ABSTRACT

Research on Board to Board Communication for a Reconfigurable Computing System

Yue Wu, M.S.E.C.E.

Advisor: Russell W. Duren, Ph.D.

Board-to-board communications are very important for interconnecting multiple FPGA boards in a reconfigurable computing cluster. Researchers at Baylor University have developed a reconfigurable computing cluster that uses the Impulse C language to provide a platform for software designers to design hardware-accelerated systems. This thesis describes the development of two Impulse C implementations for the interconnection of Xilinx FPGA boards; one using parallel and one using serial communication hardware. Impulse C is used to design a software-numerical-communication function integrated into the hardware communication system. The hardware communication protocol is designed and implemented using VHDL and Xilinx's Embedded Development Kit (EDK). The performance of the two communication systems are tested and compared by simulation and real time hardware test applications. The advantages and disadvantages between the two different communication systems are explored as part of this research.

Research on Board to Board Communication for a Reconfigurable Computing System

by

Yue Wu, B.S.

A Thesis

Approved by the Department of Electrical and Computer Engineering

_____
Kwang Y. Lee, Ph.D., Chairperson

Submitted to the Graduate Faculty of
Baylor University in Partial Fulfillment of the
Requirements for the Degree
of
Master of Science in Electrical and Computer Engineering

Approved by the Thesis Committee

_____
Russell W. Duren, Ph.D., Chairperson

_____
Michael W. Thompson, Ph.D.

_____
David B. Sturgill, Ph.D.

Accepted by the Graduate School
August 2009

_____
J. Larry Lyon, Ph.D., Dean

Page bearing signatures is kept on file in the Graduate School.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ANSI – American National Standards Institute

API – Application Programming Interface

APU – Auxiliary Processor Unit

ASIC – Application Specific Integrated Circuits

CML – Current Mode Logic

CPU – Central Processor Unit

CRC – Cyclic Redundancy Check

CSP – Communication Sequential Processes

DCM – Digital Clock Manger

DDR SDRAM – Double Data Rate Synchronous Dynamic Random Access Memory

DIMM – Dual In-line Memory Mode

DSM – Distribute Shared Memory

EDK – Embedded Development Kit

FIFO – First-In First-Out

FPGA – Field Programmable Gate Array

FSL – Fast Simplex Link

GPU – Graphics Processor Unit

HDL – Hardware Description Language

IDE – Integrated Development Environment

IP – Intellectual Property

ISE – Integrated Software Environment

JTAG – Joint Test Action Group

MGT – Multi-Gigabits Transceiver

NCD – Native Circuit Description

NGD – Native Generic Database

OPB – On-chip Peripheral Bus

PATA – Parallel Advanced Technology Attachment

PCB – Printed Circuit Board

PCS – Physical Coding Sublayer

PPC – PowerPC

PLB – Processor Local Bus

PMA – Physical Media Attachment

RAM – Random Access Memory

RCM – Reconfigurable Cluster Middleware

RTL – Register Transfer Logic

SATA – Serial Advanced Technology Attachment

SDK – Software Development Stud io

UART – Universal Asynchronous Receiver/Transmitter

UCF – User Constraint File

USB – Universal Serial Bus

VHDL – VHSIC Hardware Description Language

VHSIC – Very High Speed Integrated Circuits

XPS – Xilinx Platform Studio

XUP – Xilinx University Program

ACKNOWLEDGMENTS

I would like to thank my parents, for their infinite love, patience, and support. Thanks to Dr. Duren for providing me with this topic to work on, and teaching me the joy of research, keeping the project interesting. I would like to thank Dr. Thompson and Dr. Sturgill for their support and guidance on my thesis work.

CHAPTER ONE

Introduction

In last twenty years, Field Programmable Gate Arrays (FPGA) has become a new area of hardware application and research. Unlike Application Specific Integrated Circuit (ASIC) designs, FPGAs are very flexible in that the hardware can be altered by reprogramming. Therefore, FPGAs are offer relatively low cost with high flexibility and good performance. In fact, FPGA based computations are beginning to be widely used in many applications and research. The development of FPGA devices has progressed rapidly and current chip speeds have reached the multi-gigabit Hz level. The increases in processing speed have enhanced the desirability of FPGAs based designs. However, the speed of system interconnect has become a critical issue. In many cases the system interconnection bandwidth limitation has been the primary bottleneck of the system. It has therefore become increasingly important for system designers to pay much more attention to the development of system interconnection technologies.

In many applications individual FPGA board are designed to implement a given function. Employing several FPGA implement various functions is a common approach toward arriving at flexible, low-cost systems that are capable of very high performance. A critical issue for this type of system is the data communication between boards. There exist several communication protocols for the board-to-board data communication. The primary focus of this research project is to explore two different inter-board communication methods; a parallel communication technique and a high-speed serial communication method.

Parallel I/O technology is widely used in many interconnected systems. A parallel communication approach can use several parallel channels or buses to send several data bits simultaneously. Unlike the early serial communication techniques, the parallel technology transmits multiple bits during a transmission period. Parallel techniques attempt to enhance the speed of data link by the use of simultaneous bit transmissions. As an additional advantage, the creation of a parallel port for a typical computer system requires only a latch to copy data directly onto the data bus. Therefore, in many applications parallel I/O interfaces are easily implemented in hardware. There are, however, distinct disadvantages toward using a parallel communication approach that can limited the usefulness of a parallel approach [1]. Because the parallel communication sends data bits simultaneously, it requires many physical ports on the board. For example, if we want to transmit 16-bit width data through the parallel ports, then there must be 16 parallel I/O interfaces on the board. There is clearly a practical limit to the degree in which parallel data can be received. For example, it is likely to be impractical to use 128 parallel I/O interfaces on one board to transmit a 128-bit width data. This degree of extension of parallel I/O ports will not only require more hardware space on the board but also increase the difficulty of design because the number of PCB layers is increased. Additionally, as the bit-width is expanded crosstalk interference becomes an increasingly difficult problem. For example, the parallel cable length would be quite limited since long since crosstalk interference increases with the length of the communication link.

The serial communication method requires fewer I/O ports because for this approach a single bit is transmitted during the transmission interval. A major advantage is that less hardware space is used on the board and so the number of PCB layers is

dramatically reduced [1]. Additionally, the crosstalk interference is not a major issue for the serial connections. However, the major consideration for serial communication is the transmission speed. While a parallel link could transmit several streams of data along the multiple parallel interfaces at one time, a serial link can only transmit a single stream of data. Advances in serial communication techniques have resulted dramatically higher data rates. The invention of high speed serial I/O interconnection leads us to the multi-gigabit serial communication technology. The Serial Advanced Technology Attachment (SATA) is an interface standard for the high speed serial interconnection. However, it is up to the user to develop a data communication protocol for the high speed serial communication since the connection between data and the SATA ports is not directly specified.

The focus of this research is to investigate board-to-board communication by using both the parallel interconnection method and the high speed serial interconnection technique. The plan is to develop two different communication systems for a specific application and then to analyze the difference. In the process, we will become familiar with both the parallel ports and the serial ports data stream connections. The plan also will involve researching and developing a possible protocol for the system. The protocol will be implemented using VHDL hardware description language and the control of data stream will be implemented by using the Impulse C CoDeveloper software. A secondary emphasis of my research is to design and implement a direct Impulse C data stream to hardware communication protocol interface. The Impulse C provides a development environment for software coders. Designers could use C-like code to write programs. The VHDL code for these programs will be automatically generated by Impulse C

CoBuilder. It is noteworthy that Impulse C is very good at handling the streams and processes.

The data stream communication system used two XUP Virtex II Pro hardware development platforms as the target devices. Chapter two provides the background information of the design tools and the design languages that I used. Chapter three is mainly focused on the implementation and performance of the First-In First-Out (FIFO) which is very important for data communication. Chapter four describes the use of Impulse C and the implementation of communication interfaces with parallel hardware. Chapter five describes the research for the high speed serial communication hardware support. It also describes the protocol and explains the design and the implementation of the entire high speed serial communication system with Aurora hardware. The performance and communication results of both parallel and high speed serial interface will be discussed in chapter six. Chapter seven summarizes the entire thesis project and offers conclusions about future research areas for improving data stream communications.

CHAPTER TWO

Background Information

Reconfigurable Computing and the FPGA

Prior to the popularity of programmable logic, hardware designers often implemented systems in hardware using Application Specific Integrated Circuits (ASIC). The ASIC approach has several advantages with speed being a primary advantage. However, the problem is that the ASIC approach is not flexible. A typical ASIC design can not be altered after fabrication and therefore the cost of building an ASIC is often expensive.

Another typical approach is to use general purpose software-programmed processors. Designers could then implement systems by writing software. A primary advantage to this approach is that the software is very flexible and easy to change. Changes in system requirements could be easily accommodated by rewriting software. However, in many cases the speed of a general purpose processor is not adequate for the real-time applications [2].

Reconfigurable computing offers the possibility of hardware acceleration and yet maintains flexibility in that the system can be reprogrammed by the hardware designer. Reconfigurable computing is a computing paradigm that combines some of the flexibility of traditional software programming with the high performance offered by hardware acceleration. With those features, reconfigurable computing systems have not only the advantages of pure hardware systems but also the advantages of software program systems.

The concept of reconfigurable computing has been around since the 1960s, when a paper published by Gerald Estrin, proposed the concept of a computer made of a standard processor and an array of "reconfigurable" hardware. In such a system, the main processor acted as a controller for the reconfigurable hardware, which was dynamically changed (reconfigured) to perform specific tasks as dedicated hardware. Although his idea was very exciting, it also was far ahead of the time and was not practical because of the lack of needed technology [3]. It was not until 1980s, the advent of Field Programmable Gate Array (FPGA), the reconfigurable computing came to the new era. With the advancement of device technology, there has been a great deal of new research into creating dynamically reconfigurable computers using these relatively low-cost, easy-to-program devices. When it comes to reconfigurable computing, however, FPGA definitely stimulates the high-performance computing and complex applications which require a high degree of flexibility.

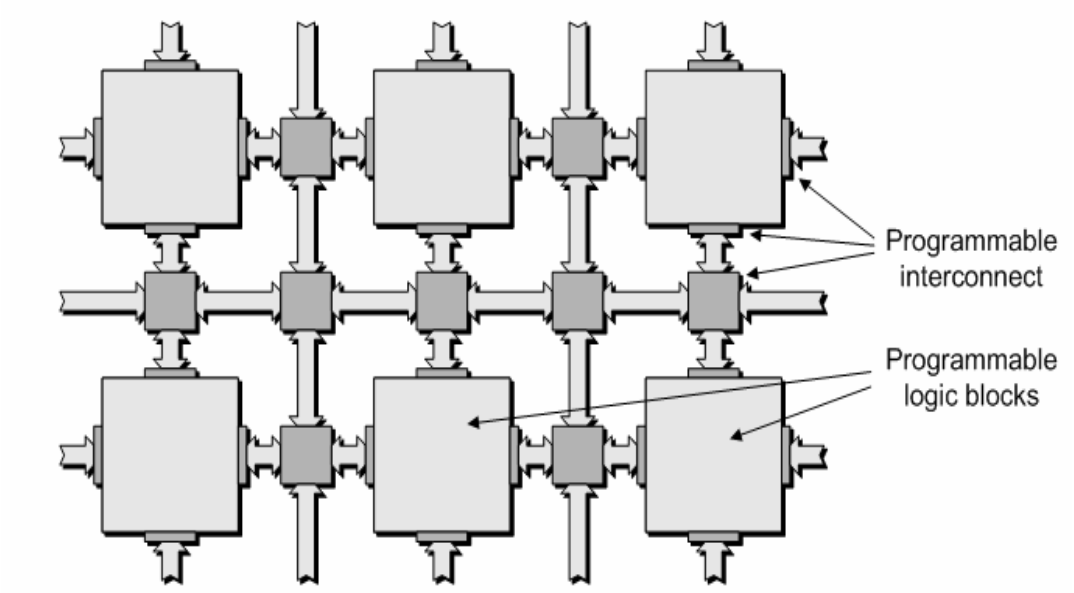


Figure 1: Reconfigurable Devices of FPGA [2]

A FPGA is a semiconductor device that could be configured by designer after manufacturing. It consists of an array of programmable logic blocks whose functionality is determined by programmable configuration bits and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together"—somewhat like a one-chip programmable breadboard. Those logic blocks are connected by a set of routing resources that are also programmable. Figure 1 shows the reconfigurable devices of FPGA.

In most FPGAs, the logic blocks can be configured to perform complex combinational functions or merely simple logic gates, they also include memory elements, which may be simple flip-flops or more complete blocks of memory. Custom logic circuits can be mapped to the reconfigurable fabric. FPGAs are programmed using a logic circuit diagram or a code like hardware description language (HDL) to specify how the chip will work.

A Hardware Description Language (HDL) is the language to describe the circuits' operation, design and organization. HDLs are standard text-based expressions of the spatial and temporal structure and behaviors of electronic systems. In contrast to a software programming language, HDL syntax and semantics include explicit notations for expressing time and concurrency, which are the primary attributes of hardware. There are two typical HDL languages: VHDL and Verilog. Verilog is much like a software design language; it is easy to use for the design of small functions. VHDL (VHSIC (Very High Speed Integrated Circuits) hardware description language) is commonly used as a design-entry language for FPGAs in electronic design automation of digital circuits. I used VHDL for my system design.

Research Information on the Reconfigurable Cluster

Co-computing is the trend of the high performance computer technology; a common approach is to use coprocessors to accelerate the speed of computationally intensive applications. The best-known uses of coprocessors are center processor units (CPUs) and graphics processor units (GPUs). Accelerated computing represents the leading edge of the high performance computing wave, using programmable logic and other nontraditional processing resources to augment clusters has become increasing popular. In high performance reconfigurable computing, FPGAs act as coprocessors. FPGA devices show great promise for cost-effective accelerated computing. Because of the tremendous potential of FPGA's, there are several groups and institutions that are researching FPGA-based reconfigurable computing clusters.

Baylor's Department of Electrical and Computer Engineering research group has been involved with reconfigurable computing clusters for several years. Many students have done research in this area. Paul Reynolds, Jeremy Stevenson, Bernard Lamb, Carmen Li Shen, Stephen Dark and Jonathan Franz investigated programming languages for software-like descriptions of hardware including System C, Handel-C, Impulse C, Carte-C and others, and the Impulse C was selected for ease of use. Carmen and Stephen investigated custom software methods for inter-board communication [13]. Willis Troy and Spenser Gilliland have developed applications using Linux, MPI and Impulse C. They use multiple XUP boards to build up a cluster computer system. In this system, the XUP boards perform like nodes, and these nodes are all connected by the Ethernet LAN to a 24 port Ethernet switch. A Mini-ITX form factor PC used as the head node of the cluster. It likes a bridge which connects the control terminator and the switch. The

switch could also connect a broadband router to the internet service [4]. Figure 2 shows the block diagram of the Reconfigurable Cluster (RC) system from Baylor group.

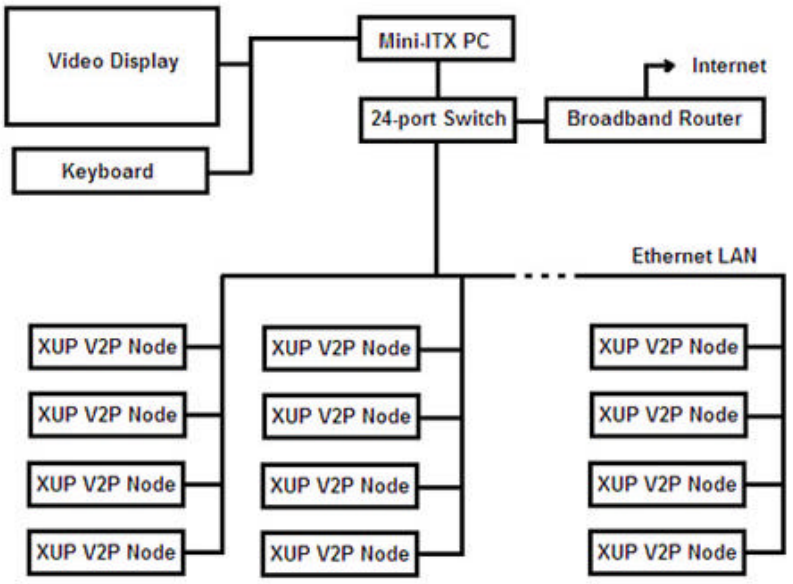


Figure 2: Block Diagram of RC System from Baylor Group

The main object of reconfigurable computing cluster is to enhance computing performance. Parallel computing is one way to accelerate the computing speed. Tasks can be sent to boards which process several tasks simultaneously. Another important feature of an RC cluster is the memory. Coprocessors may be able to process multiple tasks on one chip but often the bottle-neck is the system memory. A typical RC cluster does not have this problem. Raik Nagel and Thomas Rauber from University of Bayreuth propose the *RCM* system (Reconfigurable Cluster Middleware) which is a DSM realization consisting of several layers with different abstraction levels. They tried different applications on different layers and mix them together to combine the advantage of pure DSM systems and pure message-passing systems [5]. Ron Sass, William V.

Kritikos, Andrew G. Schmidt, Srinivas Beeravolu, Parag Beeraka, Kushal Datta, David Andrews, Richard S. Miller, and Daniel Stanzione, Jr. from University of North Carolina at Charlotte investigated the use of platform FPGAs to build cost-effective petascale computers [6]. Other researchers, Andrew G. Schmidt, William V. Kritikos, Siddhartha Datta, and Ron Sass from University of North Carolina at Charlotte presented a new idea of transfer data in and out of the hardware core, which provide feasibility of parallel board data communication [7].

The Current State of the Art in Board-to-Board Communication

In the reconfigurable computing cluster system, an important issue for parallel computing or multiple tasks processing is the communication between boards. If boards cannot share data, the RC cluster system is not useful, making the data communication between boards an important research topic.

Currently, world wide researchers focus on the ways to enhance the efficiency of data communication. There are several ways to transfer data, including serial data transfer, parallel data transfer and the new serial ATA ways. Xilinx provide several interface for data communication, including the RS-232 serial port, low-speed expansion ports, high-speed expansion port and Serial ATA Gigabit transceiver ports. However, data communication involves several issues. Various boards may have different clock rates and even within a single board, the clock rate for various functions may differ. So It is important to consider the effects of the clock frequency in order to receive the appropriate data.

Recent research by Zbyn?ek Vymazal from Czech Technical University in Prague provides a design idea of data communication of FPGA chips using Rocket IO [1]. He

analyzed the use of a fast serial bus realized by the Rocket IO transceivers for communication between FPGAs and proved the usage of the high speed serial data communication. W. V. Kritikos from University of Kansas researched the feasibility of a computing cluster network based on SATA cables which provide adequately error-free transmission for length up to meters [8]. Shouqian Yu, Lili Yi, Weihai Chen, Zhaojin Wen from Beijing University of Aeronautics & Astronautics presented a design method of asynchronous FIFO and structure of the controller which is designed with FIFO circuit block and UART circuit block within FPGA to implement the effective communication in complex control systems [9]. Their idea of FIFO design provides a good way for the implementation of inter-board communication.

However, if we want the RC cluster to be more practical, we must add the user function to the FPGAs. This results in the new problem of how to specify the user interface and connect the user data stream between boards? Impulse C provides a feasible solution for the user data stream interface. Unlike VHDL and Verilog, Impulse C is a C-like language and it provides several data stream processing functions. It can also automatically generate the required hardware/software communication channels using FSL, APU and other Xilinx interfaces. Furthermore, Impulse C can compile directly into optimized logic ready for use with Xilinx FPGAs. Thus, we can integrate the Impulse C with the data transfer protocol to implement the RC cluster design.

In my project, I use the Impulse C to generate the data stream and add hardware processing functions on the data stream. I also researched the two data communication protocols: parallel communication and SATA communication. Furthermore, I

implemented the designs with the Impulse C data stream. A comparison of the two methods was also made.

Design Tools and the System

In the project, I used several design tools such as: Xilinx ISE, Xilinx Core Generator, Impulse C and Xilinx EDK. The development system used was the Xilinx Virtex-II pro hardware platform. I will briefly introduce them in the following paragraphs.

Xilinx Integrated Software Environment (ISE) is the most popular software development tools for logic design. It is the foundation for Xilinx FPGA logic design. Because FPGA design can also be a complicated process, Xilinx has provided software development tools that allow customer to simplify some of this complexity. There are many various utilities, such as constraints entry, timing analysis, logic placement and routing, and device programming have all been integrated into ISE.

Xilinx Core Generator delivers a library of both parameterizable and point solution LogiCORE IP cores with detailed data sheet specifications. Customers could use the core generator to generate the IP core they want, and they could also modify the parameters of the IP core to meet their design requirements. The core generator can be accessed from within the ISE design environment or as a stand-alone tool. Easy access to optimized cores helps customers get the results fast and easy.

Impulse C is a revolutionary new way of creating high-performance applications intended for current and future generations of FPGA-based programmable hardware platforms. It is a function library and related compiler and debugging tools compatible with standard ANSI C that enable the development of highly parallel algorithms and

applications for current and future generations of programmable hardware platforms. Impulse C has been developed to address the needs of embedded systems designers and software programmers who wish to take advantage of programmable and reconfigurable hardware for application acceleration.

Xilinx Embedded Development Kit (EDK) is a suite of tools and Intellectual Property (IP) that enables customer to design a complete embedded processor system for implementation in a Xilinx Field Programmable Gate Array (FPGA) device [10]. EDK also provide a software design environment, customers could use c-like code to implement their design, like arrange the data input and output, design special functions like counter, etc. It likes an umbrella which covering all things related to embedded processor systems and their design.

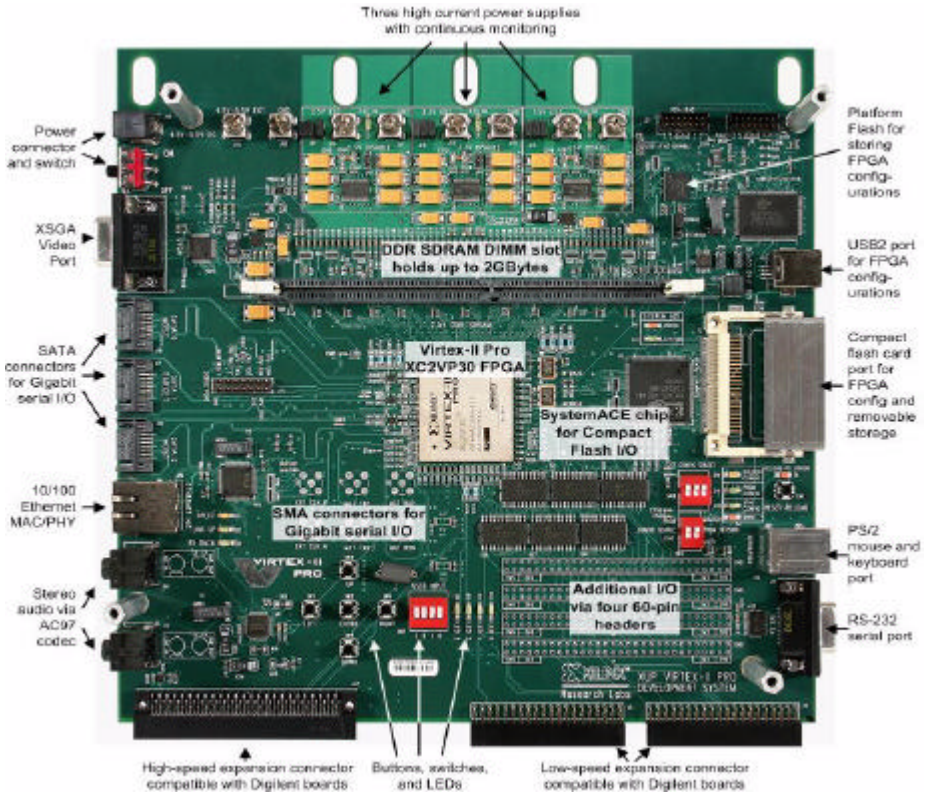


Figure 3: XUP Virtex-II Pro Development System Board [11]

All of the software mentioned above is running on the Xilinx Virtex-II pro hardware platform. Figure 3 shows the photo of XUP Virtex-II Pro development system board. It consists of a high performance Virtex-II Pro Platform FPGA surrounded by a comprehensive collection of peripheral components that can be used to create a complex system and to demonstrate the capability of the Virtex-II Pro Platform FPGA. It has roughly three million logic gates, 136 eighteen-bit multipliers and 136 block RAMs, two hard core processors (IBM PowerPCs), and a DDR SDRAM DIMM Module which holds up to 2GBytes. The board is also populated with multi-gigabit transceivers, three of which are employed as SATA interface. It also contains multiple ports: JTAG, USB, RS232, PS2 for keyboard, and mouse, Ethernet, video and has capability for serial communication and audio interface [11].

CHAPTER THREE

Implementation and the Performance of FIFO

The Idea of Using FIFO

The most important thing for the data communication is to correctly receive the data. But when the boards are processing multiple tasks, the different clock frequency requirement of the tasks may cause the data communication problem. That is, when one task sends data in a clock frequency and another task receive the data by another different clock frequency, and then it may miss some data and receive incorrectly interpret other data. Figure 4 shows one of those situations.

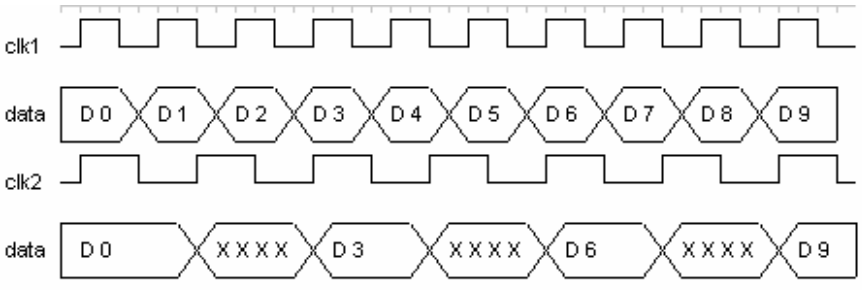


Figure 4: Transfer Data with Different Clock Frequency

In figure 4, we could see that task1 sends data on the rising edge of clk1, and task2 receives data on the rising edge of clk2. Because clk1 and clk2 has different clock frequency, so when these two clocks are all on the rising edge, the data send by task1 will be received by task2, or task2 will receive incorrect data.

Therefore, we must use some kind of data buffer prevent the data error transfer under the different clock frequency situations. That's the reason why we use FIFO for the design. FIFO means first in first out; it's a design of data structure which the first data in the memory will be taken out first. In digital designs, FIFOs are ubiquitous constructs required for data manipulation tasks such as clock domain crossing, low-latency memory buffering, and bus width conversion. FIFO could collect data together for reading and writing, so it also prevent the frequently system bus operation, save the resource of the system, enhance the efficiency of the system [12].

There are two types of FIFOs, independent or common clock domains for write and read operations. The independent clock FIFOs enables the user to implement unique clock domains on the write and read ports. The FIFOs handles the synchronization between clock domains, placing no requirements on phase and frequency relationships between clocks. A common clock domain implementation optimizes the core for data buffering within a single clock domain [12]. In my project design, in order to synchronize the different clocks, I chose the independent clock FIFOs. Figure 5 shows the FIFO with independent clock.

