rx_std_pcfifo_empty ;//rx_std_pcfifo_empty.rx_std_pcfifo_empty
wire [1:0] rx_std_wa_patternalign; // rx_std_wa_patternalign.rx_std_wa_patternalign
wire [9:0] rx_std_bitslipboundarysel; // rx_std_bitslipboundarysel.rx_std_bitslipboundarysel
reg [1:0] rx_std_bitslip=0        ;//rx_std_bitslip.rx_std_bitslip
wire [1:0] tx_cal_busy           ;//tx_cal_busy.tx_cal_busy
wire [1:0] rx_cal_busy           ;//rx_cal_busy.rx_cal_busy
wire [279:0] reconfig_to_xcvr   ;//reconfig_to_xcvr.reconfig_to_xcvr
wire [183:0] reconfig_from_xcvr  ;//reconfig_from_xcvr.reconfig_from_xcvr
wire [1:0] tx_ready              ;
wire [1:0] rx_ready              ;
//==============================================================
wire         hw_reset_n            ;
wire         phy_mgmt_clk          ;
wire         locked                ;
reg          heartbeat_led=0       ;
reg [31:0] heartbeat_cnt=0        ;
reg [15:0] tx_data=0;
wire         tx_reset              ;
wire         rx_reset              ;
reg [31:0] tx_test_cnt            ;
reg [1:0] rx_align_done=2'b0     ;
reg [5:0] wait_cnt_0 = 6'd30     ;
reg [5:0] wait_cnt_1 = 6'd30     ;
reg          rx_data_valid_0       ;

39
reg   rx_data_valid_1    ;
reg  [15:0] rx_data_0    ;
reg  [15:0] rx_data_1    ;
reg  [15:0] test_data = 16'hacef ;
wire [15:0] tx_bus_data    ;
wire [15:0] rx_data_temp_0    ;
wire [15:0] rx_data_temp_1    ;
wire   rx_fetch_en_0    ;
wire [15:0] rx_fetch_data_0    ;
wire   rx_fetch_en_1    ;
wire [15:0] rx_fetch_data_1    ;
reg     rx_data_valid_reg0    ;
reg  [15:0] rx_data_reg0    ;
reg     rx_data_valid_reg1    ;
reg  [15:0] rx_data_reg1    ;
reg    R_rx_fetch_en_0    ;
reg  [15:0] R_rx_fetch_data_0    ;
reg    R_rx_fetch_en_1    ;
reg  [15:0] R_rx_fetch_data_1    ;
reg     compare_en_0=0    ;
reg     compare_en_1=0    ;
reg  [15:0] compare_data_0    ;
reg  [15:0] compare_data_1    ;
wire     error_falg_0    ;
reg     error_flag_reg_0    ;
```verilog
type error_cnt_0 = reg [15:0]
wire error_flag_1
reg error_flag_reg_1
reg [15:0] error_cnt_1
reg tx_active
reg [1:0] rx_active
reg [13:0] rx_align_cnt_0
reg [13:0] rx_align_cnt_1
reg tx_en
reg [15:0] tx_en_time
reg rx_valid_en
reg rx_valid_en_reg
reg [15:0] rx_valid_data_cnt_0
reg [15:0] rx_valid_data_cnt_1
reg [31:0] rx_reset_cnt=0

//==============================================================================
reset_debounced#(
  .FR (125),//125MHz
  .DELAY (15) //15ms
)reset_debounced_inst(
  .I_clk (clkintop_125_p ),
  .I_reset_in_n (cpu_resetn ),
  .O_reset_out_n (hw_reset_n )
);

sys_pll sys_pll_inst (   )
```
.refclk (clkintop_125_p ),
.rst (1'b0 ),
.outclk_0 (phy_mgmt_clk ),
.locked (locked )
);

delay_reset delay_reset_inst(
.I_clk (phy_mgmt_clk ),
.I_locked ( locked ),
.O_rst (phy_mgmt_clk_reset )
);

delay_reset delay_reset_inst_tx(
.I_clk (tx_std_clkout[0] ),
.I_locked (pll_locked[0] ),
.O_rst (tx_reset )
);

delay_reset delay_reset_inst_rx(
.I_clk (rx_std_clkout[1] ),
.I_locked (pll_locked[1] ),
.O_rst (rx_reset )
);

always@(posedge phy_mgmt_clk)
begin
  if(heartbeat_cnt==32'd62500000)
    begin
      heartbeat_cnt <= 32'd0;
    end
end
heartbeat_led <= ~heartbeat_led;
else
begin
    heartbeat_cnt <= heartbeat_cnt + 32'd1;
end
end