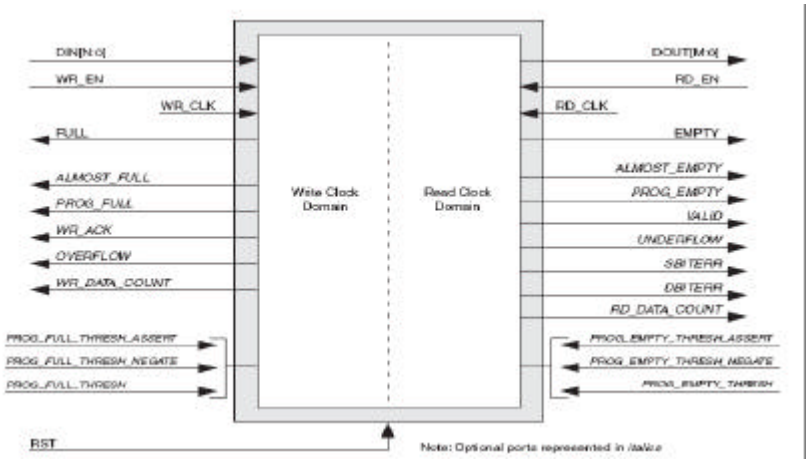


Figure 5: FIFO with Independent Clocks and Its Interface Signals [12]

In such a FIFO, there are many interface signals; some of which are not used in my design. I will briefly introduce the signals I used. First of all, the signal RST is a reset of the entire core logic (both write and read clock domain). This signal is an asynchronous reset that initializes all internal pointers and output registers. When it is enabled, it will be High for at least three read clock and write clock cycles to ensure all internal states are reset to the correct values. Signal vector DIN [N: 0] and DOUT [N: 0] are the input and output data buses used when writing and reading the FIFO. 'N' represents the width of the data. WR_CLK and RD_CLK are the clock signal interface, and they could be connected to same clock or different clocks. All signals on the write domain are synchronous to the WR_CLK and signals on the read domain will be synchronous to the RD_CLK. WR_EN and RD_EN are the signal which could be controlled by the outside logic, when the WR_EN or RD_EN is asserted, it causes the data (on DIN or DOUT) to be written or read from the FIFO. FULL and EMPTY signal are the status flag of the FIFO. When the FULL signal is asserted, it indicates that the FIFO is full. Write requests are ignored when the FIFO is full; initiating a write when the FIFO is full is non-destructive to the contents of the FIFO. When EMPTY is asserted, this signal indicates that the FIFO is empty. Read requests are ignored when the FIFO is empty; initiating a read while empty is non-destructive to the FIFO [12].

Design and Implementation

FIFO is a very important device for the data stream communication. Xilinx FPGAs can support various types of FIFOs. By using the Xilinx Core Generator, the FIFOs can be generated automatically. However, I must specify the parameters to make

the generated FIFOs meet our requirements. Table 1 shows the FIFO parameters that I specified in the Core Generator.

Table 1: FIFO Parameters Specified in Xilinx Core Generator

FIFO Parameters	Options
Read/Write Clock Domains	Independent Clocks
Memory Type	Block RAM
Read mode	Standard FIFO
Almost Full/Empty Flags	Not Selected
Programmable Full/Empty Flags	Not Selected
Handshaking	Not Selected
Data Count Outputs	Not Selected
Write/Read Data Width	16
Write/Read Data Depth	1024
Reset Pin	Asynchronous Reset

In order to learn the features of the FIFO generated by the Xilinx Core Generator, I designed a simple project to test the performance of the FIFOs and the user application logic.

To understand the nature of FIFO designs, it is important to understand how pipelining is used to maximize performance and implement synchronization logic for clock-domain crossing. Data written into the write interface may take multiple clock cycles before it can be accessed on the read interface.

FIFOs with independent write and read clocks require that interface signals be used only in their respective clock domains. The independent clocks FIFO handles all synchronization requirements, enabling the user to cross between two clock domains that have no relationship in frequency or phase. Figure 6 illustrates the functional implementation of a FIFO configured with independent clocks.

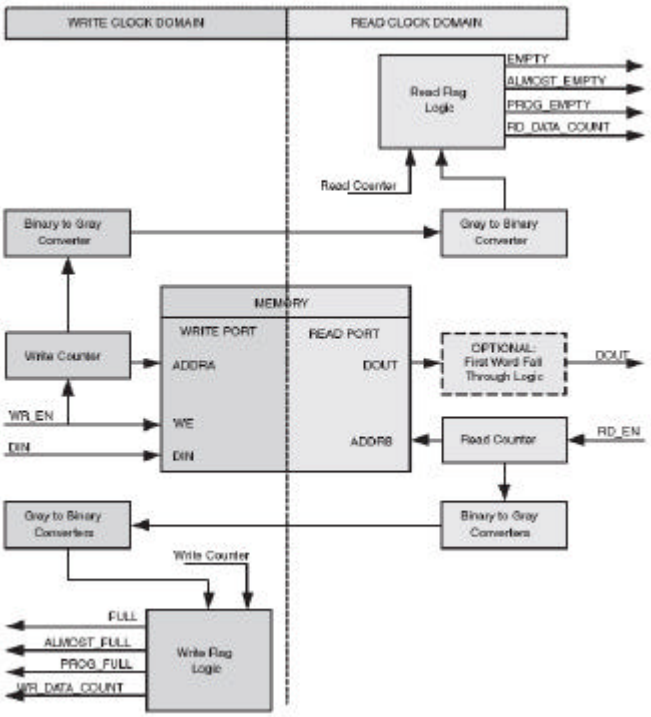


Figure 6: Function Implementation of a FIFO with Independent Clocks [12]

With the full understanding of the features of FIFOs, I designed a simple two board communication project with two FIFOs, three state machines and one checking circuit. The design diagrams of those state machines are attached in appendices. The main board contains most of the application logic: 2 FIFOs, 2 State Machines and the Checking circuit. Another board just has one state machine, but it not only controls the input and output data but also handles all the signals which come from FIFO1 and go to FIFO2. The data is generated by a counter which is controlled by the state machine one. The counter will continuously generate data to the FIFO1 and the state machine one will handle the signals from the write domain of FIFO1. State machine three will control the signals from the read domain of FIFO2. The checking circuit will receive the data from the output of FIFO2 and compare them with the counter data. The write clock of FIFO1

and the read clock of FIFO2 will be connected to the system clock from board1, and the read clock of FIFO1 and the write clock of FIFO2 is connected to the system clock from board2. Figure 7 shows the brief design of the communication project with FIFOs.

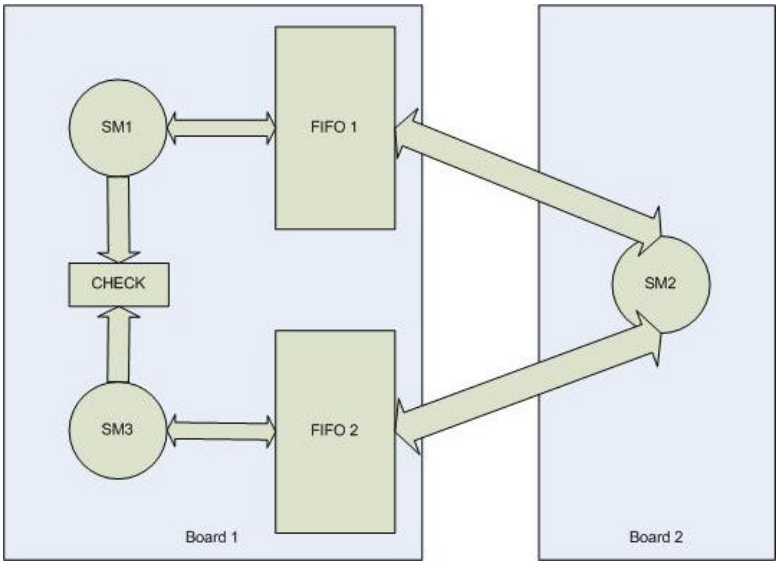


Figure 7: Brief Design of the Communication Project with FIFOs

Board 1 contains both FIFOs, two state machines that control writing to FIFO 1 and reading from FIFO 2, and circuitry that uses a simple counter to generate data to transmit and verify reception of the proper data. The counter and state machines obtain their clock signals from a 100 MHz oscillator on Board 1. Board 2 contains a state machine that reads data from FIFO 1 and writes it back to FIFO 2. This state machine uses an independent 100 MHz oscillator on Board 2. The state diagrams for the three state machines are provided in Appendix A.

One thing need to be considered carefully is the reset status. In my design, I used the asynchronous reset. The Xilinx FIFO cores synchronize the reset to the clock domain in which it is used, to ensure that the FIFO initialized to a known state. This

synchronization logic allows for proper timing of the reset logic within the core to avoid glitches and meta-stable behavior. Another important thing is the clocks of the FIFOs, the FIFO generated by the Xilinx Core Generator is designed to work only with free-running write and read clocks. It means that the clocks of the FIFOs are not recommended to be manipulated by the user logic.

The design requires that the data is to be sent outside of the board and received by another board. The data interface must be constrained to the physical ports of the board. Here, I chose the low speed expansion I/O ports. These ports use the same connector as is used for the parallel IDE interface. There are 40 pins in this connector; arraigned in two rows of 20 pins. In the upper row, pin1 has been permanently connected to the ground and pin3 is connected to the 3.3 volt power supply, pin 5 to pin 39 are open to use. In the second row, pin2 is connected to the 5 volt power supply, pin4 to pin40 are open for use. Therefore, the IDE connector provides enough pins to specify the 16-bit input and output data and other control signals. Figure 8 shows the IDE connector.

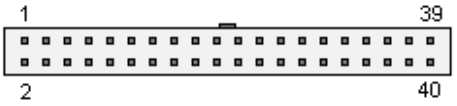


Figure 8: 40 Pin IDE Cable Connector [13]

The IDE interface requires only one cable. For this project all of the pins were to be connected straight through; from pin 1 of the first board to pin 1 of the next, from pin 4 of the first board to pin 4 of the next, and so on. Pins 2 and 3 were left open as these were hard-wired to power supply voltages on the XUP board.

Two problems were encountered when connecting these ports. Initially, I had planned to use an IDE cable to connect the two boards. However, the IDE cables did not

connect pins straight through. All of the pins that are used for ground connections in the parallel IDE standard (pins 19, 22, 24, 26, 30 and 40) were shorted together. Also, pin 28 was open in these cables. As a result, the IDE cable was not used for this project. Instead, individual wires were used to connect the two boards. The IDE cables were eventually used for the project in the next chapter. The second problem was caused by errors in the Xilinx documentation. The User Guide and Schematics differed on which FPGA pins were connected to pin 40 of the IDE connector. In some cases, the documentation said that this pin was connected to location U7 on the FPGA. In other places the documentation said it was location T7. Testing revealed that the proper location was T7. The final pin connections are provided in the User Constraint Files.

With the complete idea of the project, the next step was to write the VHDL source code. With the support of Xilinx ISE, I integrated two FIFOs into my project, designed my control logic and checking circuits and specified the interface between the logic and the FIFOs with the VHDL code. The design and coding was then straight forward and no problems were encountered. The VHDL source code files for this project are included in Appendix A. Before loading design into the hardware, several additional steps had to be completed to turn the VHDL source code into a bitstream file for the target FPGA. These steps include:

1. The User Constraint File (UCF) must be set up correctly. UCF file maps the design pins onto the physical layer of the FPGA boards, if it's not set up correctly, we cannot get correct results. The setup of UCF file could be referred to the XUPV2P_User_Guide [11]. As previously noted, one error was discovered in the Xilinx documentation. The UCF files for this project are included in Appendix A.

2. The VHDL source code is synthesized. During synthesis, behavioral information in the HDL file is translated into a structural netlist, and the design is optimized for a Xilinx target device.
3. Then the Native Generic Database (NGD) which describes the logic design will be created. Afterward the MAP program maps a logical design to a Xilinx FPGA. The input to MAP is an NGD file. The output design is an NCD (Native Circuit Description) file – a physical representation of the design mapped to the components in the FPGA.
4. After mapped NCD file is created the Place and Route program is applied. If the design is completely routed, fully routed NCD file is processed and the configuration bitstream is produced.
5. The configuration bitstream file can be converted into a PROM format file. The PROM file contains configuration data for the target FPGA device. Alternatively, the bitstream file can be loaded directly into the FPGA using a JTAG serial interface.

After those steps, the bitstream files could be downloaded to the boards. Because I designed a data communication project between two boards, there were two bitstream files, one for each board. In the follows paragraph, I will talk about the performance and the results of the FIFO data communication system.

Performance and Simulation Results

After the project is compiled by the ISE, I could download the bitstream files into the target boards. Before that step, I could also use the ISE simulator which is integrated in the ISE software to test the performance of the FIFOs. Because the simulator could

only do simulation on one board and the main functions are all located in the Board1, in order to get the ideal simulation results, the simulation is based on board1. Therefore, in the simulatio n, I simulated some signals from board2 including write and read enable signals, full and empty signals and the clock. Figure 9 shows the simulation results of the project.

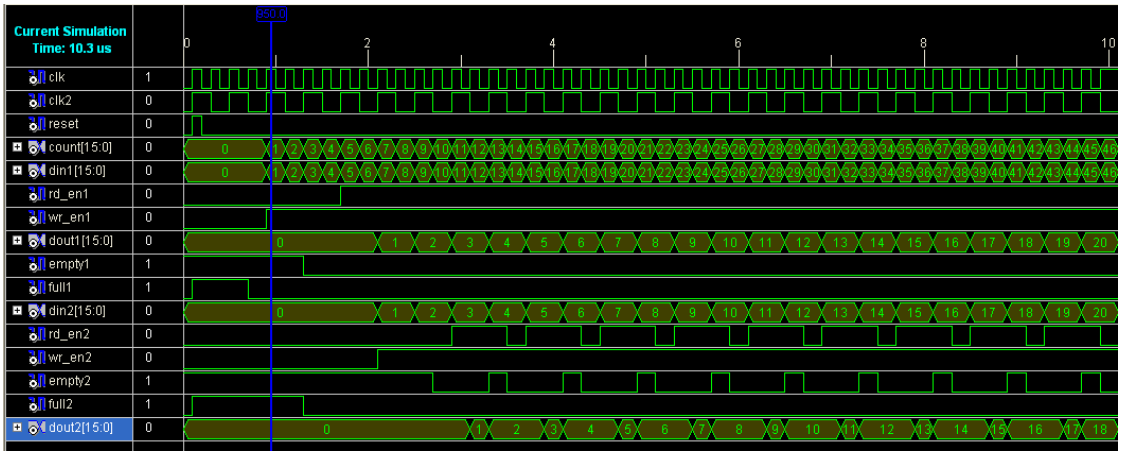


Figure9: The Simulation Results

From the figure 9, there are two different clocks. The ‘count’ signal provides the data stream, the signals ‘din1’ and ‘dout1’ are the input and output signals of FIFO1, and the signals ‘din2’ and ‘dout2’ are the input and output signals of FIFO2. Other signals are all handshaking signals from two FIFOs controlled by three state machines. It can be seen that after the system reset, the status of the two FIFOs lasts three read and write clock cycles to ensure that all internal states are reset to the correct values. The signal vector din1 receives data generated by the counter. The output dout1 is connected to the din2 which is the input of FIFO2. From the figure, it can be seen that after three clocks (clk2) of FIFO1, the dout1 will write data (started from 0) to the din2 with no errors, and then after another three clocks (clk1) of FIFO2, the output dout2 start output data. From

the figure, the data latency between the input of FIFO1 and the output of FIFO2 is about 9 clock cycles based on the board1's clock. It can be seen that the output data from dout2 matches the input data very well, that means the project will work well on the communication between two projects. The simulation results were satisfactory, so I download the two bitstream files into the target boards. The real input and output signals could be traced by the digital oscilloscope from the external parallel I/O ports. I used two LEDs to show the checking status of the circuits. If the received data is correct, then the pass LED (led3) will flash. If an error occurs, then the fail LED (led0) will flash. Although the simulation results are good, the real design had problems. In the real design, the pass led flashes during communication, but the fail LED sometimes also flashes.

One way to monitor the parallel signals directly is using a digital mixed-signal oscilloscope to trace each pin of the parallel I/O ports. The 16-bit data stream is generated as a counter and those 16-bit output signals are recorded by the oscilloscope. Figure 10-11 shows the output data from board1.

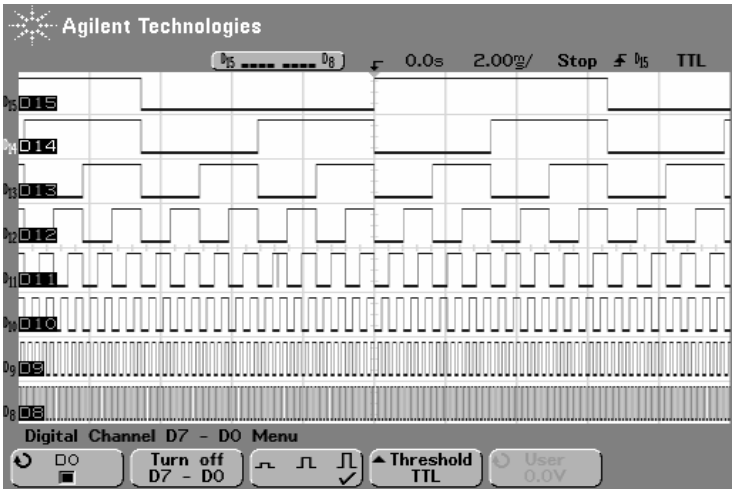


Figure 10: Output Data from Board1 (D8~D15)

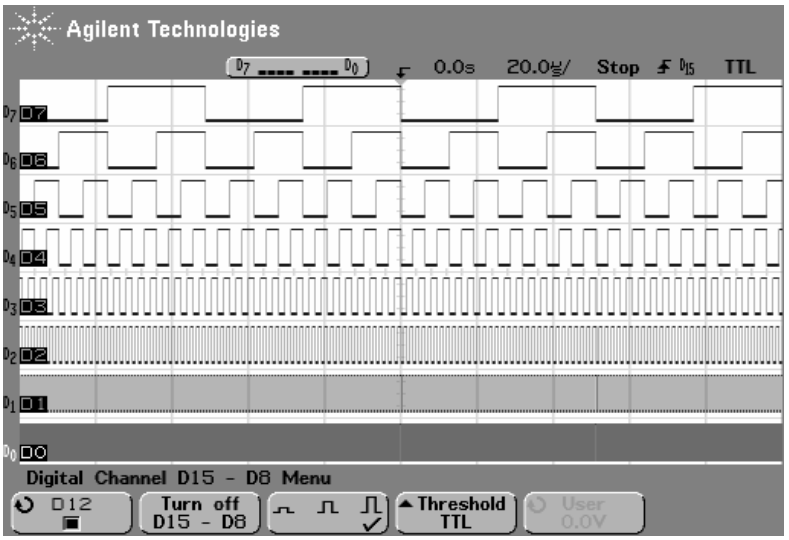


Figure 11: Output Data from Board1 (D0~D7)

From the waveform, it can be seen that the output data from board1 has no problems, each signal from D0 to D15 is very clear which means there are no errors on the output data stream from board1. But the output data from board2 to board1 has some problems. From the figure 12 and figure 13 we could see that there are some glitches on the data bits 15, 14 and 7. Figure 12~13 shows the output data from board2.

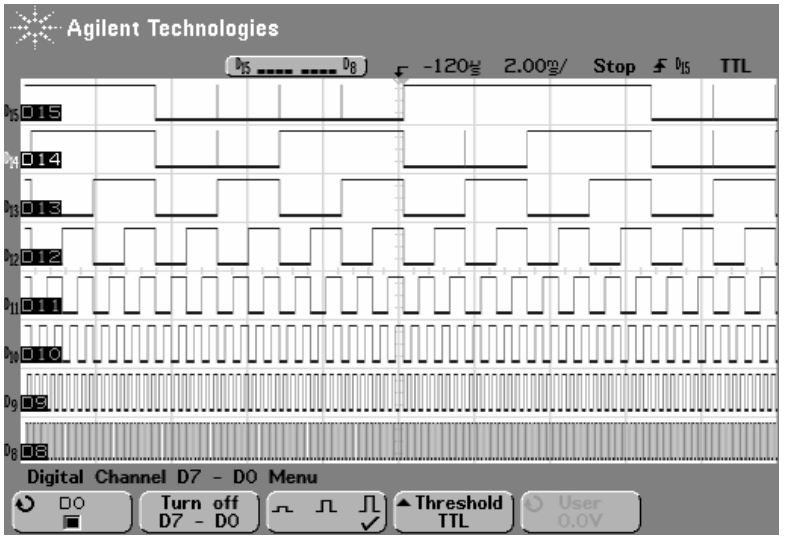


Figure 12: Output Data from Board2 (D8~D15)

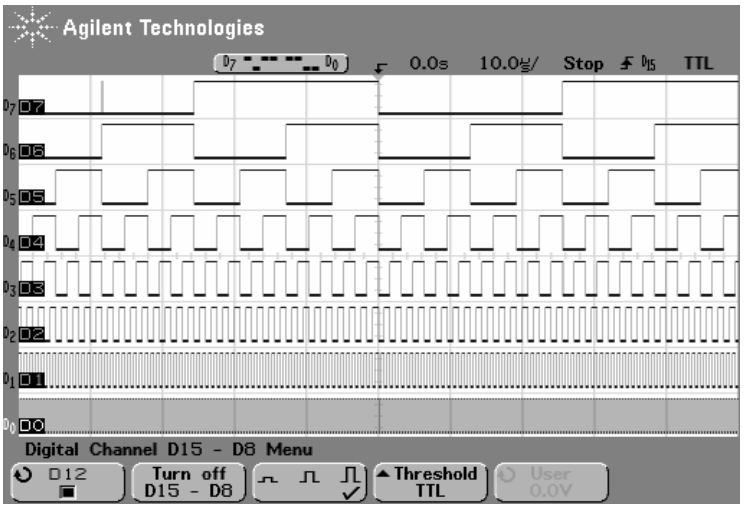


Figure 13: Output Data from Board2 (D0~D7)

In the real design, there could be several problems that cause the errors. This includes crosstalk, interference or problems caused by using discrete wires instead of a cable resulting in loose or intermittent connections. It is also possible that there some undiscovered design errors, including the state machine logic or timing. Since the main goal of this thesis was to develop interfaces compatible with Impulse C, it was decided to leave the design in this state and proceed with the Impulse C parallel interface discussed in the next chapter.

CHAPTER FOUR

Implementation of Hardware Interfaces with Parallel Hardware

Overview

Chapter 3 presented the design of a parallel interface that connected two boards. In that chapter, the design was created using standard digital design tools, in particular Xilinx's Project Navigator and VHDL. The circuits were described using VHDL and FIFOs that were created using the Xilinx CORE Generator tool. The resulting FIFO cores were included in the design as VHDL components. In order to make reconfigurable computing more accessible to programmers without a hardware background, this chapter will develop a similar circuit described using ANSI C instead of VHDL. Impulse C will be used to make this possible.

Introduction to Impulse C

Impulse C is a library of functions and related data types that provide a programming environment, and a programming model, for highly parallel applications targeting FPGA-based platforms. It is a C-to-HDL compiler from Impulse Accelerated Technology (IAT), which was founded in 2002 [14]. IAT supports Impulse C with various tools including the CoDeveloper Integrated Development Environment (IDE), the CoBuilder compiler, which compiles a subset of an Impulse C program into HDL files, and the CoMonitor application monitor tool that supports software-level debugging. The above Impulse C has been designed to simplify the expression, verification and compilation of complex applications consisting of multiple of distinct parallel processes.

It has been optimized for mixed software/hardware targets, with the goal of abstracting details of inter-process communication and allowing relatively platform-independent application design. A process is an independently executing section of code that is defined by a C subroutine called the process run function. A software process is designated to run on a conventional processor, while a hardware process is designated to run on an FPGA or other programmable hardware element. Impulse C has been developed to address the needs of embedded systems designers and software programmers who wish to take advantage of programmable and reconfigurable hardware for application acceleration [15, 16].

Impulse C is compatible with all standard ANSI C environments, allowing standard C tools to be used for designing and debugging applications targeting FPGAs. The Impulse C compiler takes a specified subset of a C program and generates FPGA hardware in the form of HDL files [14, 16]. It enables the development of highly parallel algorithms and applications for current and future generations of programmable and reconfigurable hardware platforms. Therefore, it is a part of the new trend called platform-based design which encourages the use of standard programmable platforms as an alternative to high-risk, custom ASIC solution for high performance applications. Impulse C has been created to support many types of programmable platforms using a common method of design representation and a common programming model [16].

Impulse C is especially designed for data stream processing applications. It supports a variant of communication sequential processes (CSP) programming model. The CSP programming model is modified by allowing the buffering of data being transmitted between processes, and the hardware and software processes communicate

primarily through buffered data streams that are implemented directly in hardware [16]. This buffering of data, which is implemented using single or dual-clock FIFOs generated by the compiler, makes it possible to write parallel applications at a relatively high level of abstraction, without the cycle-by-cycle synchronization that would otherwise be required. Impulse C is also flexible enough to support alternate programming models including the use of shared memory as a communication mechanism [16]. The selection of the programming model is depended on the requirements of the customers' design applications and the architectural constraints of the selected programmable target platform.

VHDL is designed specifically for Register Transfer Logic (RTL), so it, like ANSI C, is a higher-level language, but it has several disadvantages. VHDL requires deep hardware knowledge. Also, compared with C, VHDL is also not as productive for describing and debugging complex systems. Designs written in VHDL may not be portable between different FPGA types. Those drawbacks limit the use and the users of VHDL [15]. Impulse C provides a solution for that situation. Designers can write C programs in Impulse C and its compiler will translate and optimize the Impulse C programs into appropriate lower-level representations, including Register Transfer Logic (RTL) VHDL descriptions that can be synthesized to FPGAs, and standard C (with associated library calls) that can be compiled onto supported microprocessors through the use of widely available C cross-compilers compilers.

The design of an Impulse C program is pretty straight forward. Figure 14 shows the design of the process, indicating the steps from the design to final implementation of a user application. The user describes the function by writing Impulse C design files

which will be compiled by the Impulse CoBuilder to create equivalent HDL format hardware descriptions [16]. The interface will be supported with additional HDL files generated by CoBuilder. CoBuilder will also generate software language components to implement a software interface for efficient software and hardware communication.

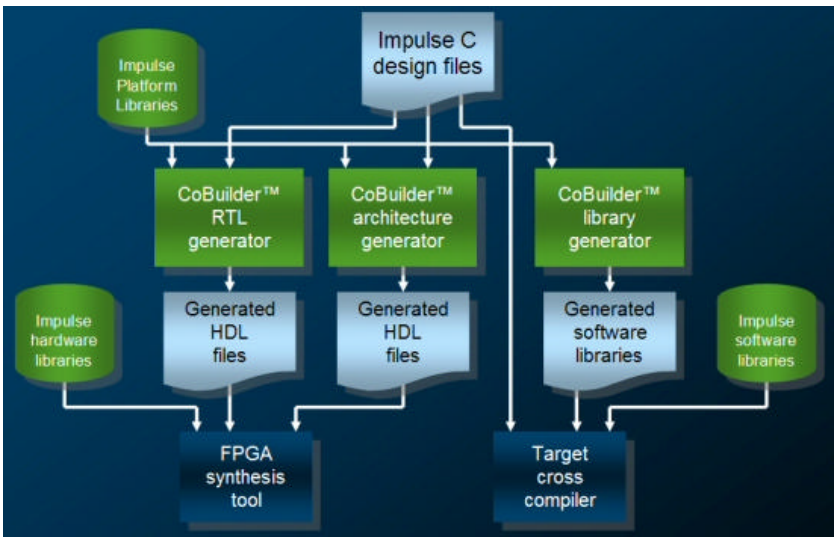


Figure 14: Design Flow of Impulse C [16]

The CoDeveloper Application Manager provides the programming environment for the use of Impulse C. It includes an Application Manager, Application Monitor, compatibility with common development environments (including Microsoft Visual C++, Metrowerks CodeWarrior and the GNU tools), platform-specific FPGA compilers and other resources for Impulse C application developers.

In all, there are many advantages for the Impulse C design. It gives C programmers a way to access to FPGAs. It allows the use of standard C tools, including debuggers. It also allows the easy creation and control of parallelism at all levels. Finally, it supports multiple FPGA-based platforms directly. Presently, Impulse C is widely used for applications including image processing and digital signal processing on

embedded systems, as well as for acceleration of high-performance computing applications including financial analytics, bioinformatics and scientific computing.

Design of the Impulse C Parallel Communication Project

In order to design an Impulse C application that goes between two boards, the user must specify the communication ports, the hardware and software processes and the input/output data streams. Although Impulse C provides a C-like programming environment, the user must learn to use multiple predefined functions. These predefined functions, along with two programmer-defined functions, make up the Impulse C Application Programming Interface (API) [16].

The Impulse C API lets the user define an application as a set of processes and create communication channels between those processes. Functions related to processes, streams, signals, registers, and shared memories make up the core of the API. Processes are the fundamental units of computation in an Impulse C application. Once they have been created, assigned and started, the processes in an application execute as independently synchronized units of code on the target hardware. Each Impulse C process has its own control flow which has access to its own local memory resources. Software processes are assigned to independently synchronized processors. In my design, I mainly used the co_process, co_stream, co_port and cosim_logwindow functions. In the Impulse C programming model, there are two important types of elements: processes and communication objects. A set of processes, connected to each other using communication objects, comprises the core of an Impulse C application. The predefined Impulse C functions that perform inter-process communication may be referenced in a software or hardware process. Thus, read/write operations on a data stream connecting

two processes can be performed by calling functions on co_stream objects within the processes. The co_stream function includes create, open, read, write, close and eos functions. Those functions handle all the processing of data streams including generate data streams, write data to the processes using streams and read the data from the processes. The co_port function provides the way to connect one end of an Impulse C stream to an HDL component that is not generated by CoDeveloper or, in the case of this project, to the external pins that will be used to connect to a second board. The other end of the I/O object must be connected to another Impulse C hardware process. In addition to the intrinsic functions for application programming described before, Impulse C also includes a collection of instrumentation functions, prefixed with cosim_ that can be used to instrument an Impulse C application for debugging and profiling purposes. When running an Impulse C application, the cosim_logwindow functions will communicate with the Impulse C application monitoring application to provide additional debugging capabilities [16]. A block diagram of the Impulse C project is provided in Figure 15.

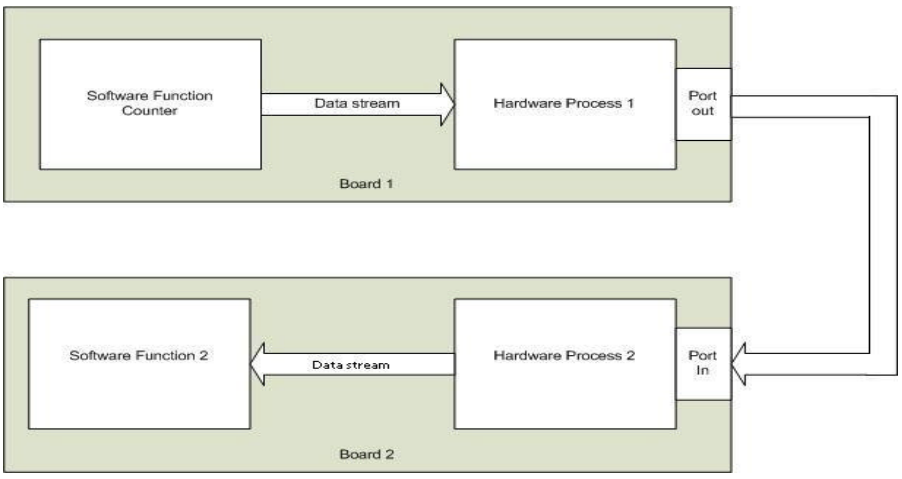


Figure 15: Impulse C Parallel Data Stream Process

Each board contains a program that is split into software and hardware processes. On the first board a software function generates the data which will be written to the hardware process. The hardware process will process the data stream and write the stream to the external hardware port. The data stream will be transferred through wires to the external hardware port on the second board. On the second board, the hardware process will open the external port to receive the data stream, and then the data stream will be read from the port. The data will be sent to another software function which will print out the results over an RS-232 serial interface.

In Impulse C CoDeveloper environment, the software function is defined by the software files. In the software files, I defined the counter which generates sequential numbers and write them to a data stream. The C code used by the software compiler for these functions is given below [17].

```
co_stream_open(count, O_WRONLY, INT_TYPE(16));     // open the stream
    printf("CPU is counting...\n\r");
    for ( i = 0; i < 1000 ; i++ ) {
          co_stream_write(count, &i, sizeof(co_int16));  // write the data to the stream
          printf("Number Generated: %d\n\r", i);
         }
co_stream_close (count);                                // close the stream
```

Hardware Process 1 reads the data from the software function and writes the data to the output port. Another important thing for the hardware process is to create and configure the port. It will open the data stream from the software process, read the data from the data stream and then write the data to the output stream which connect to the output port. Although in this example the main function of the hardware process is to connect the software data stream with the ports, in a normal reconfigurable computing application the hardware function would implementation a useful algorithm such as

digital signal processing, edge detection, or some type of bioinformatics calculation on the data stream. Part of the C code used by the hardware compiler for the hardware process is given below [17].

```
// Hardware Process
   co_stream_open(data_in, O_RDONLY, INT_TYPE(16));
   co_stream_open(data_out, O_WRONLY, INT_TYPE(16));
       while (co_stream_read(data_in, &value, sizeof(co_int16))= =co_err_none) {
              //data could be processed here
              co_stream_write(data_out, &value, sizeof(co_int16));
             }
   co_stream_close(data_in);
   co_stream_close(data_out);
// Hardware configuration
   co_stream data_in, d_out;    // define the data stream
   co_process hw1, Numgen;      // define the hardware and software process
   co_port  port_out;           // define the output port
   data_in = co_stream_create("data_in", INT_TYPE(16), BUFSIZE);
   d_out = co_stream_create("d_out", INT_TYPE(16), BUFSIZE);
   port_out = co_port_create("output_stream", co_output, d_out);
   Numgen = co_process_create("Number_Generater", (co_function)numgen, 1, data_in);
   hw1 = co_process_create("Hardware_Process_1", (co_function)Hw_Process, 2,
                                 data_in, d_out);
   co_process_config(hw1, co_loc, "PE0");      // specify the hardware processor
```

Once the programs are written they may be simulated to verify proper operation. At this point, all of the debugging is performed in C. Standard C debugging tools and the CoMonitor Application Monitor can be used to debug the programs. Once the application is verified, the Impulse C CoBuilder compiler can be called to compile the software and hardware C source files. If there are no errors, this generates the HDL files. The lower-level HDL file generated by CoBuilder includes an entity declaration which is generated as a direct result of an Impulse C process that was previously declared in the software and hardware functions. CoBuilder will create at least four physical ports for each logical input or output port in the Impulse C code. These ports are named xxx_rdy, xxx_en, xxx_eos and xxx_data. In my design, I needed dual-clocked FIFOs, so there's

another port co_clk that is generated as a result of the "Generate dual clocks" option specified in the Generate Options dialog [16, 18]. These five ports are summarized below:

1. xxx_rdy: This is an output signal. When the signal is high, the stream is ready to write or accept data (depends on the input/output stream).
2. xxx_en: This is an input signal. When the signal is asserted high, it will enable the stream write to or read from the process.
3. xxx_eos: This signal is sent form a stream when that stream is closed and will be received by another stream.
4. xxx_data: This is a signal vector contains the data which is to be read or written by the process from/to the stream.
5. co_clk: This is an input signal that is connected to the external clock.

These five signals (which have different behaviors for input and output streams) form the complete interface for one stream of an Impulse C process, as observed at the interface to the process itself. Figure 16 shows these interfaces of the data stream.

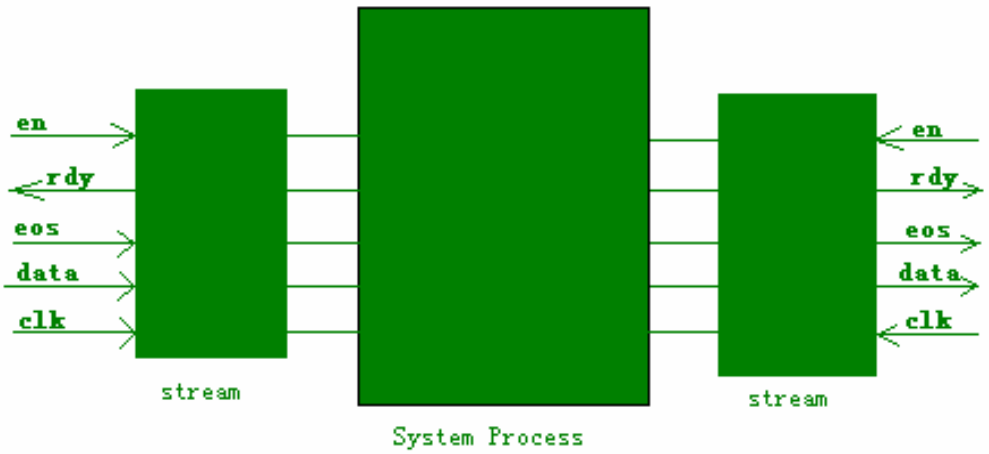


Figure 16: Interfaces of the Data Stream

Completing the Design in Xilinx Platform Studio

The Impulse C CoBuilder not only generates the HDL files but also generates an IP core that contains all the HDL files and the signal ports. The core is targeted to a specific FPGA family. To be useful this IP core must be added as a component to an embedded system. A complete embedded hardware system consists of processors, peripherals and memory blocks that are interconnected via processor buses. It also has port connections to the outside of the platform. The Xilinx Platform Studio provides a design environment that can be used to add the IP cores and integrate them with other components to an entire embedded hardware platform.

About XPS and EDK

Xilinx Platform Studio (XPS) is a suite of design tools based on a common framework which enables the design of a complete embedded processor system for implementation with a Xilinx FPGA device. It is the design development software provided in the Xilinx Embedded Development Kit (EDK). XPS consists of an interface and all the tools needed to develop the hardware and software components of an embedded processor system [10]. It also provides an entire simulation platform that enables the designer to perform system verification within the XPS environment.

The Xilinx Embedded Development Kit (EDK) is a component of the Integrated Software Environment (ISE). It includes the Xilinx Platform Studio (XPS) system tools suit for hardware design, the Software Development Kit (SDK) which is based on the Eclipse open-source framework for the embedded software application and the embedded processing Intellectual Property (IP) cores including processors and peripherals. EDK enables a designer to design a complete embedded processor system for the

implementation in a Xilinx FPGA device [10, 19]. Figure 17 shows how the tools operate together to create an embedded system.

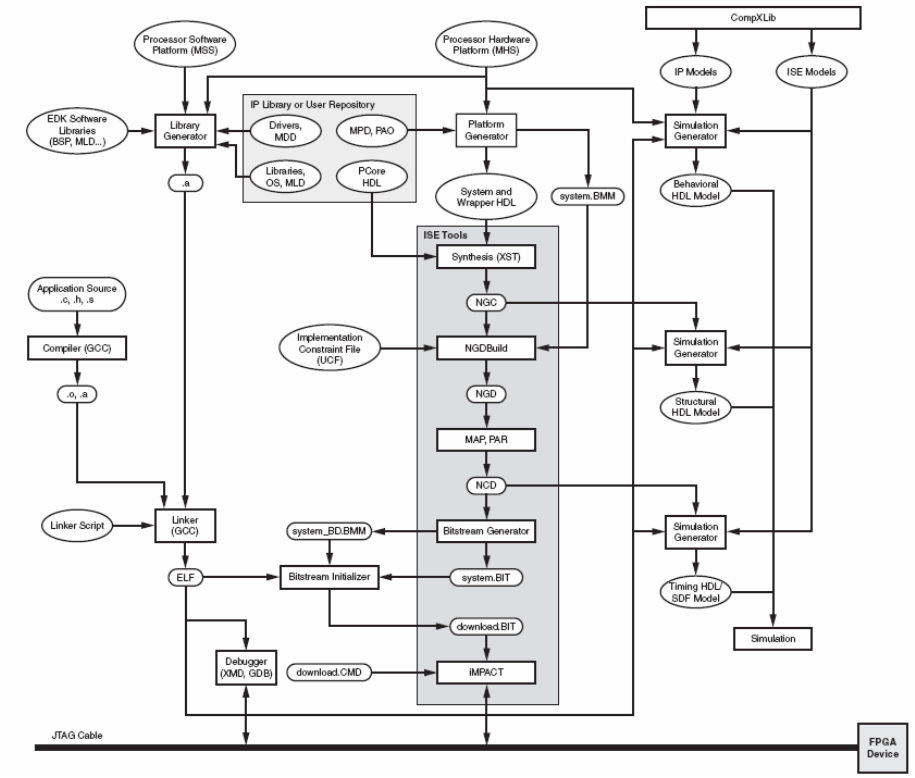


Figure 17: Embedded Development Kit (EDK) Architecture Structure [19]

Building the System with Parallel Hardware

When the design in Impulse C is completed, the next step is to import the Impulse C design into EDK. Prior to generating the hardware and software using CoBuilder, the user must select the desired platform which specifies the FPGA family and the FPGA-specific interface that will be used to connect the hardware to the embedded processor contained within the FPGA, or a generic interface that requires more effort on the part of the user to connect. For this project, I used the Xilinx Virtex II PRO PLB option in the drop down Platform Support Package selection of the Generate options. CoBuilder will

translate the C code into VHDL files which support the Virtex II PRO PLB bus, and then the hardware and software export files will be created in a directory that contains all the hardware and software files and the IP core [18]. That directory could be used by the XPS when create the new EDK project.

When creating a new design in EDK, there are lots of parameters we should specify, the first thing is to choose the target board. In my design, I use the Xilinx University Program Virtex II PRO boards, so the setting of the 'Select board' dialog should be: Board vendor: Xilinx; Board name: XUP Virtex II Pro Development System; Board revision: C. The next step is to choose the processor that I would use in my design. Here I use the PowerPC as my processor. On the PowerPC configuration page, I set the processor clock frequency to 100MHz. The bus clock frequency could equal to the processor's frequency, but in my design, I choose 25MHz as my bus clock frequency. The remaining settings were FPGA_JTAG, no On-chip memory, and no cache. In the IO device menu, I just kept the RS232 (UART) which will be the STDIN and STDOUT for the software functions. On the peripheral page I set the on-chip memory to 32K and then generated the project.

After generating the system, you must connect the IP core generated by Impulse C. In the left frame of the XPS window, there are three tabs: Project, Applications and IP Catalog. In the IP Catalog, the Impulse C generated IP core will be placed in the Project Local pcores folder. Double click that IP core to add it to the system, and then it will appear in the System Assembly View window. In the Bus Interface menu, the system components are listed. Each of them is connected to a system bus. Because the Impulse C project is designed for the PLB bus, you must find that IP core in the Bus Interface

menu and connect it with a PLB bus. Then, go to the Addresses menu, click the 'Generate Addresses' button, the system will automatically generate the system address map for all the system components including the IP core.

The last thing to do is to specify how to connect the external ports of the IP core. As described previously, the data stream of my Impulse C project contains five ports. Therefore, the IP core generated by the Impulse C CoBuilder should have these five ports. To find these, go to the Ports menu and click on the plus sign to expand the IP port list. Figure 18 gives a clear vision of the ports of the project.

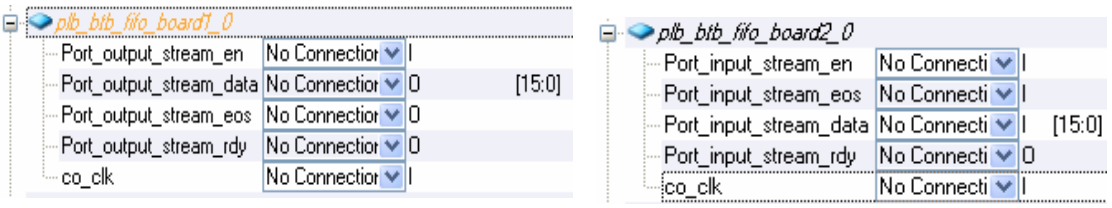


Figure 18: Project Ports of Two Boards

All these ports are configured in the VHDL files that are generated by the CoBuilder. What is needed next is to make the connections with these ports. Because these ports are all connected to the parallel hardware, they must be set as external ports. This will add the ports to the top-level entity of the VHDL description and allow the user to connect them by specifying the desired FPGA pins in the User Constrain File (UCF) in this system.

In this project, I used the parallel low speed expansion I/O ports to interconnect the XUP boards. These 40 pin I/O ports provide me enough signal interfaces for the Impulse C data stream. In this design, the output data from board1 is directly connected to the input data port. The output eos signal is connected with the input eos signal, the

output enable signal is connected with the input ready signal, the output ready signal is connected with the input enable signal and two external clocks should be connected with the clock from other board. Thus, with this logic, I could set the UCF file in XPS. Go to the Project window, unfold the project files and double click the 'system.ucf', add the constraints under the 'IO Device constraints'. Besides the configuration of the project ports, other system components (including processors, RS232 devices, etc.) should also be connected to a common bus (PLB or OPB). If the system configuration is completely done, the XPS will generate a block diagram that contains all the system hardware information. Figure 19 shows the system overview of board1. Board2 has the same architecture, with the exception that the IP block has a different name.

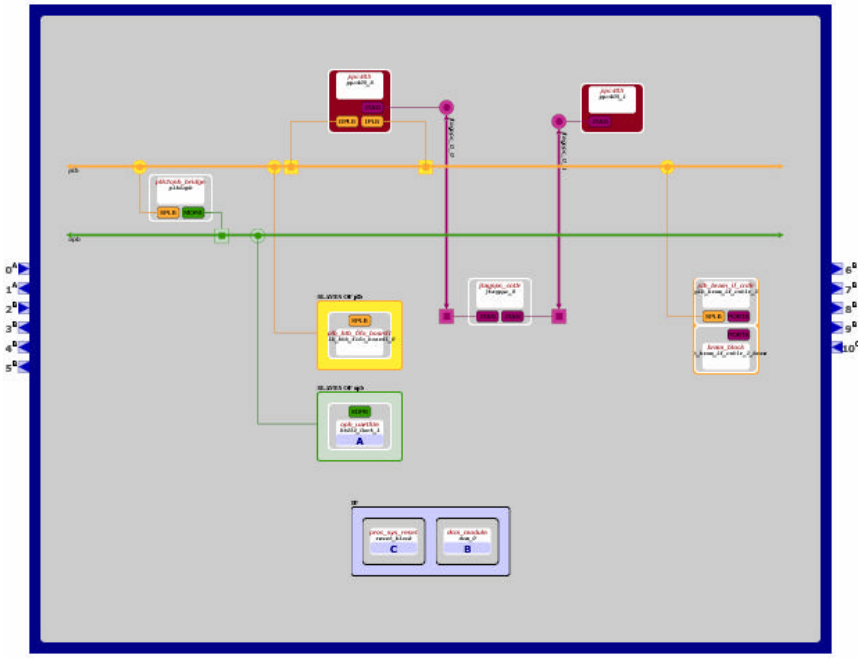


Figure 19: System Overview of the Project

The yellow line is the system PLB bus and the green line is the system OPB bus, all the components are connected to these two buses. The communication between these

two buses is established by the 'plb2opb' component. After the configuration of hardware files, the software files should added into the project. In this design, software files control and drive the data stream, establish the communication with the Personal computer through RS232 (UART). The software files are generated by the Impulse C CoBuilder. In the XPS Application window, we could add those software files (usually located on the code directory) to the 'Sources' menu. These C codes could be modified and compiled in the EDK by selecting 'Build all user applications' under the software menu. The software compilation will copy the software files (including header files) of all the system components to my design and compile them. The hardware compilation will map the user applications to the hardware and create the bitmap of the entire project. Thus, after the compilation, we can download the bitmap file to the target boards to see the implementation and results.

Simulation Results

This implementation of parallel hardware with Impulse C uses the XUP board's low speed parallel pins. The parallel hardware communication uses the FIFOs which are generated from HDL files by CoBuilder to implement Impulse C data streams. Figure 20 shows the simulation on one board.

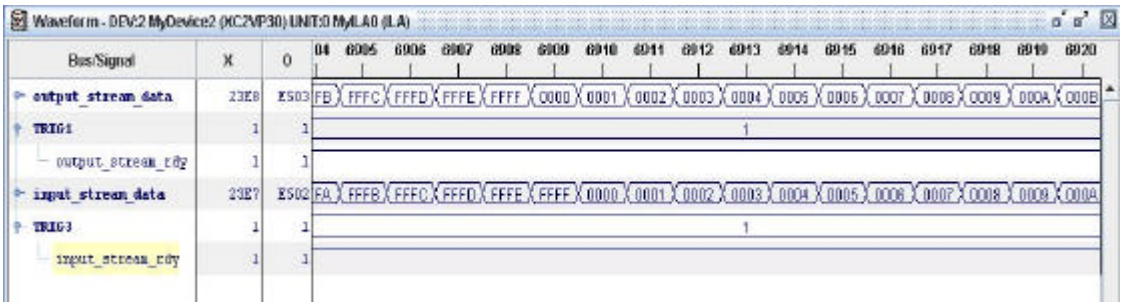


Figure 20: Simulation of Parallel Design on One Board

In the one board simulation, the transmit module and the receive module are all built in one EDK project. The data interfaces and the handshaking signals are connected directly follow the design. From Figure 16, we could see the simulation results of the parallel project. However, the simulation is based on one board and all the signals did not pass the real parallel hardware interfaces, therefore the simulation results may not represent the whole features of the entire system.

Implementation and Operation

After the implementation was completed, I examined the timing results which can be found in the in the file named system.twr. This file is located in a directory created by XPS named implementation. All timing results were satisfactory. For board 1, the minimum period for the IP core was determined to be 11.355 ns. Since the 25 MHz clock has a period of 40 ns, the clock speed is quite conservative. For board two the minimum clock period for the IP core was 11.835 ns.

Since I used the UART to monitor the signals, I could capture part of the data stream through the Hyper Terminal. Figure 21 and Figure 22 show portions of the transmitted data and received data for this project.

Figure 21: Data Transmitted from Board1

```
Number received: 988 from the input stream
Number received: 989 from the input stream
Number received: 990 from the input stream
Number received: 991 from the input stream
Number received: 992 from the input stream
Number received: 993 from the input stream
Number received: 994 from the input stream
Number received: 995 from the input stream
Number received: 996 from the input stream
Number received: 997 from the input stream
Number received: 998 from the input stream
Number received: 999 from the input stream
```

Figure 22: Data Received by Board2 in Parallel Communication

Hyper Terminal could monitor the real time results from the board, but its back scroll buffer lines are limited, thus we could not monitor the entire data stream from it. Therefore, I used Xilinx's ChipScope Pro tool to capture the data stream. ChipScope is used to monitor the real time signals which are running on the target board. I connected the ChipScope programming cable on the board2 to monitor the received data stream. In order to monitor the data stream more clearly, I use an endless counter on board1 to generate endless data stream to board2. Figure 23 shows a portion of the data waveform received by board2.

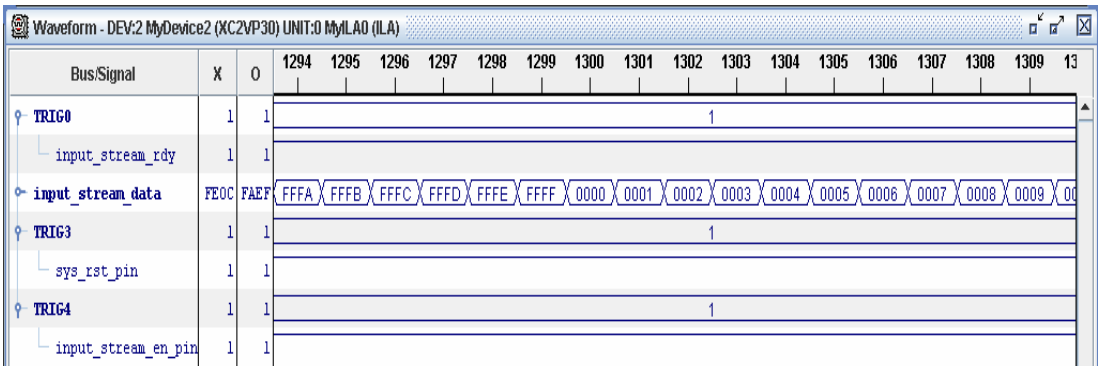


Figure 23: Received Data Stream Waveform by Board2

Problems with the Parallel Implementation

As was reported for the FIFO circuits in chapter 3, when the design was implemented and used to communicate between two boards a significant error rate was

measured. This source of the errors was investigated and determined to most likely be the result of interference. In the real design, interference may cause errors in the data transfer which are not caused by circuit design errors or timing flaws. There are many physical situations which may cause interference problems. For example, the wires of the cable may be too close to one another resulting in crosstalk. Also, in some situations the interface connection may be intermittent. Chipscope was used to monitor the effects of data erros. Figure 24 shows the data waveform without errors.

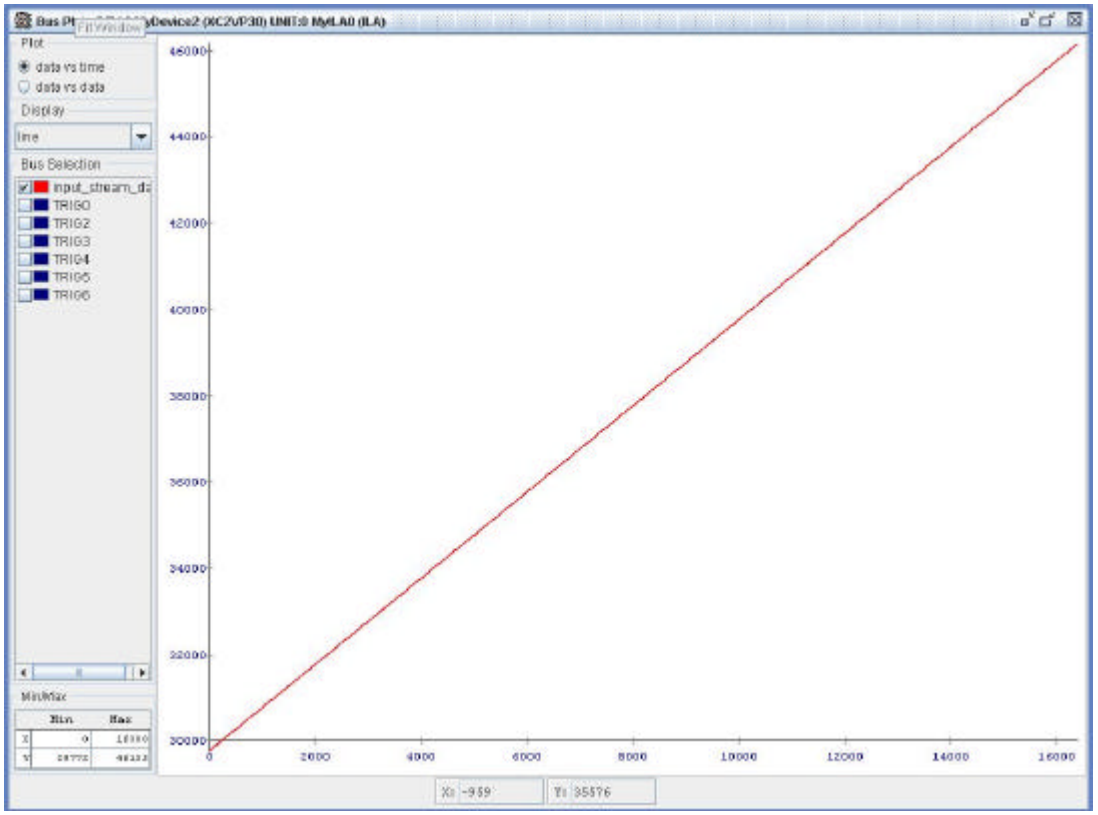


Figure 24: Data Waveform without Error

The data stream is generated by a counter; therefore the waveform should be a straight line. If there are errors on the data, some points will not follow the straight line. Figure 25 shows an example with errors.