//==============================================================
assign hsma_d = 4'b11_11;
assign rx_serialpbken = 2'b0;
assign tx_pll_refclk = refclk2_qr1_p;
assign rx_cdr_refclk = refclk2_qr1_p;//refclk3_qr1_p;
assign tx_std_coreclkin = tx_std_clkout;
assign rx_std_coreclkin = rx_std_clkout;
assign pll_powerdown[1] = {1{pll_powerdown[0]}};
assign rx_serial_data[0] = hsma_rx_p[0];
assign hsma_tx_p[0] = tx_serial_data[0];
assign rx_serial_data[1] = hsma_rx_p[1];
assign hsma_tx_p[1] = tx_serial_data[1];
assign tx_parallel_data = {
    {25'b0,tx_data[15:8],3'b0,tx_data[7:0]},
    {25'b0,tx_data[15:8],3'b0,tx_data[7:0]}};
assign rx_data_temp_0 = {rx_parallel_data[23:16],rx_parallel_data[7:0]};
assign rx_data_temp_1 = {rx_parallel_data[64+23:64+16],rx_parallel_data[64+7:64+0]};
assign hsma_tx_led = ~rx_is_lockedtodata;
assign hsma_rx_led = ~rx_is_lockedtoref;
assign user_led[0] = ~pll_locked;
assign user_led[1] = ~& rx_align_done;

assign user_led[2] = ~(error_cnt_0) | ~(error_cnt_1) | ~error_flag_reg_0 | ~error_flag_reg_1;

assign user_led[3] = heartbeat_led;

//==============================================================

always@(posedge tx_std_coreclk[0])
begin
    tx_active <= user_dipsw[0];
end

always@(posedge tx_std_coreclk[0])
begin
    if(tx_reset)
    begin
        tx_test_cnt <= 32'd0;
        tx_data <= 16'd0;
    end
    else if(tx_active)
    begin
        if(tx_ready[0])
        begin
            `ifdef SIM
            if(tx_test_cnt[11])
            `else
                if(tx_test_cnt[27])
            `endif
            begin

if(tx_en)
    tx_data <= tx_bus_data;
else
    tx_data <= test_data;
end
else
    begin
        tx_data       <= test_data;
        tx_test_cnt   <= tx_test_cnt + 32'd1;
    end
end
else
    begin
        tx_data       <= 16'd0;
        tx_test_cnt   <= 32'd0;
    end
end
always@(posedge tx_std_coreclk[0])
begin
    if(tx_reset)
        begin
            tx_en_time <= 16'd0;
            tx_en     <= 1'b0;
        end
end
else if (tx_test_cnt[11])
    begin
        if (tx_en_time >= (REPEAT_ALIGN_CYCLE + ALIGN_CYCLE))
            begin
                tx_en_time <= 16'd0;
                tx_en_time <= tx_en_time;
            end
        else
            begin
                if (tx_en_time < REPEAT_ALIGN_CYCLE)
                    begin
                        tx_en <= 1'b1;
                    end
                else
                    begin
                        tx_en <= 1'b0;
                    end
            end
    end
end

tx_data_gen tx_data_gen_inst(
    .clk (tx_std_coreclkin[0]),
    .reset (tx_reset ),
    .en (tx_en ),
    .data_out (tx_bus_data )
always@(posedge rx_std_coreclk[0])
begin
    rx_active[0] <= user_dipsw[1];
end