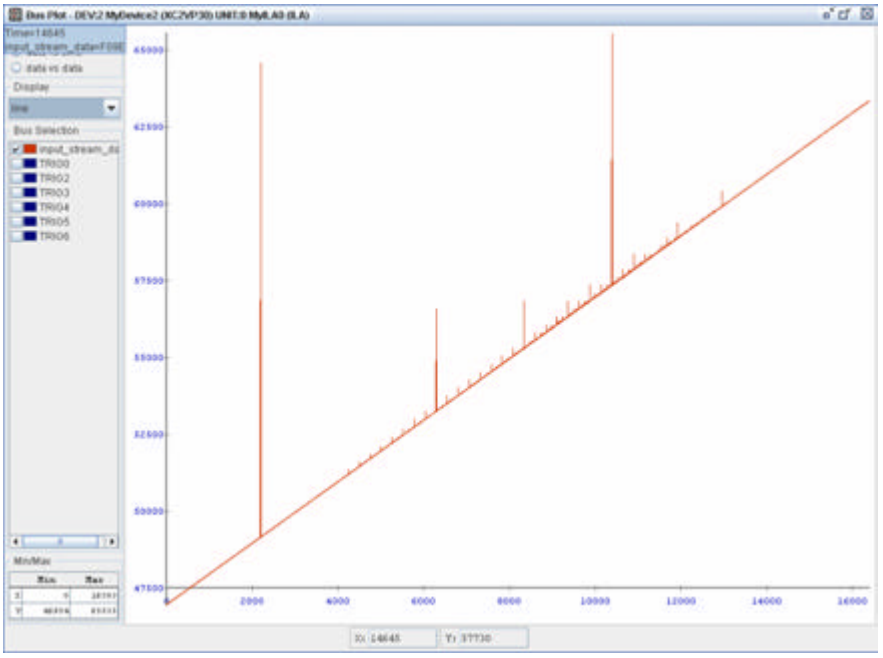


Figure 25: Data Waveform with Errors

In order to determine from the cause of the errors, I implemented the complete parallel communication system on one board, so data ports were connected directly without passing through the FPGA pins and parallel cable. For this circuit, the clock for the processor executing the software process was derived from the 100 MHz oscillator while a separate 75 MHz clock was used for the other side of the streams. This assured that the streams were crossing clock domains twice as they do when going between two boards. The use of the 75 MHz clock also stressed the timing more than the 25 MHz clock used when going between two boards. The test run on one board resulted in communication without any errors. Therefore, it seems likely that the communication errors that were encountered when talking between two boards came from the parallel hardware interfaces, the connection cables or interference. I designed a hardware module integrated to the EDK to check the error rate of the parallel hardware. I used another

counter to tracing the stream data which is received from board1 to board2. If they are the same, the pass signal counts. If error occurs, the fail signal counts. The width of data stream is 16, so the data range is 0~65535. In order to get satisfactory error rate, I need to capture millions of data from that data range, so the width of pass and fail signal are all 32. The data stream will not stop, and it starts from 0 when meet the maximum number. Figure 26, 27 shows the error tracing waveform and the results of the error tracing.

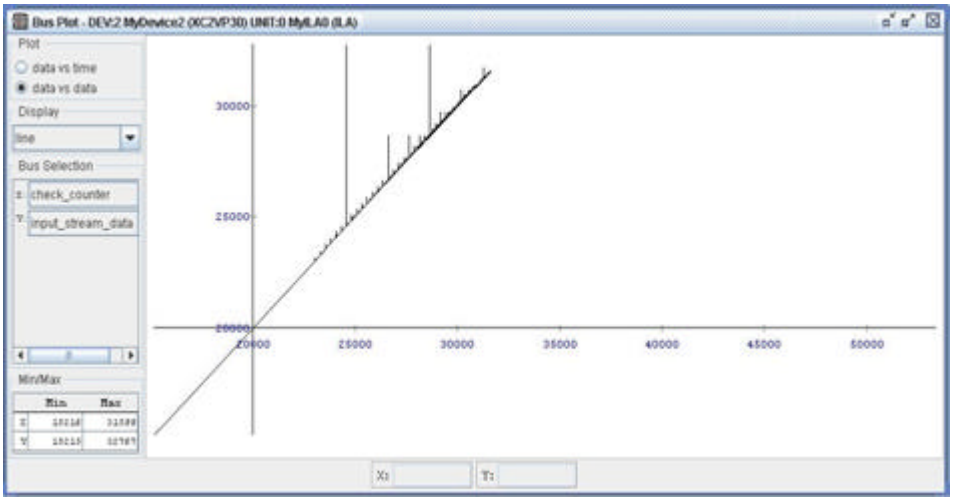


Figure 26: Stream Data Error Tracing Waveform

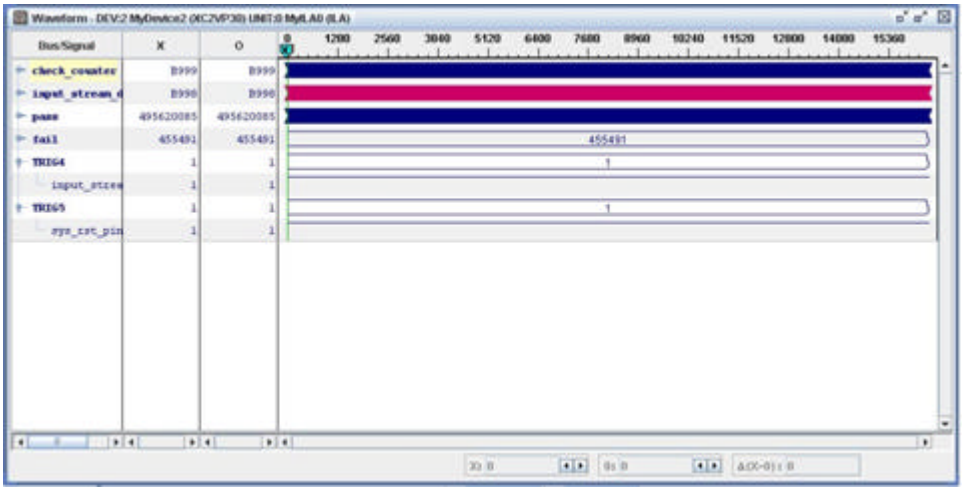


Figure 27: Results of Error Tracing from Chipscope

An error rate analysis was completed by repeating the test and tracking the number of attempts and the corresponding error rate. The results are shown in Table 2.

Table 2: Results of Error Test

PASS	FAIL	ERROR RATE
4560076	12366	0.0027
13560754	31631	0.0023
734627426	706730	0.0001
2457902894	2813547	0.0011
2486910270	3869839	0.0016
3515301914	4541729	0.0013
3608616487	6844981	0.0019

From the table, we could find the error rate after millions data tested. The error rate varies between about 0.1% to 0.2% and it is not stable. I believe this indicates that the interference is intermittent. The way to reduce such interference is using good cables which has good shield on each wire. In order to eliminate the physical situation that may cause the interference, I used good cable and ground multiple grounds of that cable to reduce the interference. I also use the falling edge of the clock as the board2's input co_clk signal to eliminating the timing errors. The circuit used a 75 MHz clock between boards. Table 3 shows the error rate results after first modification.

Table 3: Results of Error Test after First Modification

PASS	FAIL	ERROR RATE
8109723	2768	0.00034
677837452	369442	0.00054
1355411673	688117	0.00051

The clock frequency between boards could also make influence on the data transmitting. Therefore, I tried a slower 25 MHz clock as the inter-board clock frequency

and use a DCM module to assure the two clock edges were separated by the precisely 50% of the clock cycle. These modifications give me even better results. Table 4 shows the error rate results after second modification.

Table 4: Results of Error Test after Second Modification

PASS	FAIL	ERROR RATE
8953157	0	0
310544448	83959	0.00027
1571909020	984408	0.00063

With those methods, the error rate of data transfer is reduced to a very low level and the performance of parallel communication system is improved a lot, but the error transfer problem still exists. Another possibility, which was not investigated in this thesis, would be to implement a parallel interface using differential signals. However, this would double the amount of resources required for the parallel communication interface. Given these results, and the results that will be reported in the next chapter, I recommend using the serial method to transfer data instead of the parallel method in order to avoid data errors.

CHAPTER FIVE

Implementation of Serial Hardware Interfaces with the Aurora Protocol

Serial Communication and FPGA Support

Currently the clock speed of CPUs has entered the multi-gigahertz range; the primary bottleneck is the system interconnection. Most of the system interconnect used today uses parallel I/O technology. But the parallel I/O interface has obvious limitations. A large number of parallel I/O pins will not only take a lot of board area but also increase the design difficulty because the number of PCB layers is increased. As seen in the previous two chapters, timing becomes an issue at higher bandwidths. A bit arriving too early or too late could cause serious problems for the data transfer in parallel I/O technology [1]. Many serial communication protocols include a significant amount of circuitry to recover clock signals and overcome such timing issues. As frequencies increase these same measures must be applied to each bit of a parallel system. Therefore, the development of a new I/O technology that with high error correction ability and without bandwidth limitation is much more necessary now.

The new system connection evolution now leads us to the multi-gigabit serial I/O technology. One new interconnect technology is called Serial Advanced Technology Attachment (SATA). SATA, which communicates via a high-speed serial cable, is significantly different from Parallel ATA I/O interface technology. A SATA interface uses the embedded clock signal to enhance its ability of error correction. It can not only check the data but also check the transfer instructions and correct the existing errors

automatically. Thus, it is much more reliable to transfer data by using the SATA interface. Another advantage of SATA is that it could use high transfer frequency to enhance the bandwidth of data communication using less physical bit-width; only 7 pins [20]. That advantage reduces the board area required for the physical interface and potentially can reduce the number of PCB layers. Therefore, the SATA I/O interface not only enhances the data transfer ability but also reduce the system design costs.

During 2009, the Serial ATA has all replaced the Parallel ATA in most consumer PCs. In embedded application domain, PATA still remains, but along with the development of FPGA devices, the application requirements for multi-gigabit serial transceiver (MGTs) are growing. Current programmable devices that contain MGTs are able to implement a variety of high speed data transfer protocols. The Xilinx Virtex series of FPGA chips is well supported using various specialized embedded programmable modules. With highly flexible and programmable features, the Rocket-IO multi-gigabit serial transceiver could be easily integrated into systems using Xilinx Virtex-II Pro FPGA devices.

Rocket IO Transceiver

Rocket IO is a kind of high speed serial transceiver, it uses two differential signals to send and receive data. Thus, it can realize two simplex or one full-duplex data transfer. The Rocket IO transceivers on Virtex-II Pro FPGAs can communicate at speeds from 600 Mbps to 3.125 Gbps per line and deliver up to 62.5 Gbps aggregate baud rate. They support 8B/10B encoding (balance encoding), pre-emphasis, channel bonding and comma detection, programmable on-chip termination and soft IP cores for simplifying designing [21]. Therefore, Rocket IO transceivers are the ideal transceivers for the chip-

to-chip or board-to-board high speed serial data transfer. They require a power supply of 2.5V DC and use Current Mode Logic (CML) mode with a 50 ohm (or 75 ohm) internal resistor. The Pre-emphasis can compensate for high-frequency loss in the transmission media. That capability greatly improves the common-mode signal to noise ratio and the signal attenuation. Based on the Shannon formula:

$$C = W * \log_2(1 + SNR)$$ [22]

We know that when the channel capacity is a certain fixed value, then an increase of the channel bandwidth W will allow a decrease in the signal to noise ratio. Because the highest data transfer rate of Rocket IO is 3.75 Gbps, a very low signal to noise ratio will still result in acceptable performance.

Xilinx Virtex-II Pro FPGAs can contain up to 20 Rocket IO modules. Each of them is designed to operate at any serial bit rate in the range of 600 Mbps to 3.125 Gbps. Rocket IO modules consist of a Physical Media Attachment (PMA) and a Physical Coding Sublayer (PCS) [21].

There are eight clock inputs into each Rocket IO transceiver instantiation. The reference clocks REFCLK and BREFCLK are generated from an external source and presented to the FPGA differential inputs. REFCLK is designed for serial speeds from 600 Mbps to 2.499 Gbps and BREFCLK must be used for serial speeds of 2.5 Gbps or greater (up to 3.125 Gbps) [21]. One of these reference clocks is needed to drive a multi-gigabit transceiver (MGT). It is also used as an input clock of a Digital Clock Manger (DCM) to generate all of the other clocks for the MGT. A Rocket IO transceiver has the ability to encode eight bits into a 10-bit serial stream using standard 8B/10B encoding. The input data send to the transceiver will be encoded by the 8B/10B encoder, and the output data

will be decoded by the 8B/10B decoder. The 8B/10B encoder/decoder must be used in a pair. If this pair is not matched, the data will not be received correctly.

The input data to and the output data from that transceiver module are parallel data, so before they could be transmitted, they must be serialized or deserialized. The transceiver provides a serializer and deserializer to implement those requirements. The receiving data is converted from parallel to serial format by the serializer and transmitted on the TXP and TXN differential outputs. After accepting the serial differential data on its RXP and RXN inputs, data must be deserialized to parallel format by the deserializer first.

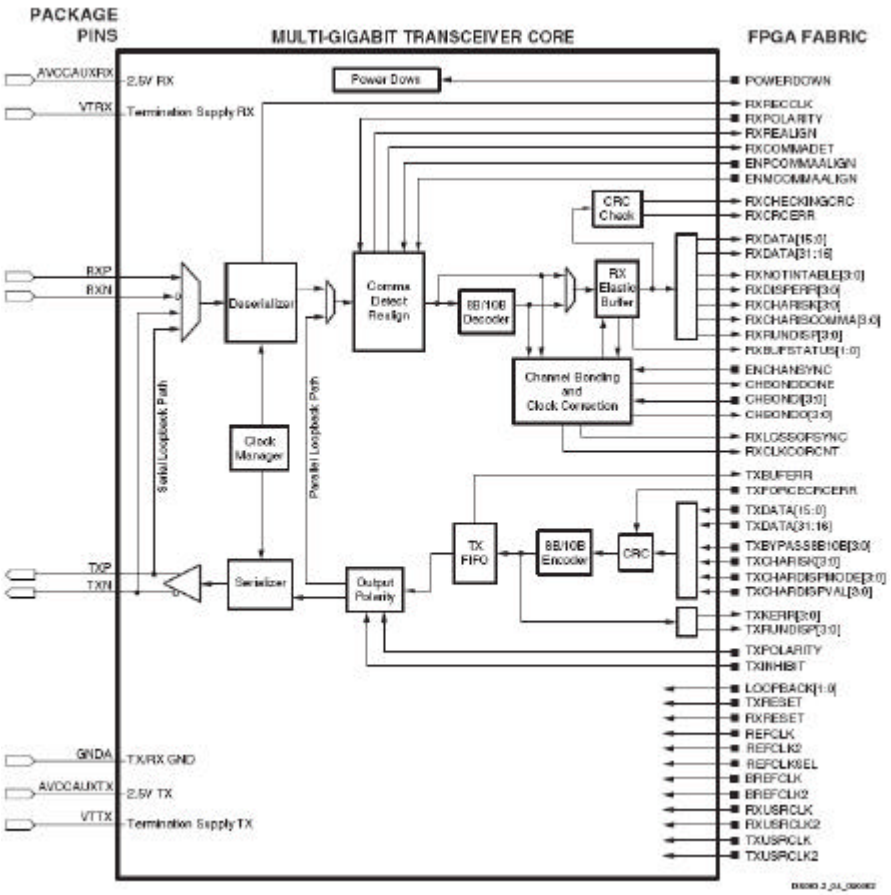


Figure 28: Rocket IO Transceiver Module Block Diagram [21]

The use of channel bonding is to aggregate several serial channels together to create one channel. Several channels are fed on the transmit side by one parallel bus and reproduced on the receive side as the identical parallel bus. This essentially implements a high-speed parallel interface where each data line in the parallel system has the circuitry required to recover the clock. The channel bonding circuitry must compensate for timing differences between the channels. Figure 28 depicts an overall block diagram of the Rocket IO Transceiver.

The Aurora Communication Protocol

The Rocket IO Transceiver module provides us the hardware support for serial communication in FPGA. We still need a protocol that supports and controls the hardware. Xilinx developed an open protocol which can be typically used in applications requiring simple, low cost, high rate, data channels. It is the Aurora communication protocol.

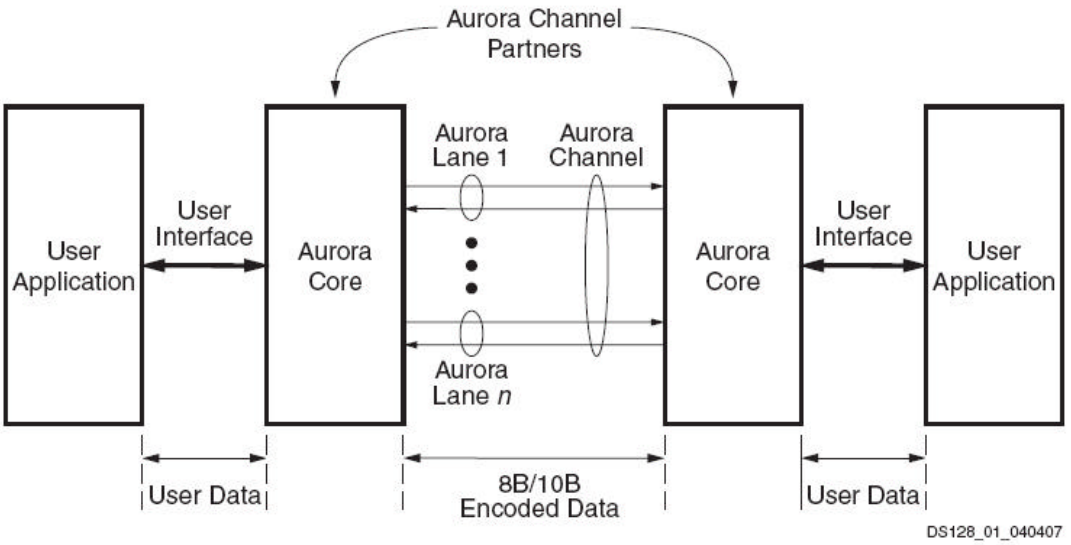


Figure 29: Aurora Channel Overview [23]

The Aurora communication protocol is a serial interconnection protocol which is designed to provide a transparent interface that can be used as a point to point serial data transfer method for the interconnection of high speed circuits or as the foundation for higher-level protocols. Aurora is a relatively simple protocol which only controls the link layer and the physical layer [23, 24]. These features let other upper-level protocols be easily applied on top of the Aurora protocol. Figure 29 shows the overview of an Aurora Channel.

The Aurora protocol can use one or more high speed serial channels to construct a higher speed channel. Connections can be full-duplex or simplex. It defines not only the physical interface, but also the package structure, the program used to embed other protocol package, data extraction and flow control. It defines the initialization program of the effective link, and it also describes the relative program that will disable the link which has excessive errors [23].

Aurora cores initialize a channel and applications can pass data across the channel as frames or streams of data. The framing user interface is Local-Link compliant. After initialization, it allows framed data to be sent across the Aurora channel. Framing interface cores tend to be larger because of their comprehensive word alignment and control character stripping logic. The streaming user interface allows users to start a single, infinite frame. After initialization, the user writes words to the frame using a simple register style interface that has a data valid signal. In my design, considering the signal interface of Impulse data stream, I chose the Aurora core stream interface for my design. The Aurora cores can be used in a very wide variety of applications such as chip-to-chip links, board-to-board and backplane links, one way connections and ASIC

connections. It is a very useful open protocol allowing a designer to design high speed serial data communication system interfaces. In summary, the Aurora core has features below [25]:

1. Provide 622 Mb/s to 100 Gb/s data throughput for data channels.
2. Supports up to 20 MGTs in Xilinx Virtex-II Pro/Pro X and Virtex-4 FX devices.
3. Supports 8B/10B encoding.
4. Low resource cost.
5. The framing and flow control is very easy to use.
6. Automatically initialize and maintains the channel.
7. Supports both simplex and full-duplex operation.
8. Framing with flow control or streaming user interface.

Design and Implementation with Aurora Hardware

In last chapter, I introduced the implementation of a board to board interface using parallel hardware. With the support of Impulse C, I could hook up the data stream with the parallel hardware provided by XUP board and realize a communication link between two boards. However, as described previously, parallel hardware has many disadvantages and is being replaced by new high speed serial communication technology. Thus, research on the board to board communications with SATA technology is very relevant. The hardware and protocol support for serial communication has already been discussed. Here, I will discuss the implementation of a serial board to board communication system using the Aurora protocol, Impulse C, and EDK.

Customize the Aurora Core

Since the Aurora protocol is an open protocol, I can modify it to meet my requirements. Xilinx Core Generator provides me a tool to customize the parameters of the Aurora core. The Aurora core is an IP core which supports the entire Aurora protocol, and it can be customized to meet a wide variety of requirements. Table 5 shows the specification of the Aurora Core parameters.

Table 5: Specification of the Aurora Core

Aurora Core Options	Parameters
Target Device	XC2VP30
HDL Source Type(VHDL/Verilog)	VHDL
Aurora Lanes	1
Lane width	2
Interface	Streaming
Special Features	None
MGT Placement	Row 1
Row 1 Clock	BREF_CLK

After the configuration, the CORE generator will create an Aurora core which could be imported to the EDK. In my design, I need to connect the data stream generated from Impulse C with the Aurora module. Like the Impulse C stream, the Aurora module has many interfaces, and those interfaces all have their special use. Since I used the Aurora stream module, the generated Aurora core is configured with a streaming user interface. Figure 30 shows the stream user interface.

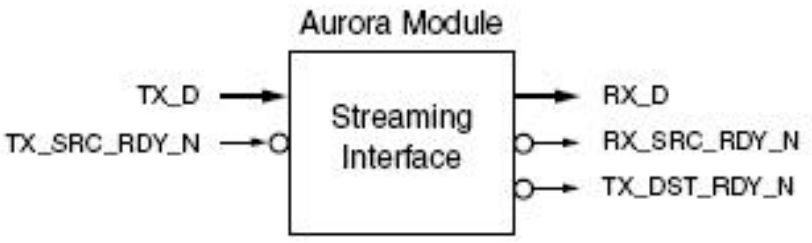


Figure 30: Aurora Core Streaming User Interface [24]

The streaming interface allows the Aurora channel to be used as a pipe. TX_D and RX_D are the data signal vectors. Other signals control the operation of the Aurora module. Signal TX_SRC_RDY_N is asserted low when the data is valid. Signal TX_DST_RDY_N is asserted low when the channel is ready to receive data. If the signal RX_SRC_RDY_N is asserted low, the data must be read immediately or it will be lost. Thus, when data is presented on the TX_D port and the signal TX_SRC_RDY_N is asserted low, the data is valid and ready to be written to the channel. If the signal TX_DST_RDY_N is asserted, the channel will start to receive the data. The received data will be stored in the Aurora core and the signal RX_SRC_RDY_N will be asserted, and data will be read through the RX_D port immediately [24].

Design Digital Clock Manager (DCM) Module

There are many different modules in my design, and the requirements of the clock frequency of those modules are different. Thus, it is necessary to design a module that could handle all the different clocks. In EDK, a digital clock manger module is automatically created when the system is created. That system DCM module can provide a system clock for my entire system.

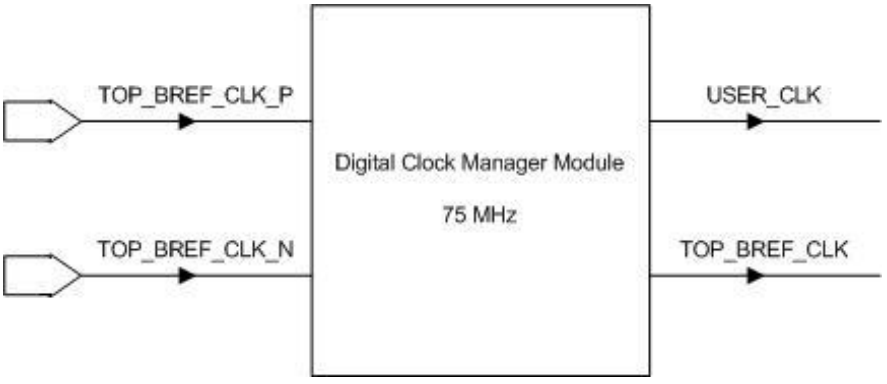


Figure 31: Digital Clock Manager User Interface

Since my designed modules still need different clock rates (e.g., user logic clocks data in and out of the Aurora core is 75MHz), I still need to design another simple clock manger module for my designed system. The designed DCM module contains two clock buffers [26]. Figure 31 shows the overview of my DCM module.

In this module, the input signals TOP_BREF_CLK_P and TOP_BREF_CLK_N are connected to the positive and negative MGT clocks which provide 75 MHz clock frequency. The 75 MHz SATA clock is obtained from a high stability (20 ppm) 3.3V LVDSL differential output oscillator, and the external MGT clock is obtained from two user-supplied SMA connectors. The outputs of the DCM are two buffered clocks with the same 75 MHz clock frequency. The output USER_CLK will be used as a common clock for the designed system module and another output TOP_BREF_CLK will be connected to the Aurora module as the main clock input [26].

Design the Entire System

In this communication system, the Impulse C data stream can not be directly hooked to the physical interfaces on the board. The Aurora core provides a bridge between the software generated data stream and the Rocket IO multi-gigabit transceiver. With the customized Aurora stream module, data can be transmitted through the high speed serial interface.

In order to simplify the interface I generated two FIFOs using Xilinx's CORE Generator program. The two FIFOs are TX_FIFO and RX_FIFO. The VHDL code instantiating the two FIFOs was added to the VHDL code instantiating the Aurora core. The whole Aurora peripheral includes two FIFOs and an Aurora stream module. The two FIFOs locate between the Impulse C data stream module and the Aurora stream module

as a buffered link. This buffered link is necessary because the On-chip Peripheral Bus (OPB) clock and the Rocket IO clock are independent on the XUPV2P board. We know that when the signal RX_SRC_RDY_N from the Aurora stream module is asserted, the data will be read through the RX_D port immediately or the data will be lost. Therefore, these FIFOs are designed as independent clock FIFOs to synchronize the different clocks and buffer the data stream to prevent the risk of data losing from the Aurora stream module. The function and features of the FIFOs are detailed described in chapter three. Initially I attempted to connect the dual-clock FIFO created by Impulse C directly to the Aurora interface. This resulted in communication errors. Using separate FIFO cores with additional control signals simplified the interface logic.

The entire Aurora peripheral contains both TX_FIFO and RX_FIFO no matter it's used on the transmit board or on the receive board. That's because the design is a full-duplex communication system. Although in my design, I didn't use the full-duplex function of the Aurora core, but that design extends the future implementation on that communication system. Since the FIFOs are embedded in the Aurora peripheral, signals from the FIFOs are connected with the Aurora stream module and the Impulse C data stream module. Thus, the input and output signals from the Aurora core which will be imported into the EDK are not from the Aurora stream module but from the TX_FIFO and the RX_FIFO. Besides the signals that will go outside of the Aurora peripheral, the inside signals which connected between the FIFOs and the Aurora stream module must be specified. The following VHDL code defines two FIFOs and the outside ports and specifies the connection with the Aurora module [26].

```
-- FIFOs:
tx_fifo_i : fifo_generator_v3_2
```

```
    port map (
     din => tx_fifo_din,
     rd_clk => USER_CLK,
     rd_en => tx_fifo_re,
     rst => Bus2IP_Reset,
     wr_clk => Bus2IP_Clk,
     wr_en => tx_fifo_we,
     almost_full => tx_fifo_almost_full,
     dout => tx_fifo_dout,
     empty => tx_fifo_empty,
     full => open,
     valid => tx_fifo_valid);
  rx_fifo_i : fifo_generator_v3_2
    port map (
     din => rx_fifo_din,
     rd_clk => Bus2IP_Clk,
     rd_en => rx_fifo_re,
     rst => Bus2IP_Reset,
     wr_clk => USER_CLK,
     wr_en => rx_fifo_we,
     almost_full => rx_fifo_almost_full,
     dout => rx_fifo_dout,
     empty => rx_fifo_empty,
     full => open,
     valid => rx_fifo_valid);

-- Initialize the ports
     tx_fifo_din <= DIN_TX;
     DOUT_RX <= rx_fifo_dout;
     tx_fifo_we <= WE_TX;
     rx_fifo_re <= RE_RX;
     FULL_TX <= tx_fifo_almost_full;
     VALID_RX <= rx_fifo_valid;
     EMPTY_RX <= rx_fifo_empty;

-- Connection between TX_FIFO and Aurora TX
     tx_src_rdy_n_i <= not tx_fifo_valid;
     tx_d_i <= tx_fifo_dout;
     tx_fifo_re <= (not tx_fifo_empty) and (not tx_dst_rdy_n_i);

 -- Connection between RX_FIFO and Aurora RX

     rx_fifo_we <= (not rx_src_rdy_n_i) and (not rx_fifo_almost_full);
     rx_fifo_din <= rx_d_i;
```

Then, with the specification of the VHDL code, the entire Aurora module has been created. The Aurora module has two domains; one domain contains all the user interfaces that I defined in the VHDL files (include signals from both TX FIFO and RX FIFO and the status signals from the Aurora core), and another domain is the serial data transfer domain which will be connected to the SATA interface on the XUP board. Figure 32 shows the main data and control interfaces of the Aurora peripheral.

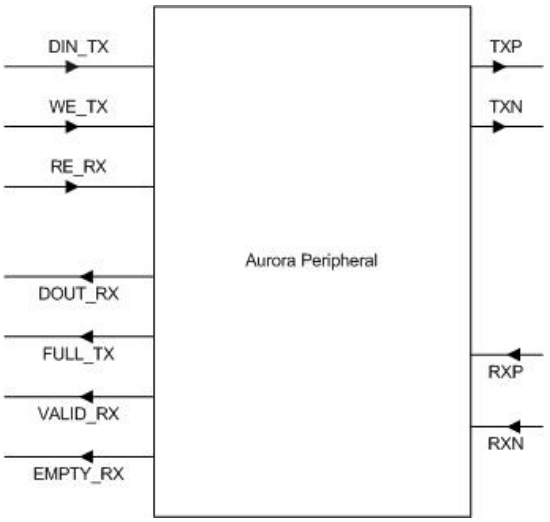


Figure 32: Aurora Peripheral Data and Control Interfaces

Import the Designed Modules to EDK

After all the system modules have been defined, I need to create those IP cores and import them into the EDK. Then, I can connect those designed modules in EDK. The IP core of Impulse C data stream module has already been created by the Impulse CoBuilder. The ports and the function of the Impulse C module have already been specified based by the Impulse C code. So I don't need to do any modification on that core. The Digital Clock Manager (DCM) module peripheral is created and configured in EDK. First, I created the DCM peripheral by using the 'Create and Import Peripheral'

menu in EDK. The DCM peripheral is very simple; it doesn't need interrupts, registers and FIFOs and it connected to the OPB bus. After creating the DCM peripheral, I added VHDL code to this peripheral to instantiate two clock buffers that will generate the clock sources BREF_CLK and USER_CLK for the Rocket IO MGTs and user applications. In my design, the module has two input ports and two output ports. Therefore, those ports must be defined in the VHDL files of that peripheral. The signal declarations and logic implementations should be added to the right position of the VHDL file of that peripheral. The idea of my design originates from the Xilinx RocketIO Transceiver User Guide. Part of the VHDL codes are shown blew [26].

```
-- Define the user ports
    TOP_BREF_CLK_P: in std_logic;
    TOP_BREF_CLK_N: in std_logic;
    TOP_BREF_CLK: out std_logic;
    USER_CLK: out std_logic;

--USER signal declarations
    component IBUFGDS_LVDS_25
       port(
              O: out std_ulogic;
               I: in std_ulogic;
              IB: in std_ulogic
             );
     end component;

     component BUFG
        port(
               O: out std_ulogic;
                I: in std_ulogic
             );
     end component;
  signal top_bref_clk_i : std_logic;
  signal user_clk_i : std_logic;

--USER logic implementation
  -- Differential Clock Buffer for top BREF_CLK
   diff_clk_buff_top_i: IBUFGDS_LVDS_25
        port map(
```

```
                    I => TOP_BREF_CLK_P,
                    IB => TOP_BREF_CLK_N,
                    O => top_bref_clk_i
                   );
-- BUFG used to drive USER_CLK on global clock net
  user_clock_bufg_i: BUFG
        port map(
                     I => top_bref_clk_i,
                    O => user_clk_i
                   );
  TOP_BREF_CLK <= top_bref_clk_i;
  USER_CLK <= user_clk_i;
```

The Aurora peripheral is also created in EDK. Based on the peripheral wizard, EDK will create a template for me. The Aurora peripheral also doesn't need interrupts, registers. Since the Aurora peripheral is connected with the Impulse C module which already contains FIFOs inside the module, it doesn't need the read and write FIFOs.

The Aurora peripheral template will be created by the end of peripheral configuration in EDK. Then, I will modify the template to include the VHDL files of Aurora core and TX/RX FIFOs. Although I don't need to create the read and write FIFOs for the Aurora peripheral, the TX and RX FIFOs however must be created and placed into the user_logic.vhd file manually. In chapter three, I detailed described how to create the FIFO component by the Xilinx CORE Generate. Thus, here, I just follow the steps to create the FIFOs that I will use to implement the TX and RX FIFOs. The CORE Generator will generate a file which named "fifo_generator_V3_2.ngc" after the FIFO is generated. I will use that file for my Aurora peripheral to implement the TX and RX FIFOs. The Aurora core is also created by the CORE Generator. I have already discussed it in the previous paragraph. The CORE Generator will create Aurora core source files after the core is generated and I needed to copy these files (include source files and cc_manager files) into the Aurora peripheral source folder within my XPS

project. After copy those files into the peripheral, the .pao file of that peripheral must be modified. The .pao file contains all the source files that compose the Aurora peripheral. Those source files which will be included in the .pao file must be listed in exactly hierarchical order. The components at the top of the hierarchy are listed at the bottom of the file. Here's the files listed blew [26].

```
lib aurora_mgt_v1_02_a aurora_pkg vhdl
lib aurora_mgt_v1_02_a channel_error_detect vhdl
lib aurora_mgt_v1_02_a idle_and_ver_gen vhdl
lib aurora_mgt_v1_02_a channel_init_sm vhdl
lib aurora_mgt_v1_02_a error_detect vhdl
lib aurora_mgt_v1_02_a sym_dec vhdl
lib aurora_mgt_v1_02_a sym_gen vhdl
lib aurora_mgt_v1_02_a chbond_count_dec vhdl
lib aurora_mgt_v1_02_a lane_init_sm vhdl
lib aurora_mgt_v1_02_a rx_stream vhdl
lib aurora_mgt_v1_02_a tx_stream vhdl
lib aurora_mgt_v1_02_a global_logic vhdl
lib aurora_mgt_v1_02_a phase_align vhdl
lib aurora_mgt_v1_02_a aurora_lane vhdl
lib aurora_mgt_v1_02_a standard_cc_module vhdl
lib aurora_mgt_v1_02_a aurora_201 vhdl
```

With those listed files, the Aurora protocol could be run properly. But those VHDL files only describe the inside functions of the Aurora core, which is not enough for the user implementation. Therefore, I needed to modify the Aurora peripheral template to specify the user ports, instantiate the Aurora core and the TX/RX FIFOs. Part of the VHDL code about the connection between FIFOs and the Aurora core has already been listed before. I also needed to design the user ports and make connection with each of the Aurora core inside module. The entire codes will be listed in the Appendix C.

After the configuration of each module, the rest thing is to connecting these modules together. Impulse C module will be hooked up to the PLB bus. The Aurora peripheral and the DCM module will be hooked up to the OPB bus. The Impulse C

module will send data stream to the TX FIFO of the Aurora peripheral, and the TX FIFO pass the data to the Aurora core. The Aurora core will serialize the data stream and transmit them through the SATA ports. On another board, the same Aurora core will receive the data and deserialize them, then transmit the deserialized data stream through the RX FIFO to the Impulse C module. Figure 33 shows the brief design of my Aurora communication system.

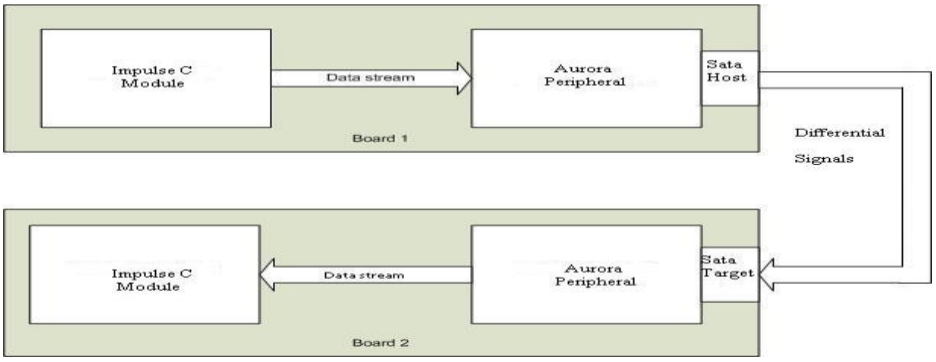


Figure 33: Aurora Communication System

On each board, the design contains combination logics and FIFOs. Each Aurora Peripheral contains both TX_FIFO and RX_FIFO, and the Impulse C module also could have a receiver function on each board. That design is based on the full-duplex communication consideration. In my project, I just use one board to transmit data and another board to receive data, so I left those functions unused. Figure 34 shows the block diagram of my designed entire system on one board in EDK. The block diagram shows the brief connection of each board. The Impulse C module uses PLB (Processor Local Bus) bus to handle the data flow, and the Aurora peripheral which includes an Aurora streaming module and two FIFOs uses the OPB (On-chip Peripheral Bus) bus to control the data flow. Both of them are controlled by the PowerPC.

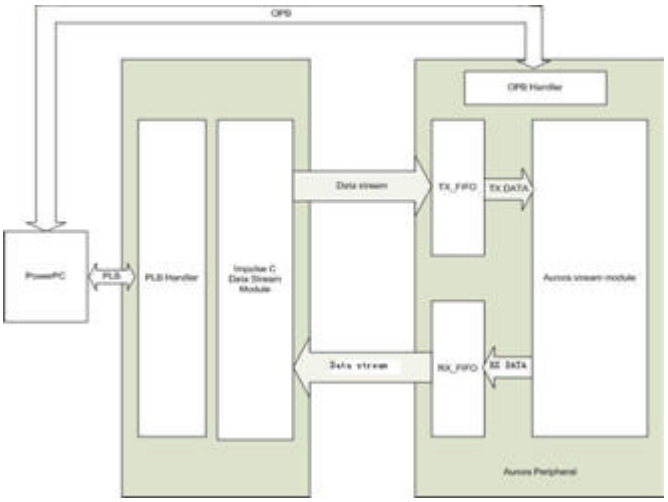


Figure 34: Communication System Block Diagram

However, signals from those modules could not be connected directly. There are still some simple logics between those modules. We could use VHDL to design the connection logics inside the module, but it's not the way to design the connection logics outside the modules. Signals come from each module could not be specified and designed in VHDL in EDK. Therefore, I needed to design other modules which contain the logic that was required to import to the EDK. In my design, there are some 'AND' and 'NOT' gates between the connections of the Impulse C and Aurora module. These gates can be seen in figures 35 and 36 below.

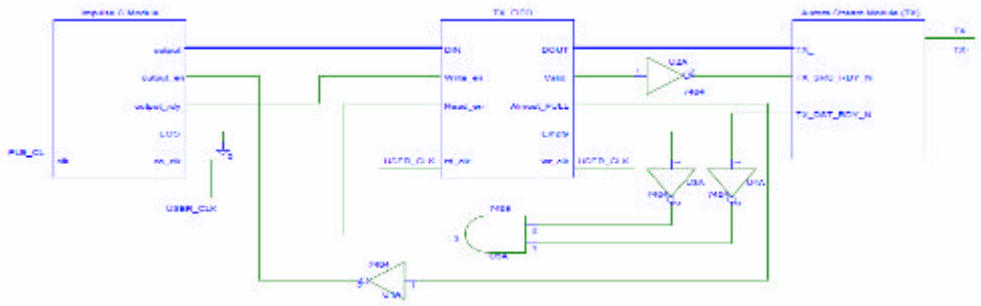


Figure 35: Schematic of Board1

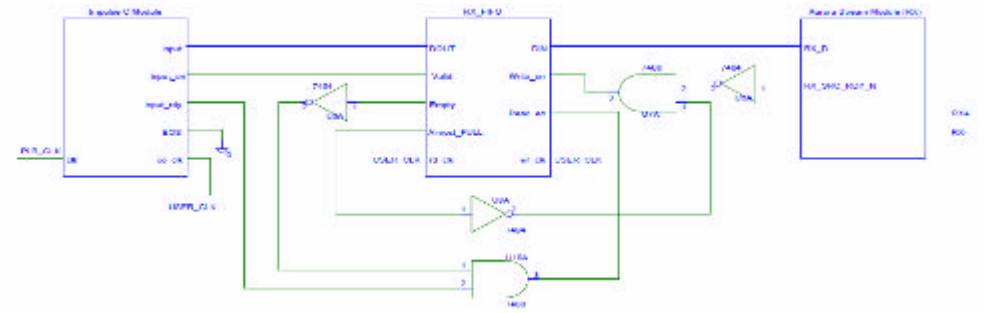


Figure 36: Schematic of Board2

The process of creating an IP core is the same as before, by using the peripheral wizard, the peripheral template is created, and then I needed to modify the VHDL files to specify the user ports and design the function of the peripheral template to meet my requirements.

When I add a peripheral using the Peripheral Wizard I must pick a bus (either the OPB or PLB) on which to connect the peripheral. Simple components, such as a 'NOT' gate do not use any of the bus' signals. In that case, none of the bus circuitry is synthesized. In other cases, the imported device may only use a few signals, such as the OPB clock and OPB reset signals. Only those portions of the bus interface circuitry that are required are synthesized.

The VHDL files will be compiled by EDK before the IP core is imported to the project. With these logic gates, I could implement my project. In the transfer domain on board1, the output data stream from Impulse C module will be directly hooked up to the data interface DIN_TX of the Aurora peripheral. When the FULL_TX signal from the Aurora peripheral is asserted high, the TX_FIFO must stop read data from the Impulse C. Therefore, this signal should be used to drive the output_stream_en signal of the Impulse C module. The logic is when TX_FIFO is not full, the output_stream_en signal will be

asserted so that the data stream could be read from the Impulse C module. If the signal output_stream_rdy is asserted, that means the stream is ready to write data, and this signal is connected directly to the WE_TX port to drive the write enable signal of the Aurora peripheral.

The received data stream will be transmitted through the TX_FIFO to the Aurora stream module. The inside logic of the Aurora peripheral between the FIFOs and the Aurora stream module has already been presented in previous paragraph. The Aurora stream module will serialize the received data stream and transmit them to the SATA ports. On another board, an Aurora stream module will receive the serialized data stream, deserialize them and transmit the data to the RX_FIFO.

The output data port DOUT_RX is connected to the input stream data port of the Impulse C module. The EMPTY_RX signal of the FIFO and the input_stream_rdy signal will drive the RE_RX signal of the FIFO. The logic is when the RX_FIFO is not empty and the Impulse C module is ready to accept data, the read enable signal will be asserted and the data from the RX_FIFO could be read to the Impulse C module. The VALID_RX signal will be wired directly to the input_stream_en signal so that when the data is valid, it will enable stream read from the FIFO. The entire schematic which shows the logic will be attached in Appendices.

In order to test the system, I tried the simulation on one board to see the communication results and monitored it by Chipscope. Figure 37 shows the simulation results captured by Chipscope. The output_stream_data is the 16-bit data stream from the counter to the first Aurora Module. The aurora_mgt_1_dout is the output data stream of the second Aurora Module.

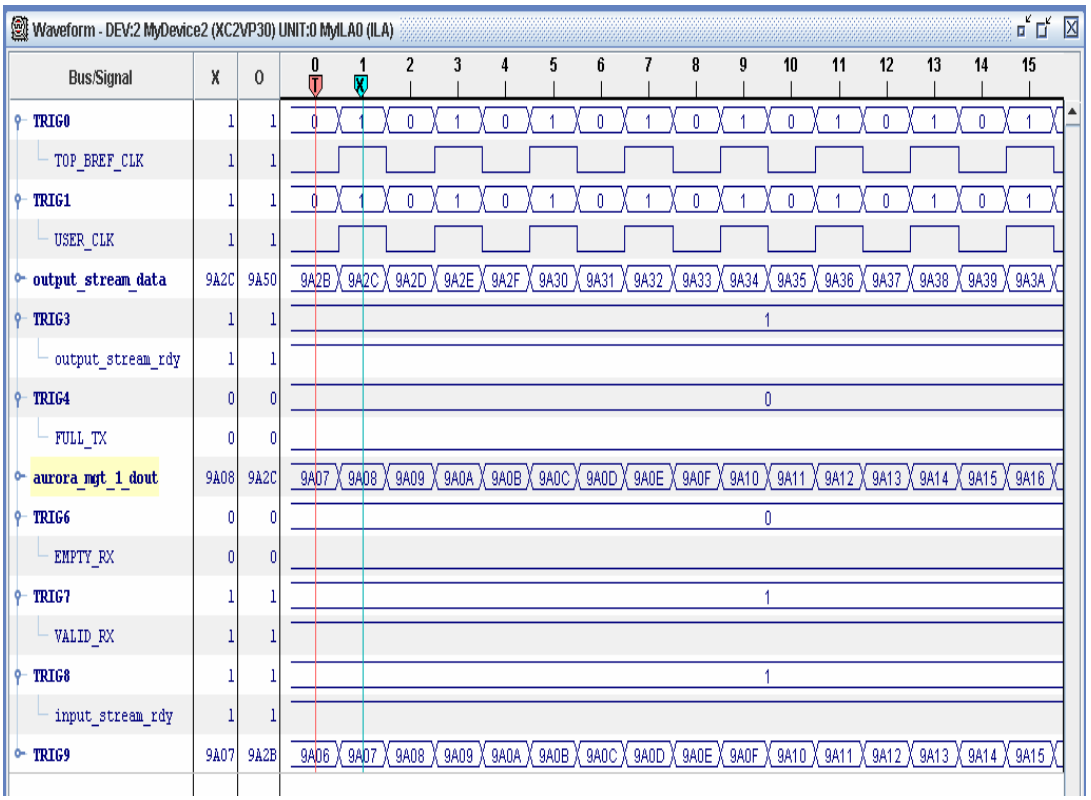


Figure 37: Chipscope Simulation Results of the Aurora System

We could see from the figure that the X cursor placed output_stream_data to the first Aurora module is 9A2C, and the output data from the second Aurora module is 9A08. Since there are many combination logic and two FIFOs between the counter and two Aurora Peripherals, data latency does exist. Figure 38 shows the data latency of the Aurora communication system. The X cursor placed the data 9A2C from the output_stream_data and the O cursor placed the data 9A2C from the output of second Aurora module. From the figure, we could clearly see that the data latency is 36 clock cycles (?(X-O)), which means the sent data from the counter on board1 will be received by board2 after 36 clock cycles. Since the FIFO depth is 1024, the data latency will not cause any transfer problem on the communication system.

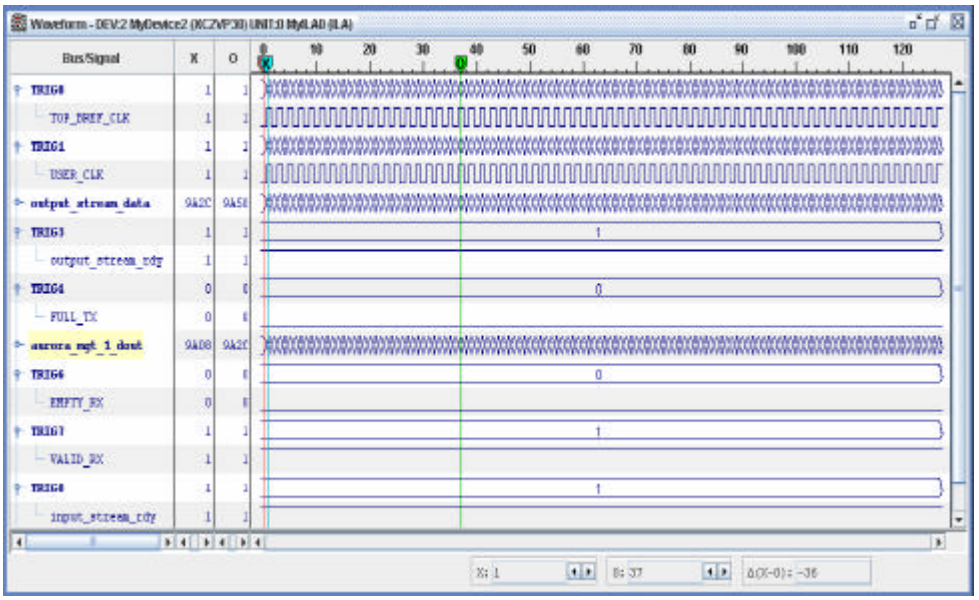


Figure 38: Data Latency of the Aurora Communication System

The entire project on one board not only established a communication channel between the Impulse C module and the Aurora module but also established a bridge between the software application and the hardware implementation. In EDK, the software source files are from the Impulse C project. I used the software files to generate and control the data stream. Therefore, before creating the bit-stream file, I needed to compile my software files first. The software compilation will copy the software files (including header files) of all the system components to my design and compile them. Then, before the hardware compilation, the last thing to do is to create the UCF files to establish the connection between the Aurora serial interfaces and the hardware serial ports. The embedded Rocket IO transceivers are used for the high speed serial communication. Three of them are equipped with the Serial Advanced Technology Attachment (SATA) connectors. These SATA connectors are split into two interface formats, two HOST ports and one TARGET port. The HOST port must be connected to

the TARGET port. The TARGET port interchanges the transmitting and receiving differential pairs to allow two XUP Virtex-II Pro Development Systems to be connected as a simple network. Unlike the parallel ports, the SATA connectors only have two differential signals for transmitting or receiving data. I don't need to specify each signal one by one, what I need to do is just initialize the MGT Locations of those differential signals. In my project, I used HOST SATA 0 interface on board1 and used TARGET SATA 1 interface on board2. Thus, according to the XUPV2P User Guide, the MGT location for board1 is X0Y1 and for board2 is X1Y1. Therefore, I must set the timing constraints for the MGT recovered clock. Since the SATA data rate is less than 2.5 Gbps, the 75 MHz clocks could have been supplied in the REFCLK inputs.

After the software and hardware compilation, I downloaded the bit files to the target board. Since the parallel data will be serialized and transmitted through the differential signaling links, the noise will be cancelled by the differential receiver. Although the differential signals could not be monitored directly by the oscilloscope, we still could use UART to monitor the process of actual data communication. In the serial communication, I use the same Impulse C program to generate the same data stream (similar to figure 16 in chapter 4). Using this approach, I obtained the same results from board2. Figure 39 shows the data received from board2.

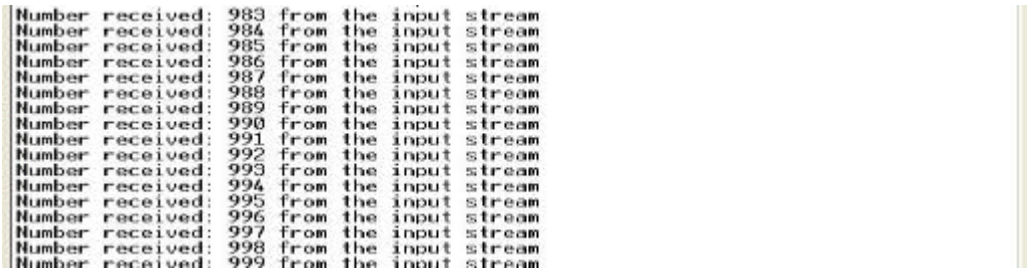


Figure 39: Data Received from Board2 in High Speed Serial Communication

The Hyper Terminal program could only show part of the data from the board2, but we also could use Chipscope to monitor the waveforms. By testing multiple times, I found that the Aurora system can transfer data without errors. Figure 40 and figure 41 shows the received data by board2 in Aurora communication system.

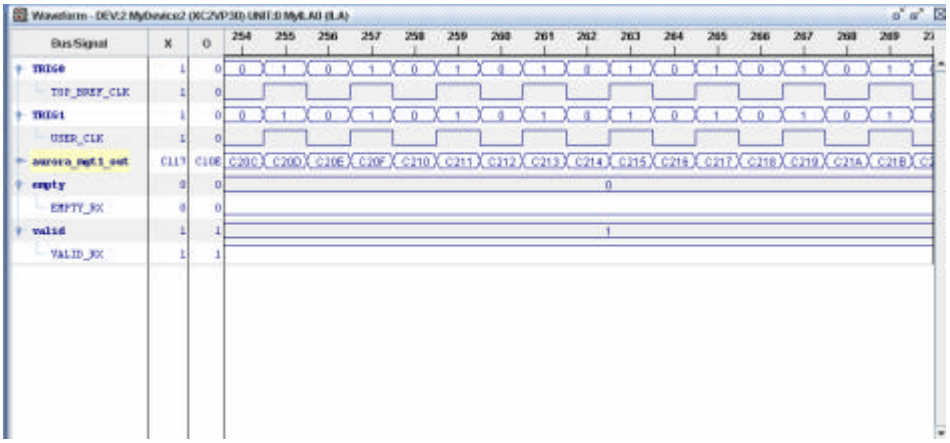


Figure 40: Received Data by Board2 in Aurora Communication System

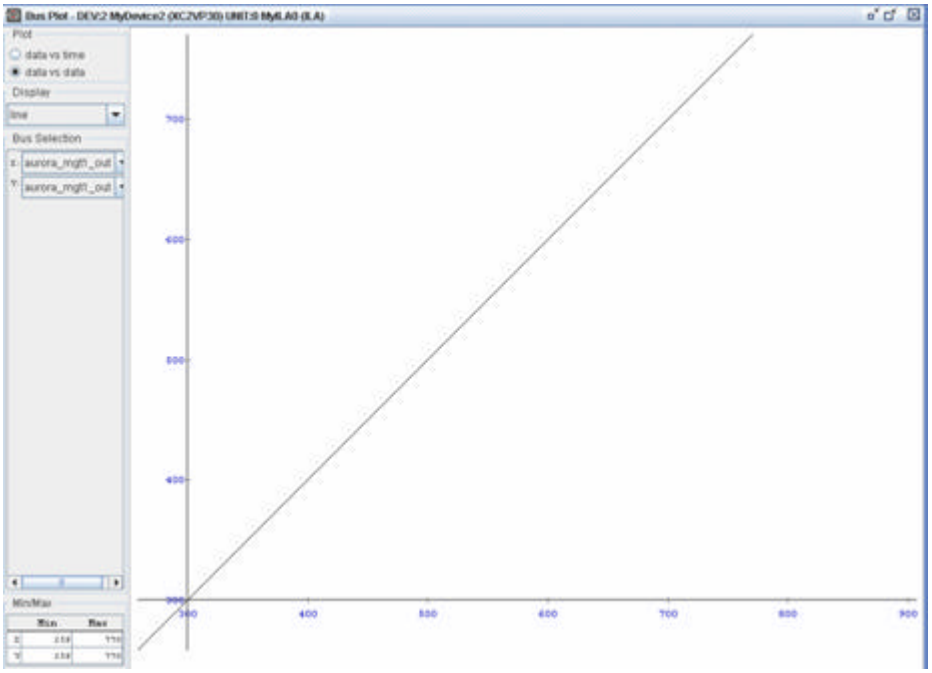


Figure 41: Bus Plot of the Received Data

CHAPTER SIX

Discussion of the Results

The implementation of parallel hardware with Impulse C uses the XUP board's low speed parallel pins. The parallel interfaces of boards are connected directly with an IDE cable. In my design, the Impulse C module can transfer data without other protocol support. Therefore, the parallel communication system consumes less internal FPGA resources. However, the parallel transfer method still has some problems. One of them is that interference causes data errors. A second problem is that the parallel method transmits data by using several parallel pins and therefore requires more FPGA pins and a large area of each board for the connection of the data pins. This makes the system layout design more difficult.

Unlike the parallel communication system, the Impulse C data stream could not be connected to the SATA ports directly. The Rocket IO transceiver and Aurora protocol are required in order to implement my design. To support the Aurora module, I needed to design several additional hardware modules. In addition to increasing the design difficulty, this approach requires more internal FPGA resources for the implementation of the design.

The transmission of data at high speed over electrical connections is very susceptible to noise and therefore designers use a number of techniques in order to reduce the impact of noise. One such technique used in SATA links is differential signaling. Since the parallel data will be serialized and transmitted through the differential signaling links, the noise will be mostly cancelled at the differential receiver. Although the

differential signals cannot be monitored directly by an oscilloscope, we can use a UART to monitor the process of data communication.