always@(posedge rx_std_coreclk[0])
begin
    if(rx_reset)
    begin
        rx_std_bitslip[0] <= 1'b0;
        rx_align_done[0] <= 1'b0;
        wait_cnt_0      <= 6'd30;
        rx_align_cnt_0 <= 13'd0;
        rx_valid_data_cnt_0 <= 16'd0;
    end
    else if(rx_active[0])
    begin
        if(rx_ready[0])
        begin
            if(!rx_align_done[0])
            begin
                if(rx_data_temp_0==test_data)
                begin
                    wait_cnt_0 <= 6'd30;
                end
            end
        end
end
```verilog
//
`ifdef SIM
    if(rx_align_cnt_0[4])
`else
    if(rx_align_cnt_0[12])
`endif
begin
    rx_std_bitslip[0] <= 1'b0;
    rx_align_done[0] <= 1'b1;
end
else
begin
    rx_align_cnt_0 <= rx_align_cnt_0 + 13'd1;
    rx_std_bitslip[0] <= 1'b0;
end
end
else if(wait_cnt_0<6'd5)
begin
    rx_std_bitslip[0] <= 1'b0;
    //
    if(wait_cnt_0==6'd0)
    begin
        wait_cnt_0 <= 6'd30;
    end
    else
    begin
        wait_cnt_0 <= wait_cnt_0 - 6'd1;
    end
    //
```
rx_align_cnt_0 <= 13'd0;
end
else
begin
rx_std_bitslip[0] <= 1'b0;
wait_cnt_0 <= wait_cnt_0 - 6'd1;
rx_align_cnt_0 <= 13'd0;
end
end
else
begin
rx_std_bitslip[0] <= 1'b0;
rx_align_done[0] <= 1'b0;
wait_cnt_0 <= 4'd15;
end
end

//==========================================================
always@(posedge rx_s

td_coreclkin[1])
begin
rx_active[1] <= user_dipsw[1];
end
always@(posedge rx_std_coreclkin[1])
begin

if(rx_reset)
begin
    rx_std_bitslip[1] <= 1'b0;
    rx_align_done[1] <= 1'b0;
    wait_cnt_1       <= 6'd30;
    rx_align_cnt_1   <= 13'd0;
    rx_valid_data_cnt_1 <= 16'd0;
end
else if(rx_active[1])
begin
    if(rx_ready[1])
    begin
        if(!rx_align_done[1])
        begin
            if(rx_data_temp_1==test_data)
            begin
                wait_cnt_1       <= 6'd30;
                //
                `ifdef SIM
                if(rx_align_cnt_1[4])
                `else
                if(rx_align_cnt_1[12])
                `endif
                begin
                    rx_std_bitslip[1] <= 1'b0;
                end
            end
        end
    end
end
```vhdl
rx_align_done[1] <= 1'b1;
end
else
begin
rx_align_cnt_1 <= rx_align_cnt_1 + 13'd1;
rx_std_bitslip[1] <= 1'b0;
end
end
else if(wait_cnt_1<6'd5)
begin
rx_std_bitslip[1] <= 1'b1;
//
if(wait_cnt_1==6'd0)
wait_cnt_1 <= 6'd30;
else
wait_cnt_1 <= wait_cnt_1 - 6'd1;
//
rx_align_cnt_1 <= 13'd0;
end
else
begin
rx_std_bitslip[1] <= 1'b0;
wait_cnt_1 <= wait_cnt_1 - 6'd1;
rx_align_cnt_1 <= 13'd0;
end
```
else

begin

    rx_std_bitslip[1] <= 1'b0;
    rx_align_done[1] <= 1'b0;
    wait_cnt_1      <= 4'd15;

end

end

//==============================================================
always@(posedge rx_std_coreclkin[0])
begin

  if(rx_align_done[0])

    begin

      if(rx_ready[0])

        begin

          rx_data_valid_0 <= 1'b1;
          rx_data_0      <= rx_data_temp_0;

        end

      else

        begin

          rx_data_valid_0 <= 1'b0;

        end

    end

end
else

    begin

        rx_data_valid_0 <= 1'b0;

    end

end

rx_data_fetch#(.ch(0))rx_data_fetch_inst0(
   .clk   (rx_std_coreclkin[0] ),
   .reset (rx_reset            ),

   .rx_valid   (rx_data_valid_0 ),
   .rx_data    (rx_data_0       ),

   .rx_fetch_en (rx_fetch_en_0   ),
   .rx_fetch_data (rx_fetch_data_0)
);

//===--------------------------------------------------------------------------------===

always@(posedge rx_std_coreclkin[1])
begin

    if(rx_align_done[1])

        begin

            if(rx_ready[1])

                begin

                    rx_data_valid_1 <= 1'b1;

                    rx_data_1     <= rx_data_temp_1;

                end

        end

end
end
else
begin
rx_data_valid_1 <= 1'b0;
end
end

else
begin
rx_data_valid_1 <= 1'b0;
end
end

rx_data_fetch#(.ch(1))rx_data_fetch_inst1(
    .clk             (rx_std_coreclk[1]),
    .reset           (rx_reset            ),
    .rx_valid        (rx_data_valid_1     ),
    .rx_data         (rx_data_1           ),
    .rx_fetch         (rx_fetch_en_1       ),
    .rx_fetch_data   (rx_fetch_data_1     )
);