Similarly, the parallel interface could be implemented using a differential pair for each data bit and control signal. It is expected that this would result in error-free transmission. However, this would result in a doubling of the FPGA pins and circuit board resources. On the Xilinx XUP board this could have been accomplished using the high-speed parallel interface connector, J37, which provides connections for 20 differential signals and 3 clock signals. However, given that the serial interface proved easy to implement and provided an error-free data link that could transfer 16-bits during every 100 MHz clock cycle, the differential parallel interface was not implemented.

All the data generated from the Impulse C module will be sent to the hardware processing module which is inside the Impulse C module. The hardware processing module can process the data stream and then transmit the data to the hardware communication module. On the receiving board, the communication module will receive the data and then send it to another hardware processing module. The hardware processing function inside the Impulse C module provides us an environment for data processing. In my design the main focus was on data communication and I did not focus on any type of data processing. However, this functionality is provided for future use. Although hardware processing functions could also be realized by using a generic hardware description language, Impulse C can optimize the design functions to efficiently use the board's resources.

The design of the high speed serial communication system is much more complicated than the parallel system since I needed to design many hardware modules to

make the connection between the Impulse C module and the Aurora module. Therefore, the high speed serial communication system requires more internal FPGA resources to implement the design. Compared with the serial communication system, the architecture of the parallel communication system is simpler. Thus, the parallel communication system consumes less internal FPGA resources than the high speed serial communication system. In order to learn the detail difference of the system utilization of each board for each communication systems, I used EDK to record the utilization of the system resources. EDK can list the system utilization of each project and each board after the compilation of the system. Table 6 shows system utilization of each board for both two communication systems.

In my test, both of the two communication systems will work for practical applications and are efficient in transmitting data. The parallel system requires less internal FPGA resources than the high speed serial system and the design of parallel system is simpler than the serial system. However, the parallel communication bandwidth is limited and the expansion in terms of the number of required pins will occupy much more board area and increase the possibility of crosstalk interference. Compared to the parallel system, the serial system requires more internal FPGA resources due to the increased design complexity. However, the serial design is very attractive in terms of reducing communication interference. In fact, the differential scheme used by the serial method reduces interference to a large enough degree that significantly higher data rates can be achieved compared to the parallel method.

Table 6: Design Utilization Summary of Boards for Two Communication Systems

Design Utilization Summary	Parallel Board1	Serial Board1	Parallel Board2	Serial Board2
Number of BSCANs	1 out of 1 100%	1 out of 1 100%	--	1 out of 1 100%
Number of BUFGMUXs	5 out of 16 31%	5 out of 16 31%	4 out of 16 25%	5 out of 16 31%
Number of DCMs	1 out of 8 12%	1 out of 8 12%	1 out of 8 12%	1 out of 8 12%
Number of External DIFFMs	1 out of 276 1%	1 out of 276 1%	1 out of 276 1%	1 out of 276 1%
Number of LOCed DIFFMs	1 out of 1 100%	1 out of 1 100%	1 out of 1 100%	1 out of 1 100%
Number of External DIFFSs	1 out of 276 1%	1 out of 276 1%	1 out of 276 1%	1 out of 276 1%
Number of LOCed DIFFSs	1 out of 1 100%	1 out of 1 100%	1 out of 1 100%	1 out of 1 100%
Number of GTs		1 out of 8 12%		1 out of 8 12%
Number of LOCed GTs	--	1 out of 1 100%	--	1 out of 1 100%
Number of External GTIPADs		2 out of 16 12%		2 out of 16 12%
Number of LOCed GTIPADs	--	0 out of 2 0%	--	0 out of 2 0%
Number of External GTOPADs		2 out of 16 12%		2 out of 16 12%
Number of LOCed GTOPADs	--	0 out of 2 0%	--	0 out of 2 0%
Number of External IOBs	31 out of 556 5%	16 out of 556 2%	35 out of 556 6%	16 out of 556 2%
Number of LOCed IOBs	31 out of 31 100%	16 out of 16 100%	31 out of 35 88%	16 out of 16 100%
Number of JTAGPPCs	1 out of 1 100%	1 out of 1 100%	1 out of 1 100%	1 out of 1 100%
Number of PPC405s	2 out of 2 100%	2 out of 2 100%	2 out of 2 100%	2 out of 2 100%
Number of RAMB16s	22 out of 136 16%	61 out of 136 44%	18 out of 136 13%	45 out of 136 33%
Number of SLICEs	2021 out of 13696 14%	2527 out of 13696 18%	1216 out of 13696 8%	2504 out of 13696 18%

In my test, both of the two communication systems will work for practical applications and are efficient in transmitting data. The parallel system requires less system resources than the high speed serial system and the design of parallel system is simpler than the serial system. However, the parallel communication bandwidth is limited and the expansion in terms of the number of required pins will occupy much more board's area and increase the crosstalk interference. Compared to the parallel system, the serial system requires more system resources due to the increased design complexity. However, the serial design is very attractive in terms of reducing communication interference. In fact, the differential scheme used by the serial method reduces interference to a large enough degree that significantly higher data rates can be achieved compared to the parallel way.

CHAPTER SEVEN

Conclusion

The Impulse C CoDeveloper can automatically generate VHDL code for a target platform and provides all the basic user interfaces for the hardware connection. It is a very good environment for mixed SW-HW code design and debugging. Impulse C is very good at handling streams and processes. These features made Impulse C the development environment of choice for my communication systems.

As a prerequisite for the design of my communication systems, I studied the features of FIFOs. Asynchronous FIFOs can process data from different clock domains. This is a requirement since various boards will operate with different clock frequencies thereby creating the need for synchronization to preventing the loss of data.

Communication between boards is vital for a reconfigurable computing cluster. In my thesis, I researched both a parallel method and a serial method for board-to-board communications. I used a common Impulse C program to generate the data stream. With the same stream interfaces generated by the Impulse C, I built two different data communication systems. These two communication systems both have their advantages and disadvantages.

The parallel communication system is easier to design with the support of EDK environment. Parallel ports can be connected directly to the hardware pins, and those input and output signals can be monitored directly. The parallel solution also requires less internal FPGA resources. However, the bandwidth and error transfer problem are two main problems for the parallel communication system. The bandwidth gain from a

parallel approach is not unlimited. The increase of the bit-width requires more hardware area for the parallel ports. Furthermore, the parallel method is very susceptible to interference. The interference will increase if the length of the parallel cable increases. The parallel solution's error correction ability is not good. Delay from even one signal may cause a serious communication problem. As the clock frequency is increased it becomes even harder for the parallel system to synchronization the signals. Advances in FPGA technology have resulted in increased clock frequencies for FPGA devices. Therefore it is increasingly difficult for the parallel communication system to properly synchronize signals for reliable data transmission.

The high speed serial communication system is more complicated than the parallel system. It needs both hardware and protocol support. Therefore, it consumes more internal FPGA resources. However, the serial solution has several advantages. It is less susceptible to interference and therefore supports high communication rates. Furthermore, since the serial method only uses a single differential pair for transmission and another one for receiving synchronization of multiple lines is not required. The Aurora protocol proved to be good method for the implementation of a high speed serial communication system. It fully supports the Rocket IO multi-gigabit transceiver modules which are embedded in Virtex-II Pro boards that were used in my project. FIFOs are used as buffer links between the user application (Impulse C module) and Aurora protocol module. The design was implemented in EDK.

Although parallel communication technology still has its advantages, the trend has been away from using parallel method and toward using high-speed serial communication systems. The advantages of high speed serial communication solutions

have led to the serial approach becoming the most popular communication technology. In the area of personal computer, this technology has already replaced all the parallel interfaces and it will become the most competitive communication technology in the embedded area in the near future.

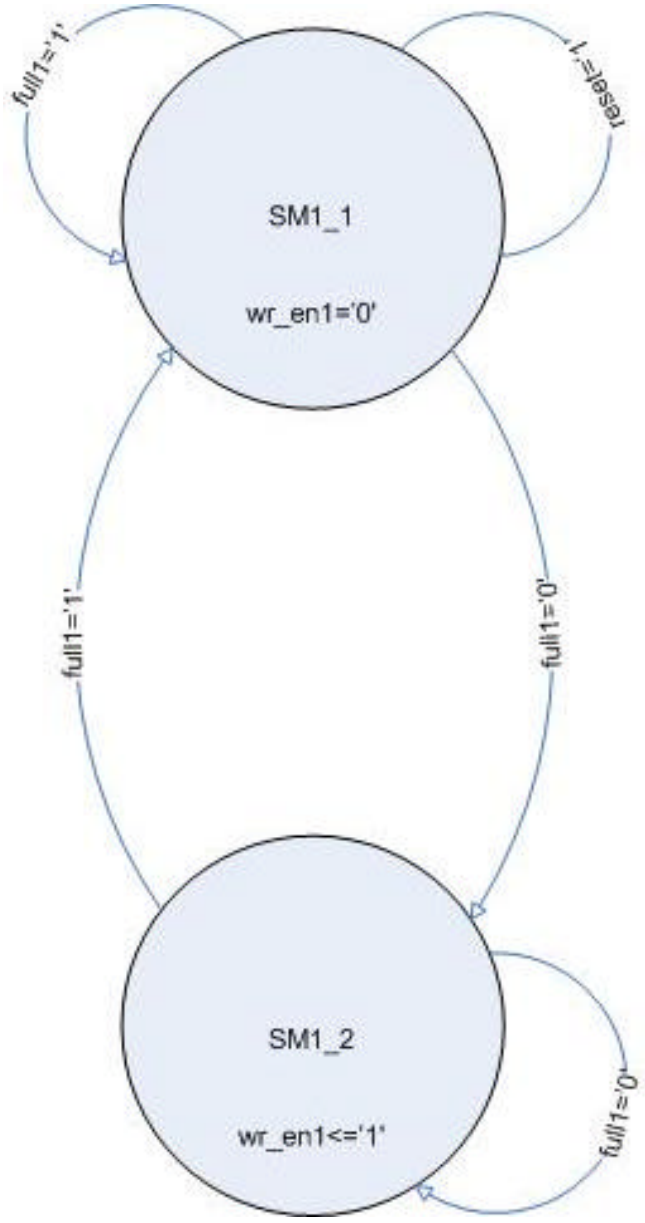
APPENDICES

APPENDIX A

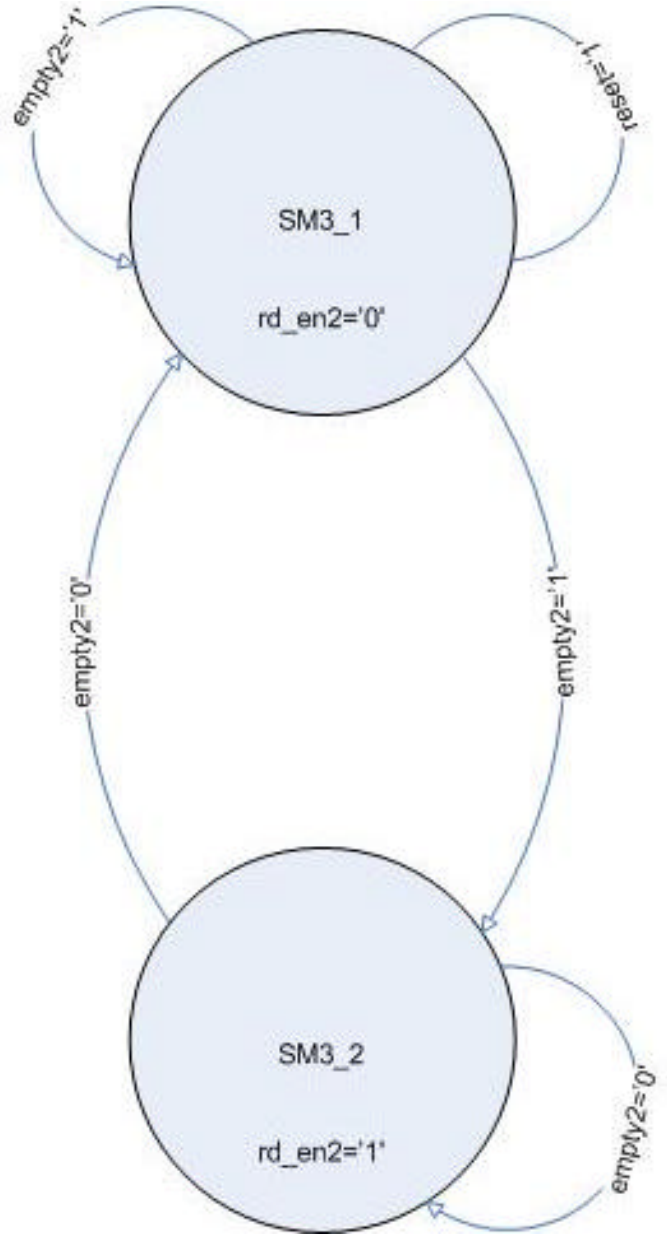
FIFO State Machine Diagrams

State Machine of Board1

State Machine 1

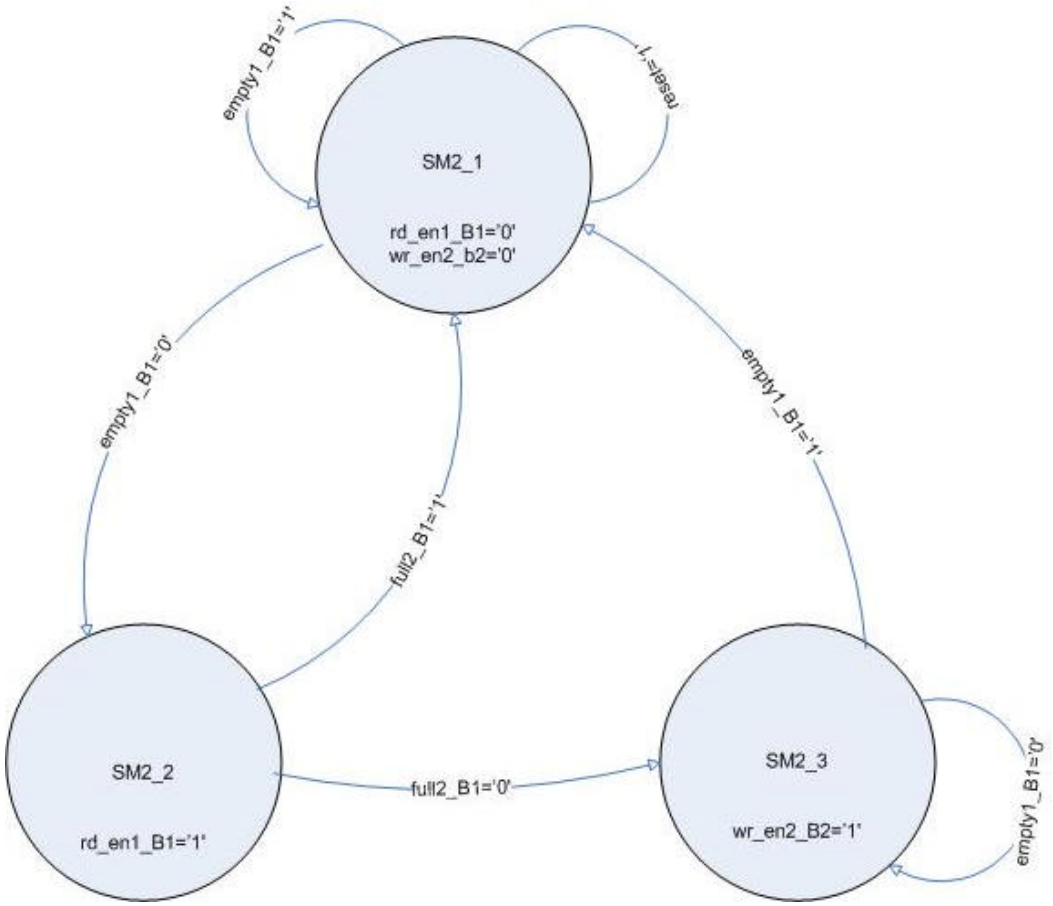


State Machine 3



State Machine of Board2

State Machine 2



There exists some improvement on state machine2 for the future implementation. First of all, I can specify the two outputs in each bubble. Another improvement is that let SM2_1 move to SM2_2 if the full2_B2 is '0', and if empty1_B1 is '0', SM2_2 will move to SM2_3. When FIFO2 is not full, the wr_en signal is asserted to '1', I can write data to FIFO2, and then if FIFO1 is not empty, read data from FIFO1. That design may eventually eliminate the probability of missing data.

FIFO Testing VHDL Source Code for Two Boards

Codes of Board1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity BTB1_V3 is

Port ( clk1 : in  STD_LOGIC;
           clk2 : in  STD_LOGIC;
           data_in : in  STD_LOGIC_VECTOR (15 downto 0);
           wr_en2_B1 : in  STD_LOGIC;
           rd_en1_B1 : in  STD_LOGIC;
           reset : in  STD_LOGIC;
           empty1_B1 : out  STD_LOGIC;
           full2_B1 : out  STD_LOGIC;
           data_out : out  STD_LOGIC_VECTOR (15 downto 0);
           pass : out  STD_LOGIC := '0';
                    LED2 : out  STD_LOGIC;
                    --rst_b2: out std_logic;
           fail : out  STD_LOGIC:= '0');

end BTB1_V3;

architecture Behavioral of BTB1_V3 is

component fifo
      port (
      din: IN std_logic_VECTOR(15 downto 0);
      rd_clk: IN std_logic;
      rd_en: IN std_logic;
      rst: IN std_logic;
      wr_clk: IN std_logic;
      wr_en: IN std_logic;
      dout: OUT std_logic_VECTOR(15 downto 0);
      empty: OUT std_logic;
      full: OUT std_logic);
END component;

-- Synplicity black box declaration
attribute box_type : string;
attribute box_type of fifo : component is "black_box";

type state_type1 is (sm1_1,sm1_2);
signal state1: state_type1;

type state_type3 is (sm3_1,sm3_2);
signal state3: state_type3;
```

```
signal din1: std_logic_VECTOR(15 downto 0);
signal din2: std_logic_VECTOR(15 downto 0);
signal rd_clk1:  std_logic;
signal rd_clk2:  std_logic;
signal rd_en1:  std_logic;
signal rd_en2:  std_logic;
signal wr_clk1:  std_logic;
signal wr_clk2:  std_logic;
signal wr_en1:  std_logic;
signal wr_en2:  std_logic;
signal dout1:  std_logic_VECTOR(15 downto 0);
signal dout2:  std_logic_VECTOR(15 downto 0);
signal empty1: std_logic;
signal empty2: std_logic;
signal full1: std_logic;
signal full2: std_logic;
signal rst1: std_logic;
signal rst2: std_logic;
signal count : std_logic_VECTOR(15 downto 0):=(others => '0');
signal count2 : std_logic_VECTOR(15 downto 0):=(others => '0');
signal rd_en2_dly : std_logic;

begin

F1: fifo
         port map (
                  din => din1,
                  rd_clk => rd_clk1,
                  rd_en => rd_en1,
                  rst => rst1,
                  wr_clk => wr_clk1,
                  wr_en => wr_en1,
                  dout => dout1,
                  empty => empty1,
                  full => full1);

F2: fifo
         port map (
                  din => din2,
                  rd_clk => rd_clk2,
                  rd_en => rd_en2,
                  rst => rst2,
                  wr_clk => wr_clk2,
                  wr_en => wr_en2,
                  dout => dout2,
                  empty => empty2,
                  full => full2);

-- initialize
   wr_clk1 <= clk1;
      rd_clk1 <= clk2;
      rd_clk2 <= clk1;
      wr_clk2 <= clk2;
      rst1 <= reset;
      rst2 <= reset;
      din1 <= count;
```

```
      din2 <= data_in;
      data_out <= dout1;
      rd_en1 <= rd_en1_B1;
   wr_en2 <= wr_en2_B1;
      empty1_B1 <= empty1;
   full2_B1 <= full2;
      --rst_b2 <= reset;
      --test <= dout1(0);
      --data_in <= data_out;
      --din2 <= dout1;


counter:process (clk1,reset)
begin
   if ( reset ='1') then
         count <=(others => '0');
      elsif (not full1 and clk1)='1' and clk1'event then
--    elsif (clk_B1)='1' and clk_B1'event then
      count <= count + 1;
   end if;
end process counter;



SM1_p1: process (clk1,reset)
begin

    if ( reset ='1') then state1 <=sm1_1;
       elsif (clk1='1' and clk1'event) then
             case state1 is

                  when sm1_1 => if full1='1' then
                            state1 <= sm1_1;
                   else
                            state1 <= sm1_2;
                   end if;
                  when sm1_2 => if full1='1' then
                            state1 <= sm1_1;
                   else
                            state1 <= sm1_2;
                   end if;
               end case;
end if;

end process SM1_p1;

SM1_p2: process (state1)

begin

    case state1 is
           when sm1_1 => wr_en1 <= '0';
              when sm1_2 => wr_en1 <= '1';
       end case;
end process SM1_p2;

SM3_p1: process (clk1,reset)
```

```
begin

       if ( reset ='1') then state3 <=sm3_1;
       elsif (clk1='1' and clk1'event) then
             case state3 is
                  when sm3_1 =>  if empty2 = '1' then
                                    state3 <= sm3_1;
                                 else
                                    state3 <= sm3_2;
                                 end if;

                  when sm3_2 => if empty2 = '1' then
                                    state3 <= sm3_1;
                                else
                                    state3 <= sm3_2;
                                end if;
              end case;
       end if;

end process SM3_p1;

SM3_p2: process (state3)
begin
    case state3 is
            when sm3_1 => rd_en2 <= '0';
                  when sm3_2 => rd_en2 <= '1';
   end case;
end process SM3_p2;
--rd_en2 <= not(empty2);


Chk: process (clk1, reset)
begin
if ( reset ='1') then
         count2 <= "1111111111111111";
            pass <= '1';
      fail <= '1';
      elsif clk1='1' and clk1'event then
            if ((rd_en2='1') ) then
               count2 <= count2 + 1;
            end if;
         rd_en2_dly <= rd_en2;
            if rd_en2_dly='1' then
               if (dout2 = count2) then
                     pass <= '0';
                        fail <= '1';
                  else
                  pass <= '1';
            fail <= '0';
                  end if;
            end if;
   end if;

end process Chk;

CHK_B1: process(reset,clk1)
```

```
begin
 if (clk1'event and clk1 = '1' ) then
      if (reset = '1') then
      -- rst_b2 <= '1';
          LED2 <= '0';
      else
          --rst_b2 <= '0';
          LED2 <= '1';
      end if;
 end if;
end Process CHK_B1;

end Behavioral;
```

Codes of Board2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity Board2 is
    Port ( data_in : in  STD_LOGIC_VECTOR (15 downto 0);
           rd_en1_B1 : out  STD_LOGIC;
           empty1_B1 : in  STD_LOGIC;
           data_out : out  STD_LOGIC_VECTOR (15 downto 0);
           clk_B2 : out  STD_LOGIC;
                    clk : in std_logic;
                    reset : in STD_LOGIC;
           wr_en2_B2 : out  STD_LOGIC;
           full2_B1 : in  STD_LOGIC);
end Board2;

architecture Behavioral of Board2 is

type state_type2 is (sm2_1,sm2_2,sm2_3);
signal state2: state_type2;

begin

clk_B2 <= clk;

SM2_p1: process (clk,reset)
begin

    if (reset ='1') then state2 <=sm2_1;
       elsif (clk='1' and clk'event) then
             case state2 is
                      when sm2_1 => if empty1_B1='1' then
                                        state2 <= sm2_1;
```

```
                                    else
                                       state2 <= sm2_2;
                                end if;
                           when sm2_2 => if full2_B1='1' then
                                            state2 <= sm2_1;
                                    else
                                            state2 <= sm2_3;
                                    end if;
                           when sm2_3 => if empty1_B1='1' then
                                            state2 <= sm2_1;
                                     else
                                            state2 <= sm2_3;
                                     end if;
                  end case;
          end if;

end process SM2_p1;

SM2_p2: process (state2)
begin

    case state2 is
           when sm2_1 => rd_en1_B1 <= '0';wr_en2_B2 <= '0';
              when sm2_2 => rd_en1_B1 <= '1';
              when sm2_3 => wr_en2_B2<='1'; data_out <= data_in;
              --when sm1_4 => din1 <= count;
          end case;


end process SM2_p2;

end Behavioral;
```

User Constraint File

Board1

```
#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "clk1"  LOC = "AJ15" | IOSTANDARD = LVCMOS25 ;
NET "clk2"  LOC = "N6" | IOSTANDARD = LVTTL ;
NET "data_in<0>"  LOC = "L5" | IOSTANDARD = LVTTL ;
NET "data_in<1>"  LOC = "M2" | IOSTANDARD = LVTTL ;
NET "data_in<2>"  LOC = "P9" | IOSTANDARD = LVTTL ;
NET "data_in<3>"  LOC = "M4" | IOSTANDARD = LVTTL ;
NET "data_in<4>"  LOC = "N1" | IOSTANDARD = LVTTL ;
NET "data_in<5>"  LOC = "P8" | IOSTANDARD = LVTTL ;
NET "data_in<6>"  LOC = "N4" | IOSTANDARD = LVTTL ;
NET "data_in<7>"  LOC = "P3" | IOSTANDARD = LVTTL ;
NET "data_in<8>"  LOC = "R8" | IOSTANDARD = LVTTL ;
NET "data_in<9>"  LOC = "P5" | IOSTANDARD = LVTTL ;
NET "data_in<10>"  LOC = "R2" | IOSTANDARD = LVTTL ;
NET "data_in<11>"  LOC = "R6" | IOSTANDARD = LVTTL ;
NET "data_in<12>"  LOC = "R4" | IOSTANDARD = LVTTL ;
```

```
NET "data_in<13>"  LOC = "U1" | IOSTANDARD = LVTTL ;
NET "data_in<14>"  LOC = "T5" | IOSTANDARD = LVTTL ;
NET "data_in<15>"  LOC = "T7" | IOSTANDARD = LVTTL ;

NET "data_out<0>"  LOC = "N5" | IOSTANDARD = LVTTL ;
NET "data_out<1>"  LOC = "L4" | IOSTANDARD = LVTTL ;
NET "data_out<2>"  LOC = "N2" | IOSTANDARD = LVTTL ;
NET "data_out<3>"  LOC = "R9" | IOSTANDARD = LVTTL ;
NET "data_out<4>"  LOC = "M3" | IOSTANDARD = LVTTL ;
NET "data_out<5>"  LOC = "P1" | IOSTANDARD = LVTTL ;
NET "data_out<6>"  LOC = "P7" | IOSTANDARD = LVTTL ;
NET "data_out<7>"  LOC = "N3" | IOSTANDARD = LVTTL ;
NET "data_out<8>"  LOC = "P2" | IOSTANDARD = LVTTL ;
NET "data_out<9>"  LOC = "R7" | IOSTANDARD = LVTTL ;
NET "data_out<10>"  LOC = "P4" | IOSTANDARD = LVTTL ;
NET "data_out<11>"  LOC = "T2" | IOSTANDARD = LVTTL ;
NET "data_out<12>"  LOC = "R5" | IOSTANDARD = LVTTL ;
NET "data_out<13>"  LOC = "R3" | IOSTANDARD = LVTTL ;
NET "data_out<14>"  LOC = "V1" | IOSTANDARD = LVTTL ;
NET "data_out<15>"  LOC = "T6" | IOSTANDARD = LVTTL;

NET "empty1_B1"  LOC = "T4" | IOSTANDARD = LVTTL ;
NET "fail"  LOC = "AA5" | IOSTANDARD = LVTTL ;
NET "full2_B1"  LOC = "U3" | IOSTANDARD = LVTTL ;
NET "LED2"  LOC = "AA6" | IOSTANDARD = LVTTL ;
NET "pass"  LOC = "AC4" | IOSTANDARD = LVTTL ;
NET "rd_en1_B1"  LOC = "U2" | IOSTANDARD = LVTTL ;
NET "reset"  LOC = "AG5" | IOSTANDARD = LVTTL ;
NET "wr_en2_B1"  LOC = "T3" | IOSTANDARD = LVTTL ;

#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints
```

Board2

```
#PACE: Start of Constraints generated by PACE

#PACE: Start of PACE I/O Pin Assignments
NET "clk"  LOC = "AJ15" | IOSTANDARD = LVCMOS25 ;
NET "clk_B2"  LOC = "N6" | IOSTANDARD = LVTTL ;
NET "data_in<0>"  LOC = "N5" | IOSTANDARD = LVTTL ;
NET "data_in<1>"  LOC = "L4" | IOSTANDARD = LVTTL ;
NET "data_in<2>"  LOC = "N2" | IOSTANDARD = LVTTL ;
NET "data_in<3>"  LOC = "R9" | IOSTANDARD = LVTTL ;
NET "data_in<4>"  LOC = "M3" | IOSTANDARD = LVTTL ;
NET "data_in<5>"  LOC = "P1" | IOSTANDARD = LVTTL ;
NET "data_in<6>"  LOC = "P7" | IOSTANDARD = LVTTL ;
NET "data_in<7>"  LOC = "N3" | IOSTANDARD = LVTTL ;
NET "data_in<8>"  LOC = "P2" | IOSTANDARD = LVTTL ;
NET "data_in<9>"  LOC = "R7" | IOSTANDARD = LVTTL ;
NET "data_in<10>"  LOC = "P4" | IOSTANDARD = LVTTL ;
NET "data_in<11>"  LOC = "T2" | IOSTANDARD = LVTTL ;
NET "data_in<12>"  LOC = "R5" | IOSTANDARD = LVTTL ;
NET "data_in<13>"  LOC = "R3" | IOSTANDARD = LVTTL ;
```

```
NET "data_in<14>"  LOC = "V1" | IOSTANDARD = LVTTL ;
NET "data_in<15>"  LOC = "T6" | IOSTANDARD = LVTTL ;

NET "data_out<0>"  LOC = "L5" | IOSTANDARD = LVTTL ;
NET "data_out<1>"  LOC = "M2" | IOSTANDARD = LVTTL ;
NET "data_out<2>"  LOC = "P9" | IOSTANDARD = LVTTL ;
NET "data_out<3>"  LOC = "M4" | IOSTANDARD = LVTTL ;
NET "data_out<4>"  LOC = "N1" | IOSTANDARD = LVTTL ;
NET "data_out<5>"  LOC = "P8" | IOSTANDARD = LVTTL ;
NET "data_out<6>"  LOC = "N4" | IOSTANDARD = LVTTL ;
NET "data_out<7>"  LOC = "P3" | IOSTANDARD = LVTTL ;
NET "data_out<8>"  LOC = "R8" | IOSTANDARD = LVTTL ;
NET "data_out<9>"  LOC = "P5" | IOSTANDARD = LVTTL ;
NET "data_out<10>"  LOC = "R2" | IOSTANDARD = LVTTL ;
NET "data_out<11>"  LOC = "R6" | IOSTANDARD = LVTTL ;
NET "data_out<12>"  LOC = "R4" | IOSTANDARD = LVTTL ;
NET "data_out<13>"  LOC = "U1" | IOSTANDARD = LVTTL ;
NET "data_out<14>"  LOC = "T5" | IOSTANDARD = LVTTL ;
NET "data_out<15>"  LOC = "T7" | IOSTANDARD = LVTTL ;

NET "empty1_B1"  LOC = "T4" | IOSTANDARD = LVTTL ;
NET "full2_B1"  LOC = "U3" | IOSTANDARD = LVTTL ;
NET "rd_en1_B1"  LOC = "U2" | IOSTANDARD = LVTTL ;
NET "reset"  LOC = "AG5" | IOSTANDARD = LVTTL ;
NET "wr_en2_B2"  LOC = "T3" | IOSTANDARD = LVTTL ;

#PACE: Start of PACE Area Constraints

#PACE: Start of PACE Prohibit Constraints

#PACE: End of Constraints generated by PACE
```

APPENDIX B

Impulse C Source Code for Two Boards

Codes of Board1

```
//  Main function.
//

#include "co.h"

#include <stdio.h>

extern co_architecture co_initialize(int arg);

int main()
{
   int arg = 0;
   char c;
   co_architecture arch = co_initialize(arg);
   co_execute(arch);

   //IF_SIM(
   printf("Board1's job is done ^ ^...\n");
   c = getc(stdin);
  // )
   return 0;
}

// Process to be executed on the target CPU.
//

#include "co.h"
#include "cosim_log.h"
#include <stdio.h>

#ifdef IMPULSE_C_TARGET
#define printf xil_printf
#endif

void numgen(co_stream count)
{
```

```
 co_int16 i;

   IF_SIM(cosim_logwindow log = cosim_logwindow_create("numgen");)

   co_stream_open(count, O_WRONLY, INT_TYPE(16));

   printf("CPU is counting...\n\r");
   for ( i = 0; i<1000; i++ ) {
      co_stream_write(count, &i, sizeof(co_int16));
      printf("Number Generated: %d \n\r", i);
   }

   co_stream_close(count);
   printf("Count done\n\r");

}

// Hardware processes and configuration code.
//

#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"

#define MONITOR

#ifdef MONITOR
#include "cosim_log.h"
#endif

// Software process
extern void numgen(co_stream count);

void Hw_Process(co_stream data_in, co_stream data_out)
{
#ifdef MONITOR
   IF_SIM(cosim_logwindow log;)
#endif
   co_int16 value;

#ifdef MONITOR
   IF_SIM(log = cosim_logwindow_create("Hw_Process");)
#endif
```

```
    co_stream_open(data_in, O_RDONLY, INT_TYPE(16));
    co_stream_open(data_out,O_WRONLY, INT_TYPE(16));

    while (co_stream_read(data_in, &value, sizeof(co_int16))==co_err_none) {
       //value=value+10;//data could be processed here
       co_stream_write(data_out, &value, sizeof(co_int16));
    }
    co_stream_close(data_in);
    co_stream_close(data_out);
}

void config(void * arg)
{
    co_stream data_in, d_out;
    co_process hw1, Numgen;
    co_port port_out;

 #define BUFSIZE 1024
#ifdef MONITOR
    IF_SIM(cosim_logwindow_init();)
#endif

    data_in = co_stream_create("data_in",INT_TYPE(16),BUFSIZE);
    d_out = co_stream_create("d_out", INT_TYPE(16), BUFSIZE);
    port_out = co_port_create("output_stream", co_output, d_out);
    Numgen = co_process_create("Number_Generater", (co_function)numgen,
                               1,
                               data_in);
    hw1 = co_process_create("Hardware_Process_1", (co_function)Hw_Process,
                               2,
                               data_in,
                               d_out);
    co_process_config(hw1, co_loc, "PE0");
}

co_architecture co_initialize(int arg)
{
    return co_architecture_create("btb_fifo_board1", "generic", config, (void *) arg);
}
```

Codes of Board2

```
// Main function.
//
#include "co.h"
```

```
#include <stdio.h>

extern co_architecture co_initialize(int arg);

int main()
{
   int arg = 0;
   char c;
   co_architecture arch = co_initialize(arg);
   co_execute(arch);

   //IF_SIM(
   printf("Communication is finished. Press any key to exit the program...\n");
   c = getc(stdin);
  // )
   return 0;
}

// Process to be executed on the target CPU.
//

#include "co.h"
#include "cosim_log.h"
#include <stdio.h>

#ifdef IMPULSE_C_TARGET
#define printf xil_printf
#endif

void test(co_stream tst)
{
   co_int16 testdata;

  IF_SIM(cosim_logwindow log = cosim_logwindow_create("test");)

   co_stream_open(tst, O_RDONLY, INT_TYPE(16));
   printf("Test started...\n\r");
   while (co_stream_read(tst, &testdata, sizeof(co_int16))==co_err_none) {
         printf("Number received: %d from the input stream\n\r", testdata);
   }
   co_stream_close(tst);
   printf("Test done\n\r");
}
// Hardware processes and configuration code.
//
```

```
#ifdef WIN32
#include <windows.h>
#endif
#include <stdio.h>
#include "co.h"

#define MONITOR

#ifdef MONITOR
#include "cosim_log.h"
#endif

// Software process
extern void test(co_stream tst);

void Hw_Process2(co_stream data_in, co_stream data_out)
{
#ifdef MONITOR
   IF_SIM(cosim_logwindow log;)
#endif
   co_int16 value;

#ifdef MONITOR
   IF_SIM(log = cosim_lo gwindow_create("Hw_Process");)
#endif

   co_stream_open(data_in, O_RDONLY, INT_TYPE(16));
   co_stream_open(data_out,O_WRONLY, INT_TYPE(16));

   while (co_stream_read(data_in, &value, sizeof(co_int16))==co_err_none) {
         //data could be processed here
      co_stream_write(data_out, &value, sizeof(co_int16));
   }
   co_stream_close(data_in);
   co_stream_close(data_out);
}

void config(void * arg)
{
   co_stream data_out, d_in;
   co_process  hw2, Test;
   co_port port_in;

   #define BUFSIZE 1024

#ifdef MONITOR
```

```
    IF_SIM(cosim_logwindow_init();)
#endif

    data_out = co_stream_create("data_out", INT_TYPE(16), BUFSIZE);
    d_in = co_stream_create("d_in", INT_TYPE(16), BUFSIZE);
    port_in = co_port_create("input_stream", co_input, d_in);
    hw2 = co_process_create("Hardware_Process_2", (co_function)Hw_Process2,
                                2,
                                d_in,
                                data_out);
    Test = co_process_create("Test", (co_function)test,
                                 1,
                                 data_out);

    co_process_config(hw2, co_loc, "PE0");
}

co_architecture co_initialize(int arg)
{
    return co_architecture_create("btb_fifo_board2", "generic", config, (void *) arg);
}
```

APPENDIX C

Source Codes of Designed IP Cores

Aurora Module Added Codes

```
------------------------------------------------------------------------------
-- Filename:          aurora_mgt.vhd
-- Version:           1.01.a
-- Description:       Top level design, instantiates IPIF and user
logic.
-- Date:              Sat Apr 04 16:16:16 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------
    -- ADD USER PORTS BELOW THIS LINE ------------------
       DIN_TX                  : in  std_logic_vector(0 to 15);
    DOUT_RX              : out std_logic_vector(0 to 15);
       WE_TX                : in  std_logic;
      -- RE_TX                 : in  std_logic;
       RE_RX                : in  std_logic;
    FULL_TX              : out std_logic;
       VALID_RX             : out std_logic;
       EMPTY_RX             : out std_logic;
    TOP_BREF_CLK : in std_logic;
    USER_CLK : in std_logic;
    HARD_ERROR : out std_logic;
    SOFT_ERROR : out std_logic;
    LANE_UP : out std_logic;
    CHANNEL_UP : out std_logic;
       RXP : in std_logic;
    RXN : in std_logic;
    TXP : out std_logic;
    TXN : out std_logic;

    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------


      -- MAP USER PORTS BELOW THIS LINE ------------------
       DIN_TX        => DIN_TX,
    DOUT_RX    => DOUT_RX,
       WE_TX     => WE_TX,
       --RE_TX     => RE_TX,
       RE_RX     => RE_RX,
    FULL_TX   => FULL_TX,
       VALID_RX  => VALID_RX,
       EMPTY_RX  => EMPTY_RX,
            TOP_BREF_CLK => TOP_BREF_CLK,
```

```
      USER_CLK => USER_CLK,
      HARD_ERROR => HARD_ERROR,
      SOFT_ERROR => SOFT_ERROR,
      LANE_UP => LANE_UP,
      CHANNEL_UP => CHANNEL_UP,
      RXP => RXP,
      RXN => RXN,
      TXP => TXP,
      TXN => TXN,

      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------

------------------------------------------------------------------------
-------
-- Filename:          user_logic.vhd
-- Version:           1.01.a
-- Description:       User logic.
-- Date:              Sat Apr 04 16:16:16 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------
-------
    -- ADD USER PORTS BELOW THIS LINE ------------------
       DIN_TX                   : in  std_logic_vector(0 to 15);
    DOUT_RX              : out std_logic_vector(0 to 15);
       WE_TX                 : in  std_logic;
       --RE_TX                 : in  std_logic;
       RE_RX                 : in  std_logic;
    FULL_TX              : out std_logic;
       VALID_RX              : out std_logic;
       EMPTY_RX              : out std_logic;
       TOP_BREF_CLK : in std_logic;
    USER_CLK : in std_logic;
    HARD_ERROR : out std_logic;
    SOFT_ERROR : out std_logic;
    LANE_UP : out std_logic;
    CHANNEL_UP : out std_logic;
    RXP : in std_logic;
    RXN : in std_logic;
    TXP : out std_logic;
    TXN : out std_logic;

    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------



  --USER signal declarations added here, as needed for user logic
   -- External Register Declarations --
signal HARD_ERROR_Buffer : std_logic;
signal SOFT_ERROR_Buffer : std_logic;
signal LANE_UP_Buffer : std_logic;
signal CHANNEL_UP_Buffer : std_logic;
signal TXP_Buffer : std_logic;
signal TXN_Buffer : std_logic;
-- Wire Declarations --
```

```
-- Stream TX Interface
signal tx_d_i : std_logic_vector(0 to 15);
signal tx_src_rdy_n_i : std_logic;
signal tx_dst_rdy_n_i : std_logic;
-- Stream RX Interface
signal rx_d_i : std_logic_vector(0 to 15);
signal rx_src_rdy_n_i : std_logic;
-- Error Detection Interface
signal hard_error_i : std_logic;
signal soft_error_i : std_logic;
-- Status
signal channel_up_i : std_logic;
signal lane_up_i : std_logic;
-- Clock Compensation Control Interface
signal warn_cc_i : std_logic;
signal do_cc_i : std_logic;
--TX & RX FIFO signals
  signal tx_fifo_empty : std_logic;
  signal rx_fifo_empty : std_logic;
  signal tx_fifo_we    : std_logic := '0';
  signal rx_fifo_we    : std_logic := '0';
  signal tx_fifo_re    : std_logic := '0';
  signal rx_fifo_re    : std_logic := '0';
  signal tx_fifo_valid : std_logic;
  signal rx_fifo_valid : std_logic;
  signal tx_fifo_almost_full : std_logic;
  signal rx_fifo_almost_full : std_logic;
  signal tx_fifo_dout  : std_logic_vector(15 downto 0);
  signal rx_fifo_dout  : std_logic_vector(15 downto 0);
  signal tx_fifo_din   : std_logic_vector(15 downto 0);
  signal rx_fifo_din   : std_logic_vector(15 downto 0);

-- Component Declarations --
component aurora_201
generic (
EXTEND_WATCHDOGS : boolean := FALSE
);
port (
-- LocalLink TX Interface
TX_D : in std_logic_vector(0 to 15);
TX_SRC_RDY_N : in std_logic;
TX_DST_RDY_N : out std_logic;
-- LocalLink RX Interface
RX_D : out std_logic_vector(0 to 15);
RX_SRC_RDY_N : out std_logic;
-- MGT Serial I/O
RXP : in std_logic;
RXN : in std_logic;
TXP : out std_logic;
TXN : out std_logic;
-- MGT Reference Clock Interface
TOP_BREF_CLK : in std_logic;
-- Error Detection Interface
HARD_ERROR : out std_logic;
SOFT_ERROR : out std_logic;
-- Status
CHANNEL_UP : out std_logic;
```

```
LANE_UP : out std_logic;
-- Clock Compensation Control Interface
WARN_CC : in std_logic;
DO_CC : in std_logic;
-- System Interface
DCM_NOT_LOCKED : in std_logic;
USER_CLK : in std_logic;
RESET : in std_logic;
POWER_DOWN : in std_logic;
LOOPBACK : in std_logic_vector(1 downto 0)
);
end component;
component STANDARD_CC_MODULE
port (
-- Clock Compensation Control Interface
WARN_CC : out std_logic;
DO_CC : out std_logic;
-- System Interface
DCM_NOT_LOCKED : in std_logic;
USER_CLK : in std_logic;
CHANNEL_UP : in std_logic
);
end component;
 -- FIFO component from CORE Generator
  component fifo_generator_v3_2
    port (
      din         : IN std_logic_VECTOR(15 downto 0);
      rd_clk      : IN std_logic;
      rd_en       : IN std_logic;
      rst         : IN std_logic;
      wr_clk      : IN std_logic;
      wr_en       : IN std_logic;
      almost_full : OUT std_logic;
      dout        : OUT std_logic_VECTOR(15 downto 0);
      empty       : OUT std_logic;
      full        : OUT std_logic;
      valid       : OUT std_logic
    );
  end component;


begin

  --USER logic implementation added here
  HARD_ERROR <= HARD_ERROR_Buffer;
SOFT_ERROR <= SOFT_ERROR_Buffer;
LANE_UP <= LANE_UP_Buffer;
CHANNEL_UP <= CHANNEL_UP_Buffer;
TXP <= TXP_Buffer;
TXN <= TXN_Buffer;
-- Register User I/O --
-- Register User Outputs from core.
process (USER_CLK)
begin
  if (USER_CLK 'event and USER_CLK = '1') then
      HARD_ERROR_Buffer <= hard_error_i;
      SOFT_ERROR_Buffer <= soft_error_i;
```

```
      LANE_UP_Buffer <= lane_up_i;
      CHANNEL_UP_Buffer <= channel_up_i;
  end if;
end process;

-- Aurora core instantiation
aurora_module_i : aurora_201
port map (
-- LocalLink TX Interface
TX_D => tx_d_i,
TX_SRC_RDY_N => tx_src_rdy_n_i,
TX_DST_RDY_N => tx_dst_rdy_n_i,
-- LocalLink RX Interface
RX_D => rx_d_i,
RX_SRC_RDY_N => rx_src_rdy_n_i,
-- MGT Serial I/O
RXP => RXP,
RXN => RXN,
TXP => TXP_Buffer,
TXN => TXN_Buffer,
-- MGT Reference Clock Interface
TOP_BREF_CLK => TOP_BREF_CLK,
-- Error Detection Interface
HARD_ERROR => hard_error_i,
SOFT_ERROR => soft_error_i,
-- Status
CHANNEL_UP => channel_up_i,
LANE_UP => lane_up_i,
-- Clock Compensation Control Interface
WARN_CC => warn_cc_i,
DO_CC => do_cc_i,
-- System Interface
DCM_NOT_LOCKED => '0',
USER_CLK => USER_CLK,
RESET => Bus2IP_Reset,
POWER_DOWN => '0',
LOOPBACK => "00"
);
standard_cc_module_i : STANDARD_CC_MODULE
port map (
-- Clock Compensation Control Interface
WARN_CC => warn_cc_i,
DO_CC => do_cc_i,
-- System Interface
DCM_NOT_LOCKED => '0',
USER_CLK => USER_CLK,
CHANNEL_UP => channel_up_i
);

-- FIFOs:
tx_fifo_i : fifo_generator_v3_2
    port map (
      din => tx_fifo_din,
      rd_clk => USER_CLK,
      rd_en => tx_fifo_re,
      rst => Bus2IP_Reset,
      wr_clk => Bus2IP_Clk,
```

```
      wr_en => tx_fifo_we,
      almost_full => tx_fifo_almost_full,
      dout => tx_fifo_dout,
      empty => tx_fifo_empty,
      full => open,
      valid => tx_fifo_valid);

  rx_fifo_i : fifo_generator_v3_2
    port map (
      din => rx_fifo_din,
      rd_clk => Bus2IP_Clk,
      rd_en => rx_fifo_re,
      rst => Bus2IP_Reset,
      wr_clk => USER_CLK,
      wr_en => rx_fifo_we,
      almost_full => rx_fifo_almost_full,
      dout => rx_fifo_dout,
      empty => rx_fifo_empty,
      full => open,
      valid => rx_fifo_valid);

            -- initialize
       tx_fifo_din <= DIN_TX;
    DOUT_RX <= rx_fifo_dout;
       tx_fifo_we <= WE_TX;
       --tx_fifo_re <= RE_TX ;
       rx_fifo_re <= RE_RX ;
    FULL_TX <= tx_fifo_almost_full ;
       VALID_RX <= rx_fifo_valid;
       EMPTY_RX <= rx_fifo_empty ;

            -- Connection between TX FIFO and Aurora TX
  tx_src_rdy_n_i <= not tx_fifo_valid;
  tx_d_i <= tx_fifo_dout;
  tx_fifo_re <= (not tx_fifo_empty) and (not tx_dst_rdy_n_i);

  -- Connection between RX FIFO and Aurora RX
  rx_fifo_we <= (not rx_src_rdy_n_i) and (not rx_fifo_almost_full);
  rx_fifo_din <= rx_d_i;
```

Digital Clock Manager Module Added Codes

```
------------------------------------------------------------------------
-------
-- Filename:          mgt_dcm.vhd
-- Version:           1.00.a
-- Description:       Top level design, instantiates IPIF and user
logic.
-- Date:              Thu Mar 19 21:07:03 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------
-------

-- ADD USER PORTS BELOW THIS LINE ------------------
```

```
       TOP_BREF_CLK_P : in std_logic;
    TOP_BREF_CLK_N : in std_logic;
    TOP_BREF_CLK : out std_logic;
    USER_CLK : out std_logic;
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------

-- MAP USER PORTS BELOW THIS LINE ------------------
         TOP_BREF_CLK_P => TOP_BREF_CLK_P,
      TOP_BREF_CLK_N => TOP_BREF_CLK_N,
      TOP_BREF_CLK => TOP_BREF_CLK,
      USER_CLK => USER_CLK,
      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------

------------------------------------------------------------------------------
-------
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:              Thu Mar 19 21:07:03 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------
-------

-- ADD USER PORTS BELOW THIS LINE ------------------
       TOP_BREF_CLK_P : in std_logic;
    TOP_BREF_CLK_N : in std_logic;
    TOP_BREF_CLK : out std_logic;
    USER_CLK : out std_logic;
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------

--USER signal declarations added here, as needed for user logic
  component IBUFGDS_LVDS_25
    port(
      O : out std_ulogic;
      I : in std_ulogic;
      IB : in std_ulogic
    );
  end component;

  component BUFG
    port(
      O : out std_ulogic;
      I : in std_ulogic
    );
  end component;

  signal top_bref_clk_i : std_logic;
  signal user_clk_i : std_logic;

begin

  --USER logic implementation added here
```

```
  -- Differential Clock Buffer for top BREF_CLK
  diff_clk_buff_top_i : IBUFGDS_LVDS_25
  port map(
    I => TOP_BREF_CLK_P,
    IB => TOP_BREF_CLK_N,
    O => top_bref_clk_i
  );
  -- BUFG used to drive USER_CLK on global clock net
  user_clock_bufg_i : BUFG
  port map(
    I => top_bref_clk_i,
    O => user_clk_i
  );

  TOP_BREF_CLK <= top_bref_clk_i;
  USER_CLK <= user_clk_i;
```

N-Gate Added Codes

```
------------------------------------------------------------------------
-------
-- Filename:          n_gate.vhd
-- Version:           1.00.a
-- Description:       Top level design, instantiates IPIF and user
logic.
-- Date:              Mon Mar 30 22:06:34 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------
-------

-- ADD USER PORTS BELOW THIS LINE ------------------
       input                          : in std_logic;
       output                         : out std_logic;
    --USER ports added here
-- ADD USER PORTS ABOVE THIS LINE ------------------
-- MAP USER PORTS BELOW THIS LINE ------------------
            input => input,
            output => output,
-- USER ports mapped here
-- MAP USER PORTS ABOVE THIS LINE ------------------
------------------------------------------------------------------------
-------
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:              Mon Mar 30 22:06:34 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------
-------

    -- ADD USER PORTS BELOW THIS LINE ------------------
       input                          : in std_logic;
       output                         : out std_logic;
```

```
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------
  --USER signal declarations added here, as needed for user logic
  signal input1  : std_logic;
  signal output1 : std_logic;
begin
  --USER logic implementation added here
  output1 <= not input1;
  input1 <= input;
  output <= output1;
```

And-Gate Added Code

```
------------------------------------------------------------------------------
-- Filename:          and_gate.vhd
-- Version:           1.00.a
-- Description:       Top level design, instantiates IPIF and user
logic.
-- Date:              Fri Apr 10 17:21:48 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------
    -- ADD USER PORTS BELOW THIS LINE ------------------
       INPUT1  : in  std_logic;
       INPUT2  : in  std_logic;
       OUTPUT  : out std_logic;
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------
      -- MAP USER PORTS BELOW THIS LINE -------------------
            INPUT1 => INPUT1,
            INPUT2 => INPUT2,
            OUTPUT => OUTPUT,
      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------

------------------------------------------------------------------------------
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:              Fri Apr 10 17:21:48 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------
-- ADD USER PORTS BELOW THIS LINE ------------------
       INPUT1  : in  std_logic;
       INPUT2  : in  std_logic;
       OUTPUT  : out std_logic;
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------
  --USER signal declarations added here, as needed for user logic
begin
  --USER logic implementation added here
  OUTPUT <= INPUT1 and INPUT2;
```

```
------------------------------------------------------------------------------
-- Filename:          rct.vhd
-- Version:           1.00.a
-- Description:       Top level design, instantiates IPIF and user
logic.
-- Date:              Sun Jun 07 18:17:02 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------

   -- ADD USER PORTS BELOW THIS LINE ------------------
       inputd : in std_logic_vector(0 to 15);
       inputc : in std_logic;
       inputr : in std_logic;
       pass : out std_logic_vector(0 to 31);
       fail : out std_logic_vector(0 to 31);
       CT : out std_logic_vector(0 to 15);
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE ------------------

      -- MAP USER PORTS BELOW THIS LINE ------------------
            inputd => inputd,
            inputc => inputc,
            inputr => inputr,
            pass => pass,
            fail => fail,
            CT => CT,
      --USER ports mapped here
      -- MAP USER PORTS ABOVE THIS LINE ------------------

------------------------------------------------------------------------------
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:              Sun Jun 07 18:17:02 2009 (by Create and Import
Peripheral Wizard)
-- VHDL Standard:     VHDL'93
------------------------------------------------------------------------------

 -- ADD USER PORTS BELOW THIS LINE ------------------
       inputd : in std_logic_vector(0 to 15);
       inputc : in std_logic;
       inputr : in std_logic;
       pass : out std_logic_vector(0 to 31);
       fail : out std_logic_vector(0 to 31);
       CT : out std_logic_vector(0 to 15);
    --USER ports added here
    -- ADD USER PORTS ABOVE THIS LINE --

  --USER signal declarations added here, as needed for user logic
  signal isd : std_logic_vector(0 to 15);
  signal cntr : std_logic_vector(0 to 15):= (others => '0');
  signal pas : std_logic_vector(0 to 31):= (others => '0');
  signal fal : std_logic_vector(0 to 31):= (others => '0');
```

```
begin
isd <= inputd;
pass <= pas;
fail <= fal;
CT <= cntr;
  process(inputc, inputr)
  begin
    if (inputr = '1') then
        pas <= '0';
        fal <= '0';
    elsif(inputc)= '1' and inputc'event then
         if (cntr = isd) then
                  pas <= pas + 1;
                  cntr <= cntr + 1;
         else
                  fal <= fal + 1;
                  cntr <= cntr + 1;
         end if;
    end if;
  end process;
```