//==============================================================
//RX error detcter
//==============================================================

always@(posedge rx_std_coreclk[0])
begin
    rx_data_valid_reg0 <= rx_fetch_en_0;
end
rx_data_reg0 <= rx_fetch_data_0;

R_rx_fetch_en_0 <= rx_data_valid_reg0;
R_rx_fetch_data_0 <= rx_data_reg0;

rx_data_valid_reg1 <= rx_fetch_en_1;
rx_data_reg1 <= rx_fetch_data_1;

R_rx_fetch_en_1 <= rx_data_valid_reg1;
R_rx_fetch_data_1 <= rx_data_reg1;
end
always@(posedge rx_std_coreclkin[0])
begin
  if(rx_align_done[0])
  begin
    if(rx_data_valid_reg0 && ((R_rx_fetch_data_0==test_data) &&
      (rx_data_reg0==~test_data)))
    begin
      compare_en_0 <= 1'b1;
    end
  end
end
always@(posedge rx_std_coreclkin[0])
begin
  if(rx_reset)
begin
    compare_data_0    <= 16'h0;
end

if(R_rx_fetch_en_0)
begin
    compare_data_0    <= compare_data_0 + 16'd1;
end
else
begin
    compare_data_0    <= 16'h0;
end
end
assign error_flag_0   = (compare_data_0==R_rx_fetch_data_0)? 1'b0 : 1'b1;
always@(posedge rx_std_coreclk[0])
begin
    if(rx_reset)
    begin
        error_cnt_0       <= 16'd0;
        error_flag_reg_0  <= 1'b0;
    end
else if(R_rx_fetch_en_0 & error_falg_0)
begin
    error_cnt_0       <= error_cnt_0 + 16'd1;
    error_flag_reg_0  <= 1'b1;
end
else

begin

    error_flag_reg_0 <= 1'b0;

end

delimiter

//==============================================================always@(posedge rx_std_coreclkin[1])
begin

if(rx_align_done[1])

begin

    if(rx_data_valid_reg1 && (R_rx_fetch_data_1==test_data) &&
(rx_data_reg1==~test_data)))

begin

    compare_en_1 <= 1'b1;

end

end
always@(posedge rx_std_coreclkin[1])
begin

if(rx_reset)

begin

    compare_data_1 <= 16'h0;

end

if(R_rx_fetch_en_1)

begin

    compare_data_1 <= compare_data_1 + 16'd1;

end
end
else
begin
compare_data_1 <= 16'h0;
end
end

assign error_flag_1 = (compare_data_1==R_rx_fetch_data_1)? 1'b0 : 1'b1;
always@(posedge rx_std_coreclk[1])
begin
if(rx_reset)
begi
error_cnt_1 <= 16'd0;
error_flag_reg_1 <= 1'b0;
end
else if(R_rx_fetch_en_1 & error_flag_1)
begi
error_cnt_1 <= error_cnt_1 + 16'd1;
error_flag_reg_1 <= 1'b1;
end
else
begi
error_flag_reg_1 <= 1'b0;
end
end

//============================================================================
=============================================
transceiver\_native\_PHY transceiver\_native\_PHY\_inst {

/*input wire [0:0] */.pll\_powerdown (pll\_powerdown ),//pll\_powerdown.pll\_powerdown

/*input wire [0:0] */.tx\_analogreset (tx\_analogreset ),//tx\_analogreset.tx\_analogreset

/*input wire [0:0] */.tx\_digitalreset (tx\_digitalreset ),//tx\_digitalreset.tx\_digitalreset

/*input wire [0:0] */.tx\_pll\_refclk (tx\_pll\_refclk ),//tx\_pll\_refclk.tx\_pll\_refclk

/*output wire [0:0] */.tx\_serial\_data (tx\_serial\_data ),//tx\_serial\_data.tx\_serial\_data

/*output wire [0:0] */.pll\_locked (pll\_locked ),//pll\_locked.pll\_locked

/*input wire [0:0] */.rx\_analogreset (rx\_analogreset ),//rx\_analogreset.rx\_analogreset

/*input wire [0:0] */.rx\_digitalreset (rx\_digitalreset ),//rx\_digitalreset.rx\_digitalreset

/*input wire [0:0] */.rx\_cdr\_refclk (rx\_cdr\_refclk ),//rx\_cdr\_refclk.rx\_cdr\_refclk

/*output wire [0:0] */.rx\_serial\_data (rx\_serial\_data ),//rx\_serial\_data.rx\_serial\_data

/*output wire [0:0] */.rx\_is\_lockedtoref (rx\_is\_lockedtoref ),//rx\_is\_lockedtoref.rx\_is\_lockedtoref

/*output wire [0:0] */.rx\_is\_lockedtodata (rx\_is\_lockedtodata ),//rx\_is\_lockedtodata.rx\_is\_lockedtodata

/*input wire [43:0] */.tx\_parallel\_data (tx\_parallel\_data ),//tx\_parallel\_data.tx\_parallel\_data

/*output wire [63:0] */.rx\_parallel\_data (rx\_parallel\_data ),//rx\_parallel\_data.rx\_parallel\_data

/*input wire [0:0] */.tx\_std\_coreclkin (tx\_std\_coreclkin ),//tx\_std\_coreclkin.tx\_std\_coreclkin

/*input wire [0:0] */.rx\_std\_coreclkin (rx\_std\_coreclkin ),//rx\_std\_coreclkin.rx\_std\_coreclkin

/*output wire [0:0] */.tx\_std\_clkout (tx\_std\_clkout ),//tx\_std\_clkout.tx\_std\_clkout

/*output wire [0:0] */.rx\_std\_clkout (rx\_std\_clkout ),//rx\_std\_clkout.rx\_std\_clkout

/*output wire [0:0] */.tx\_std\_pcfifo\_full (tx\_std\_pcfifo\_full ),//tx\_std\_pcfifo\_full.tx\_std\_pcfifo\_full

59
transceiver_PHY_reset transceiver_PHY_reset_inst (
/*input wire */.clock (phy_mgmt_clk ),//clock.clk
/*input wire */.reset (phy_mgmt_clk_reset ),//reset.reset
/*output wire [0:0] */.pll_powerdown (pll_powerdown[0] ),//pll_powerdown pll_powerdown
/*output wire [0:0] */.tx_analogreset (tx_analogreset ),//tx_analogreset.tx_analogreset
/*output wire [0:0] */.tx_digitalreset (tx_digitalreset ),//tx_digitalreset.tx_digitalreset
/*output wire [0:0] */.tx_ready (tx_ready ),//tx_ready.tx_ready
/*input wire [0:0] */.pll_locked (pll_locked ),//pll_locked_pll_locked
/*input wire [0:0] */.pll_select (0),//pll_select_pll_select
/*input wire [0:0] */.tx_cal_busy (tx_cal_busy ),//tx_cal_busy.tx_cal_busy
/*output wire [0:0] */.rx_analogreset (rx_analogreset ),//rx_analogreset.rx_analogreset
/*output wire [0:0] */.rx_digitalreset (rx_digitalreset ),//rx_digitalreset.rx_digitalreset
/*output wire [0:0] */.rx_std_pcfifo_empty (rx_std_pcfifo_empty),//rx_std_pcfifo_empty rx_std_pcfifo_empty
/*input wire [0:0] */.rx_std_wa_patternalign (rx_std_wa_patternalign),//rx_std_wa_patternalign rx_std_wa_patternalign
/*output wire [4:0] */.rx_std_bitslipboundarysel(rx_std_bitslipboundarysel), // rx_std_bitslipboundarysel rx_std_bitslipboundarysel
/*input wire [0:0] */.rx_clkslip (rx_std_bitslip ),//rx_std_bitslip.rx_std_bitslip
/*output wire [0:0] */.tx_cal_busy (tx_cal_busy ),//tx_cal_busy.tx_cal_busy
/*output wire [0:0] */.rx_cal_busy (rx_cal_busy ),//rx_cal_busy.rx_cal_busy
/*input wire [139:0] */.reconfig_to_xcvr (reconfig_to_xcvr ),//reconfig_to_xcvr.reconfig_to_xcvr
/*output wire [91:0] */.reconfig_from_xcvr (reconfig_from_xcvr ),//reconfig_from_xcvr.reconfig_from_xcvr
);
/*output wire [0:0] */.rx_ready (rx_ready ),//rx_ready.rx_ready

/*input wire [0:0] */.rx_is_lockedtodata (rx_is_lockedtodata ),//rx_is_lockedtodata.rx_is_lockedtodata

/*input wire [0:0] */.rx_cal_busy (rx_cal_busy ) //rx_cal_busy.rx_cal_busy

);

transceiver_PHY_reconfig transceiver_PHY_reconfig_inst ( 

/*output wire */.reconfig_busy () ,//reconfig_busy.reconfig_busy

/*input wire */.mgmt_clk_clk (phy_mgmt_clk ),//mgmt_clk_clk.clk

/*input wire */.mgmt_rst_reset (phy_mgmt_clk_reset ),//mgmt_rst_reset.reset

/*input wire [6:0] */.reconfig_mgmt_address (0),//reconfig_mgmt.address

/*input wire */.reconfig_mgmt_read (0),//.read

/*output wire [31:0] */.reconfig_mgmt_readdata () ,//readdata

/*output wire */.reconfig_mgmt_waitrequest () ,//waitrequest

/*input wire */.reconfig_mgmt_write (0),//.write

/*input wire [31:0] */.reconfig_mgmt_writedata (0),//.writedata

/*output wire [139:0] */.reconfig_to_xcvr (reconfig_to_xcvr ) ,//reconfig_to_xcvr.reconfig_to_xcvr

/*input wire [91:0] */.reconfig_from_xcvr (reconfig_from_xcvr ) 
//reconfig_from_xcvr.reconfig_from_xcvr

);

dendmodule
APPENDIX B

DATA GENERATOR CODE

This module is used to generate incremental data to be transmitted.
module tx_data_gen(  
    input clk ,
    input reset ,
    input en ,
    output [15:0] data_out
);

//================================================
reg [15:0] R_data_out ;
reg [15:0] R_data_tx ;

reg [7 :0] R_traffic_cycle ;
reg R_traffic_valid0 ;
reg R_traffic_valid1 ;
reg R_id_change ;
reg [2 :0] R_id ;

//================================================
always@(posedge clk)
begin
    if(reset)
        begin
            R_traffic_cycle <= 8'd0;
        end
    else if(en)
        begin
            if(R_traffic_cycle<=8'd166)
R_traffic_cycle <= R_traffic_cycle + 8'd1;
else
R_traffic_cycle <= 8'd0;
end
end
always@(posedge clk)
begin
if((R_traffic_cycle<=8'd106) || ((R_traffic_cycle>8'd107) && (R_traffic_cycle<=8'd164)))
    R_traffic_valid0 <= 1'b1;
else
    R_traffic_valid0 <= 1'b0;
end
always@(posedge clk)
begin
R_traffic_valid1 <= R_traffic_valid0;
end
always@(posedge clk)
begin
if(reset)
begin
    R_id <= 3'd0;
end
else if(en)
begin
if(R_traffic_valid0)
begin
if(R_id<3'd7)
    R_id    <= R_id + 3'd1;
end
else
begin
    R_id    <= 3'd0;
end
end
else
begin
    R_id    <= 3'd0;
end
end
always@(posedge clk)
begin
if(reset)
    R_id_change <= 1'b0;
else if(!R_traffic_valid0 & R_traffic_valid1)
    R_id_change <= ~R_id_change;
end
always@(posedge clk)
begin
if(reset)
begin
    R_data_out <= 16'd0;
end

else if(en)
begin
    if(R_traffic_valid0)
begin
        if(!R_id_change)
begin
            case(R_id)
            3'd1,3'd2,3'd3,3'd4:
            begin
                R_data_out <= 16'haa55;
            end
            3'd5:
            begin
                R_data_out <= 16'd1;
            end
            3'd6:
            begin
                R_data_out <= 16'd100;
            end
            3'd7:
            begin
                R_data_out <= R_data_tx;
            end
        end
end
end
endcase
end
else
begin
case(R_id)
3'd1,3'd2,3'd3,3'd4:
begin
R_data_out <= 16'haa55;
end
3'd5:
begin
R_data_out <= 16'd2;
end
3'd6:
begin
R_data_out <= 16'd50;
end
3'd7:
begin
R_data_out <= R_data_tx;
end
endcase
end
end
else

begin

R_data_out <= R_data_out + 16'h1;

end

end

always@(posedge clk)

begin

if(R_id==3'd7)

begin

R_data_tx <= R_data_tx + 16'd1;

end

else

begin

R_data_tx <= 16'd0;

end

end

assign data_out = R_data_out;

endmodule
APPENDIX C

DATA EXTRACT CODE

This module is used to receive and extract data from the data stream.
module rx_data_fetch#(
    parameter ch = 0
)(

    input    clk ,
    input    reset ,
    input    rx_valid ,
    input [15:0] rx_data ,
    output   rx_fetch_en ,
    output [15:0] rx_fetch_data
);

//==============================================================

    reg        R_rx_valid_0 ;
    reg [15:0]    R_rx_data_stream ;
    reg        R_rx_valid_1 ;
    reg [15:0]    R_rx_data_1 ;
    reg        R_rx_valid_2 ;
    reg [15:0]    R_rx_data_2 ;
    reg        R_rx_valid_3 ;
    reg [15:0]    R_rx_data_3 ;
    reg        R_rx_valid_4 ;
    reg [15:0]    R_rx_data_4 ;
    reg        R_rx_valid_5 ;
    reg [15:0]    R_rx_data_5 ;
    reg [15:0]    R_fetch_count ;
    reg [15:0]    R_fetch_len ;
reg  R_fetch_valid ;
reg  R_rx_fetch_en ;
reg  [15:0] R_rx_fetch_data ;

//==============================================
always@(posedge clk)
begin
    R_rx_valid_0 <= rx_valid ;
    R_rx_data_stream <= rx_data ;
    R_rx_valid_1 <= R_rx_valid_0;
    R_rx_data_1 <= R_rx_data_stream ;
    R_rx_valid_2 <= R_rx_valid_1;
    R_rx_data_2 <= R_rx_data_1 ;
    R_rx_valid_3 <= R_rx_valid_2;
    R_rx_data_3 <= R_rx_data_2 ;
    R_rx_valid_4 <= R_rx_valid_3;
    R_rx_data_4 <= R_rx_data_3 ;
    R_rx_valid_5 <= R_rx_valid_4;
    R_rx_data_5 <= R_rx_data_4 ;
end

always@(posedge clk)
begin
    if(reset)
    begin
        R_fetch_len <= 16'd0;
    end
end
R_fetch_valid <= 1'b0;

//
R_fetch_count <= 16'd0;
R_rx_fetch_en  <= 1'b0;
R_rx_fetch_data <= 16'd0;

end

else if(R_rx_valid_5 | R_rx_valid_4 | R_rx_valid_3 | R_rx_valid_2 | R_rx_valid_1)
begin

if((R_rx_data_5==16'haa55) && (R_rx_data_4==16'haa55) && (R_rx_data_3==16'haa55)
&& (R_rx_data_2==16'haa55))
begin

if(ch==0)
begin

if(R_rx_data_1==16'h0001)
begin

R_fetch_len     <= R_rx_data_stream;
R_fetch_valid   <= 1'b1;
end
else
begin
R_fetch_valid   <= 1'b0;
end
end
else if(ch==1)
begin

if(R_rx_data_1==16'h0002)

end

end

72
begin
    R_fetch_len <= R_rx_data_stream;
    R_fetch_valid <= 1'b1;
end
else
    begin
        R_fetch_valid <= 1'b0;
    end
end

end

//=================================================
if(R_fetch_valid)
begin
    if(R_fetch_count<=R_fetch_len)
    begin
        R_fetch_count <= R_fetch_count + 16'd1;
        //
        R_rx_fetch_en <= 1'b1;
        R_rx_fetch_data <= R_rx_data_stream;
    end
else
    begin
        R_fetch_count <= 16'd0;
        R_fetch_valid <= 1'b0;
    end
end

//===============================================
if(R_fetch_valid)
begin
    if(R_fetch_count<=R_fetch_len)
    begin
        R_fetch_count <= R_fetch_count + 16'd1;
        //
        R_rx_fetch_en <= 1'b1;
        R_rx_fetch_data <= R_rx_data_stream;
    end
else
    begin
        R_fetch_count <= 16'd0;
        R_fetch_valid <= 1'b0;
    end
end

//===============================================


R_rx_fetch_en <= 1'b0;

end

end

else

begin

R_rx_fetch_en <= 1'b0;

end

end

assign rx_fetch_en = R_rx_fetch_en;
assign rx_fetch_data = R_rx_fetch_data;

endmodule
REFERENCES