APPENDIX D

User Constraint Files of EDK Projects

UCF of Board1 for Parallel Communication

```
##############################################################################
#####
## This system.ucf file is generated by Base System Builder based on
the
## settings in the selected Xilinx Board Definition file. Please add
other
## user constraints to this file based on customer design
specifications.
##############################################################################
#####

Net sys_clk_pin LOC=AJ15;
Net sys_clk_pin IOSTANDARD = LVCMOS25;
Net sys_rst_pin LOC=AG5;
Net sys_rst_pin IOSTANDARD = LVTTL;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;
Net sys_rst_pin TIG;
NET "C405RSTCORERESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTCHIPRESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTSYSRESETREQ" TPTHRU = "RST_GRP";
TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS  TIG;

## IO Devices constraints

Net "output_stream_data_pin<0>" LOC = "N5" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<1>" LOC = "L4" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<2>" LOC = "N2" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<3>" LOC = "R9" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<4>" LOC = "M3" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<5>" LOC = "P1" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<6>" LOC = "P7" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<7>" LOC = "N6" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<8>" LOC = "L5" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
```

```
Net "output_stream_data_pin<9>" LOC = "M2" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<10>" LOC = "P9" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<11>" LOC = "M4" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<12>" LOC = "N1" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<13>" LOC = "P8" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<14>" LOC = "N4" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_data_pin<15>" LOC = "P3" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24;
Net "output_stream_en_pin" LOC = "R7" | IOSTANDARD = LVTTL ;
Net "output_stream_rdy_pin" LOC = "R5" | IOSTANDARD = LVTTL ;
Net "output_stream_eos_pin" LOC = "P4" | IOSTANDARD = LVTTL ;
Net "USER_CLK_pin" LOC = "U2" | IOSTANDARD = LVTTL ;

## Add some grounds
Net "ExternalPort_0" LOC= "N3" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_1" LOC= "R8" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_2" LOC= "P5" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_3" LOC= "R2" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_4" LOC= "R4" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;

#### Module onewire_0 constraints

Net fpga_0_onewire_0_ONEWIRE_DQ LOC=J3;
Net fpga_0_onewire_0_ONEWIRE_DQ IOSTANDARD = LVTTL;
Net fpga_0_onewire_0_ONEWIRE_DQ SLEW = SLOW;
Net fpga_0_onewire_0_ONEWIRE_DQ DRIVE = 8;

#### Module RS232_Uart_1 constraints

Net fpga_0_RS232_Uart_1_RX_pin LOC=AJ8;
Net fpga_0_RS232_Uart_1_RX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin LOC=AE7;
Net fpga_0_RS232_Uart_1_TX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin SLEW = SLOW;
Net fpga_0_RS232_Uart_1_TX_pin DRIVE = 12;

Net fpga_0_net_gnd_pin LOC=G12;
Net fpga_0_net_gnd_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_pin SLEW = SLOW;
Net fpga_0_net_gnd_pin DRIVE = 6;
Net fpga_0_net_gnd_1_pin LOC=D15;
Net fpga_0_net_gnd_1_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_1_pin SLEW = SLOW;
Net fpga_0_net_gnd_1_pin DRIVE = 6;
Net fpga_0_net_gnd_2_pin LOC=E15;
Net fpga_0_net_gnd_2_pin IOSTANDARD = LVTTL;
```

```
Net fpga_0_net_gnd_2_pin SLEW = SLOW;
Net fpga_0_net_gnd_2_pin DRIVE = 6;
Net fpga_0_net_gnd_3_pin LOC=G10;
Net fpga_0_net_gnd_3_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_3_pin SLEW = SLOW;
Net fpga_0_net_gnd_3_pin DRIVE = 6;
Net fpga_0_net_gnd_4_pin LOC=E10;
Net fpga_0_net_gnd_4_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_4_pin SLEW = SLOW;
Net fpga_0_net_gnd_4_pin DRIVE = 6;
Net fpga_0_net_gnd_5_pin LOC=G8;
Net fpga_0_net_gnd_5_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_5_pin SLEW = SLOW;
Net fpga_0_net_gnd_5_pin DRIVE = 6;
Net fpga_0_net_gnd_6_pin LOC=H9;
Net fpga_0_net_gnd_6_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_6_pin SLEW = SLOW;
Net fpga_0_net_gnd_6_pin DRIVE = 6;
```

UCF of Board2 for Parallel Communication

```
##############################################################################
#####
## This system.ucf file is generated by Base System Builder based on
the
## settings in the selected Xilinx Board Definition file. Please add
other
## user constraints to this file based on customer design
specifications.
##############################################################################
#####

Net sys_clk_pin LOC=AJ15;
Net sys_clk_pin IOSTANDARD = LVCMOS25;
Net sys_rst_pin LOC=AG5;
Net sys_rst_pin IOSTANDARD = LVTTL;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;
Net sys_rst_pin TIG;
NET "C405RSTCORERESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTCHIPRESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTSYSRESETREQ" TPTHRU = "RST_GRP";
TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS  TIG;

## IO Devices constraints

Net "input_stream_data_pin<0>" LOC = "N5" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24;
Net "input_stream_data_pin<1>" LOC = "L4" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<2>" LOC = "N2" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<3>" LOC = "R9" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
```

```
Net "input_stream_data_pin<4>" LOC = "M3" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<5>" LOC = "P1" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<6>" LOC = "P7" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<7>" LOC = "N6" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<8>" LOC = "L5" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<9>" LOC = "M2" | IOSTANDARD = LVTTL | SLEW =
FAST | DRIVE = 24 ;
Net "input_stream_data_pin<10>" LOC = "P9" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24 ;
Net "input_stream_data_pin<11>" LOC = "M4" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24 ;
Net "input_stream_data_pin<12>" LOC = "N1" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24 ;
Net "input_stream_data_pin<13>" LOC = "P8" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24 ;
Net "input_stream_data_pin<14>" LOC = "N4" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24 ;
Net "input_stream_data_pin<15>" LOC = "P3" | IOSTANDARD = LVTTL | SLEW
= FAST | DRIVE = 24 ;
Net "input_stream_en_pin" LOC = "R5" | IOSTANDARD = LVTTL ;
Net "input_stream_rdy_pin" LOC = "R7" | IOSTANDARD = LVTTL ;
Net "input_stream_eos_pin" LOC = "P4" | IOSTANDARD = LVTTL ;
Net "input_stream_clk_pin" LOC = "U2" | IOSTANDARD = LVTTL ;

## Add some grounds
Net "ExternalPort_0" LOC= "N3" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_1" LOC= "R8" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_2" LOC= "P5" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_3" LOC= "R2" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;
Net "ExternalPort_4" LOC= "R4" | IOSTANDARD = LVTTL | slew = slow |
drive = 6;

#### Module onewire_0 constraints

Net fpga_0_onewire_0_ONEWIRE_DQ LOC=J3;
Net fpga_0_onewire_0_ONEWIRE_DQ IOSTANDARD = LVTTL;
Net fpga_0_onewire_0_ONEWIRE_DQ SLEW = SLOW;
Net fpga_0_onewire_0_ONEWIRE_DQ DRIVE = 8;

#### Module RS232_Uart_1 constraints

Net fpga_0_RS232_Uart_1_RX_pin LOC=AJ8;
Net fpga_0_RS232_Uart_1_RX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin LOC=AE7;
Net fpga_0_RS232_Uart_1_TX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin SLEW = SLOW;
Net fpga_0_RS232_Uart_1_TX_pin DRIVE = 12;
```

```
Net fpga_0_net_gnd_pin LOC=G12;
Net fpga_0_net_gnd_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_pin SLEW = SLOW;
Net fpga_0_net_gnd_pin DRIVE = 6;
Net fpga_0_net_gnd_1_pin LOC=D15;
Net fpga_0_net_gnd_1_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_1_pin SLEW = SLOW;
Net fpga_0_net_gnd_1_pin DRIVE = 6;
Net fpga_0_net_gnd_2_pin LOC=E15;
Net fpga_0_net_gnd_2_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_2_pin SLEW = SLOW;
Net fpga_0_net_gnd_2_pin DRIVE = 6;
Net fpga_0_net_gnd_3_pin LOC=G10;
Net fpga_0_net_gnd_3_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_3_pin SLEW = SLOW;
Net fpga_0_net_gnd_3_pin DRIVE = 6;
Net fpga_0_net_gnd_4_pin LOC=E10;
Net fpga_0_net_gnd_4_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_4_pin SLEW = SLOW;
Net fpga_0_net_gnd_4_pin DRIVE = 6;
Net fpga_0_net_gnd_5_pin LOC=G8;
Net fpga_0_net_gnd_5_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_5_pin SLEW = SLOW;
Net fpga_0_net_gnd_5_pin DRIVE = 6;
Net fpga_0_net_gnd_6_pin LOC=H9;
Net fpga_0_net_gnd_6_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_6_pin SLEW = SLOW;
Net fpga_0_net_gnd_6_pin DRIVE = 6;
```

UCF of Board1 for MGT Serial Communication

```
##############################################################################
#####
## This system.ucf file is generated by Base System Builder based on
the
## settings in the selected Xilinx Board Definition file. Please add
other
## user constraints to this file based on customer design
specifications.
##############################################################################
#####

Net sys_clk_pin LOC=AJ15;
Net sys_clk_pin IOSTANDARD = LVCMOS25;
Net sys_rst_pin LOC=AG5;
Net sys_rst_pin IOSTANDARD = LVTTL;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;
Net sys_rst_pin TIG;
NET "C405RSTCORERESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTCHIPRESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTSYSRESETREQ" TPTHRU = "RST_GRP";
TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS  TIG;
```

```
## IO Devices constraints
#### Module onewire_0 constraints

Net fpga_0_onewire_0_ONEWIRE_DQ LOC=J3;
Net fpga_0_onewire_0_ONEWIRE_DQ IOSTANDARD = LVTTL;
Net fpga_0_onewire_0_ONEWIRE_DQ SLEW = SLOW;
Net fpga_0_onewire_0_ONEWIRE_DQ DRIVE = 8;

#### Module RS232_Uart_1 constraints

Net fpga_0_RS232_Uart_1_RX_pin LOC=AJ8;
Net fpga_0_RS232_Uart_1_RX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin LOC=AE7;
Net fpga_0_RS232_Uart_1_TX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin SLEW = SLOW;
Net fpga_0_RS232_Uart_1_TX_pin DRIVE = 12;

Net fpga_0_net_gnd_pin LOC=G12;
Net fpga_0_net_gnd_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_pin SLEW = SLOW;
Net fpga_0_net_gnd_pin DRIVE = 6;
Net fpga_0_net_gnd_1_pin LOC=D15;
Net fpga_0_net_gnd_1_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_1_pin SLEW = SLOW;
Net fpga_0_net_gnd_1_pin DRIVE = 6;
Net fpga_0_net_gnd_2_pin LOC=E15;
Net fpga_0_net_gnd_2_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_2_pin SLEW = SLOW;
Net fpga_0_net_gnd_2_pin DRIVE = 6;
Net fpga_0_net_gnd_3_pin LOC=G10;
Net fpga_0_net_gnd_3_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_3_pin SLEW = SLOW;
Net fpga_0_net_gnd_3_pin DRIVE = 6;
Net fpga_0_net_gnd_4_pin LOC=E10;
Net fpga_0_net_gnd_4_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_4_pin SLEW = SLOW;
Net fpga_0_net_gnd_4_pin DRIVE = 6;
Net fpga_0_net_gnd_5_pin LOC=G8;
Net fpga_0_net_gnd_5_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_5_pin SLEW = SLOW;
Net fpga_0_net_gnd_5_pin DRIVE = 6;
Net fpga_0_net_gnd_6_pin LOC=H9;
Net fpga_0_net_gnd_6_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_6_pin SLEW = SLOW;
Net fpga_0_net_gnd_6_pin DRIVE = 6;

##### Timing Contraints for the MGT Recovered clock ######

NET
aurora_mgt_0/aurora_mgt_0/USER_LOGIC_I/aurora_module_i/lane_0_mgt_i/RXR
ECCLK PERIOD=75 MHz;

##### Timing Contraints for the MGT reference clock ######

##### Timing Contraints for the MGT reference clock ######
NET USER_CLK PERIOD = 75 MHz;
NET TOP_BREF_CLK PERIOD = 75 MHz;
```

```
###### Connect the external MGT clock inputs #######
NET TOP_BREF_CLK_P_pin LOC=F16;
NET TOP_BREF_CLK_P_pin IOSTANDARD = LVDS_25;
NET TOP_BREF_CLK_N_pin LOC=G16;
NET TOP_BREF_CLK_N_pin IOSTANDARD = LVDS_25;
###### MGT Locations #######

# X0Y1 (SATA 0 HOST) aurora_mgt_0
INST
aurora_mgt_0/aurora_mgt_0/USER_LOGIC_I/aurora_module_i/lane_0_mgt_i
LOC=GT_X0Y1;

###### Connect LEDs to the Aurora Status outputs ########

# LED 0
NET CHANNEL_UP_pin LOC = AC4;
Net CHANNEL_UP_pin IOSTANDARD = LVTTL;
Net CHANNEL_UP_pin SLEW = SLOW;
Net CHANNEL_UP_pin DRIVE = 12;
# LED 1
NET HARD_ERROR_pin LOC = AC3;
Net HARD_ERROR_pin IOSTANDARD = LVTTL;
Net HARD_ERROR_pin SLEW = SLOW;
Net HARD_ERROR_pin DRIVE = 12;
# LED 2
NET SOFT_ERROR_pin LOC = AA6;
Net SOFT_ERROR_pin IOSTANDARD = LVTTL;
Net SOFT_ERROR_pin SLEW = SLOW;
Net SOFT_ERROR_pin DRIVE = 12;
# LED 3
NET LANE_UP_pin LOC = AA5;
Net LANE_UP_pin IOSTANDARD = LVTTL;
Net LANE_UP_pin SLEW = SLOW;
Net LANE_UP_pin DRIVE = 12;
```

UCF of Board2 for MGT Serial Communication

```
##############################################################################
#####
## This system.ucf file is generated by Base System Builder based on
the
## settings in the selected Xilinx Board Definition file. Please add
other
## user constraints to this file based on customer design
specifications.
##############################################################################
#####

Net sys_clk_pin LOC=AJ15;
Net sys_clk_pin IOSTANDARD = LVCMOS25;
Net sys_rst_pin LOC=AG5;
Net sys_rst_pin IOSTANDARD = LVTTL;
## System level constraints
Net sys_clk_pin TNM_NET = sys_clk_pin;
```

```
TIMESPEC TS_sys_clk_pin = PERIOD sys_clk_pin 10000 ps;
Net sys_rst_pin TIG;
NET "C405RSTCORERESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTCHIPRESETREQ" TPTHRU = "RST_GRP";
NET "C405RSTSYSRESETREQ" TPTHRU = "RST_GRP";
TIMESPEC "TS_RST1" = FROM CPUS THRU RST_GRP TO FFS  TIG;

## IO Devices constraints

#### Module onewire_0 constraints

Net fpga_0_onewire_0_ONEWIRE_DQ LOC=J3;
Net fpga_0_onewire_0_ONEWIRE_DQ IOSTANDARD = LVTTL;
Net fpga_0_onewire_0_ONEWIRE_DQ SLEW = SLOW;
Net fpga_0_onewire_0_ONEWIRE_DQ DRIVE = 8;

#### Module RS232_Uart_1 constraints

Net fpga_0_RS232_Uart_1_RX_pin LOC=AJ8;
Net fpga_0_RS232_Uart_1_RX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin LOC=AE7;
Net fpga_0_RS232_Uart_1_TX_pin IOSTANDARD = LVCMOS25;
Net fpga_0_RS232_Uart_1_TX_pin SLEW = SLOW;
Net fpga_0_RS232_Uart_1_TX_pin DRIVE = 12;

Net fpga_0_net_gnd_pin LOC=G12;
Net fpga_0_net_gnd_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_pin SLEW = SLOW;
Net fpga_0_net_gnd_pin DRIVE = 6;
Net fpga_0_net_gnd_1_pin LOC=D15;
Net fpga_0_net_gnd_1_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_1_pin SLEW = SLOW;
Net fpga_0_net_gnd_1_pin DRIVE = 6;
Net fpga_0_net_gnd_2_pin LOC=E15;
Net fpga_0_net_gnd_2_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_2_pin SLEW = SLOW;
Net fpga_0_net_gnd_2_pin DRIVE = 6;
Net fpga_0_net_gnd_3_pin LOC=G10;
Net fpga_0_net_gnd_3_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_3_pin SLEW = SLOW;
Net fpga_0_net_gnd_3_pin DRIVE = 6;
Net fpga_0_net_gnd_4_pin LOC=E10;
Net fpga_0_net_gnd_4_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_4_pin SLEW = SLOW;
Net fpga_0_net_gnd_4_pin DRIVE = 6;
Net fpga_0_net_gnd_5_pin LOC=G8;
Net fpga_0_net_gnd_5_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_5_pin SLEW = SLOW;
Net fpga_0_net_gnd_5_pin DRIVE = 6;
Net fpga_0_net_gnd_6_pin LOC=H9;
Net fpga_0_net_gnd_6_pin IOSTANDARD = LVTTL;
Net fpga_0_net_gnd_6_pin SLEW = SLOW;
Net fpga_0_net_gnd_6_pin DRIVE = 6;

##### Timing Contraints for the MGT Recovered clock ######
```

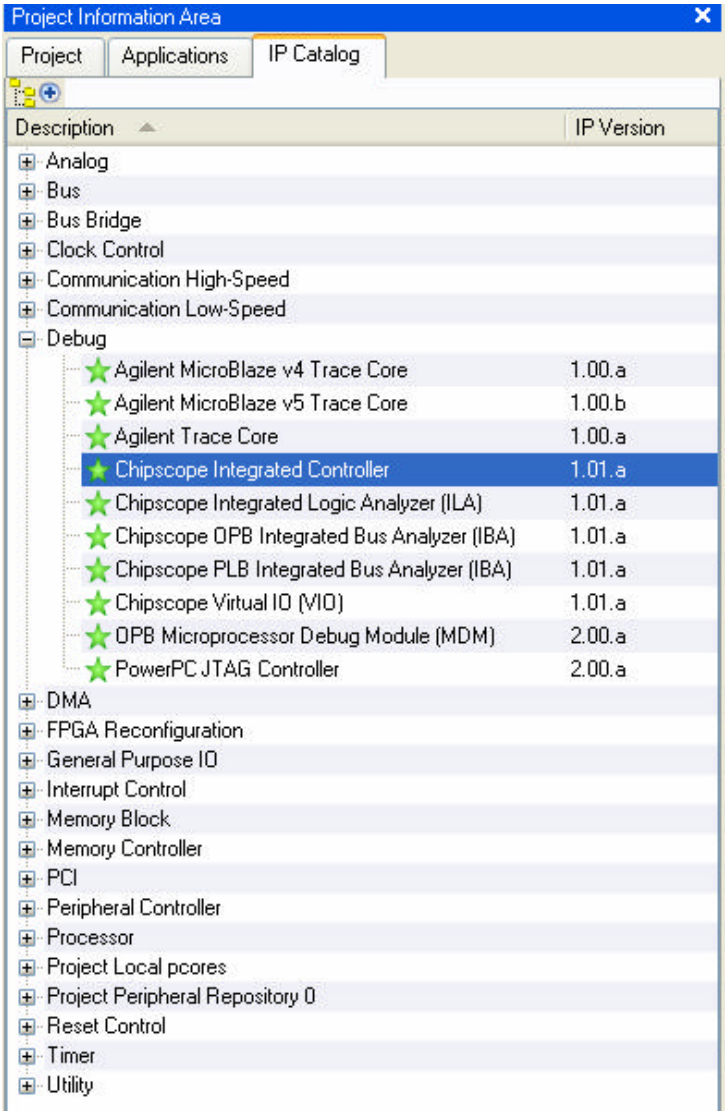
```
NET
aurora_mgt_0/aurora_mgt_0/USER_LOGIC_I/aurora_module_i/lane_0_mgt_i/RXR
ECCLK PERIOD=75 MHz;
##### Timing Contraints for the MGT reference clock ######


##### Timing Contraints for the MGT reference clock ######
NET USER_CLK PERIOD = 75 MHz;
NET TOP_BREF_CLK PERIOD = 75 MHz;
###### Connect the external MGT clock inputs #######
NET TOP_BREF_CLK_P_pin LOC=F16;
NET TOP_BREF_CLK_P_pin IOSTANDARD = LVDS_25;
NET TOP_BREF_CLK_N_pin LOC=G16;
NET TOP_BREF_CLK_N_pin IOSTANDARD = LVDS_25;
###### MGT Locations #######

# X1Y1 (SATA 1 TARGET) aurora_mgt_0
INST
aurora_mgt_0/aurora_mgt_0/USER_LOGIC_I/aurora_module_i/lane_0_mgt_i
LOC=GT_X1Y1;

###### Connect LEDs to the Aurora Status outputs ########

# LED 0
NET CHANNEL_UP_pin LOC = AC4;
Net CHANNEL_UP_pin IOSTANDARD = LVTTL;
Net CHANNEL_UP_pin SLEW = SLOW;
Net CHANNEL_UP_pin DRIVE = 12;
# LED 1
NET HARD_ERROR_pin LOC = AC3;
Net HARD_ERROR_pin IOSTANDARD = LVTTL;
Net HARD_ERROR_pin SLEW = SLOW;
Net HARD_ERROR_pin DRIVE = 12;
# LED 2
NET SOFT_ERROR_pin LOC = AA6;
Net SOFT_ERROR_pin IOSTANDARD = LVTTL;
Net SOFT_ERROR_pin SLEW = SLOW;
Net SOFT_ERROR_pin DRIVE = 12;
# LED 3
NET LANE_UP_pin LOC = AA5;
Net LANE_UP_pin IOSTANDARD = LVTTL;
Net LANE_UP_pin SLEW = SLOW;
Net LANE_UP_pin DRIVE = 12;
```

APPENDIX E

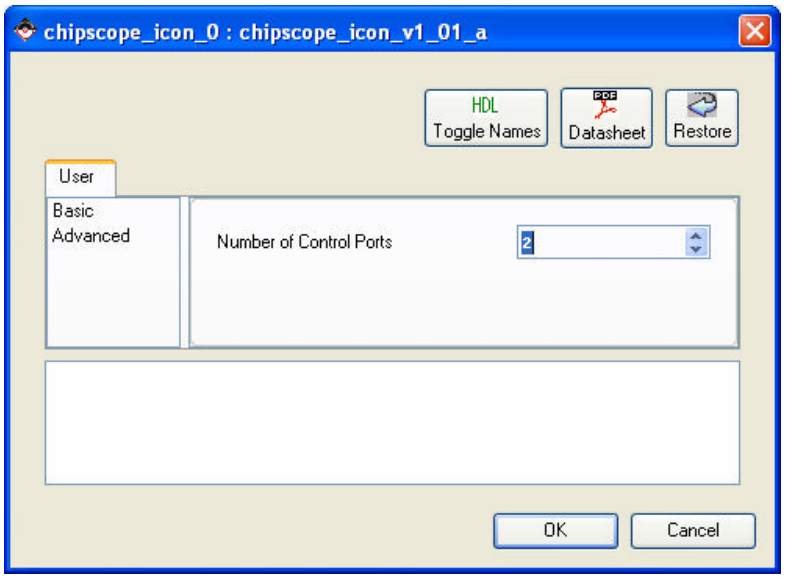
Chipscope User Guide

This appendix shows how to insert Chipscope IP cores to the EDK project and use it to monitor the signals.

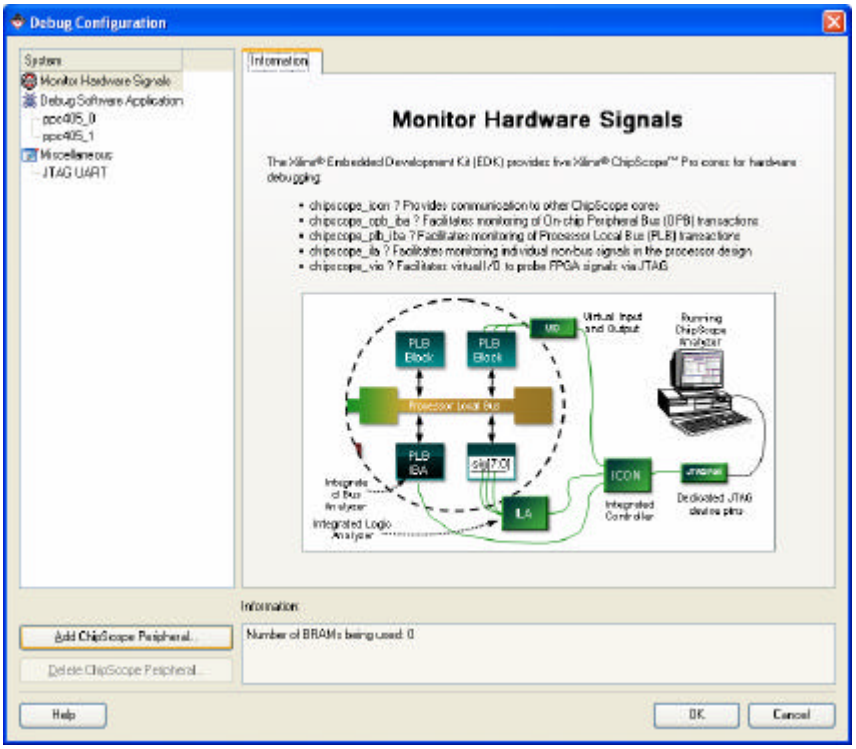
First, open the EDK project, and then choose the Chipscope Integrated Controllers from the IP Catalog (double click to add it to the project):



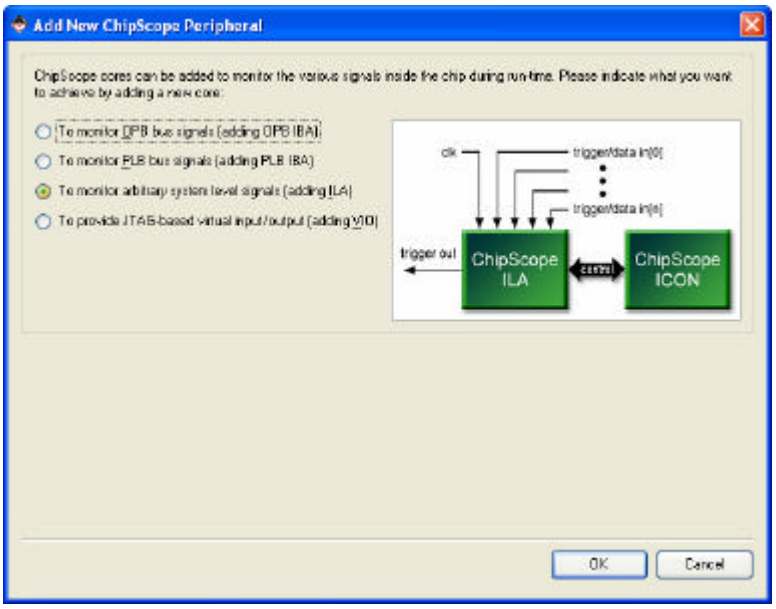
Then configure its parameters and set the number of control ports to 2:



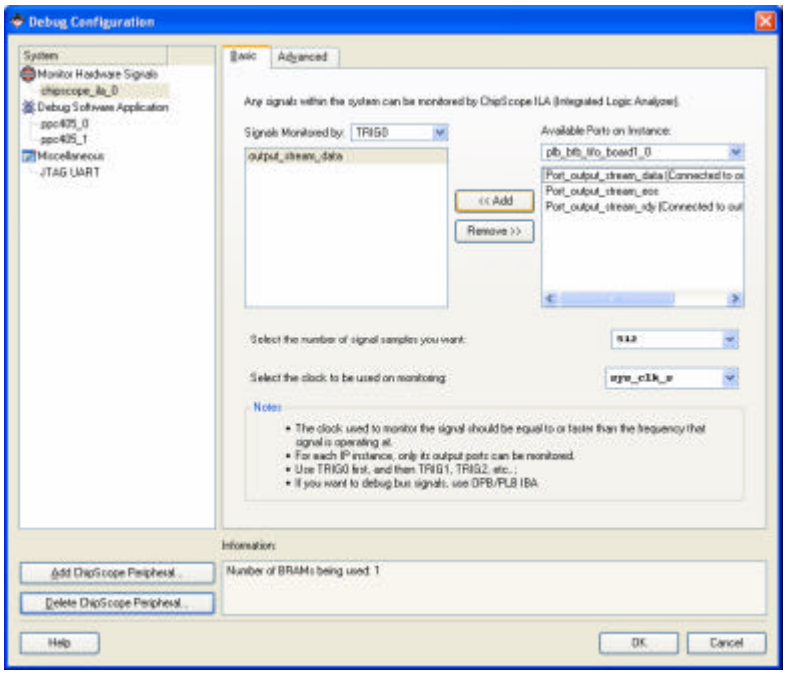
After that step, the Chipscope Integrated controller has been added to the EDK project. Then, go to the debug menu and select the Debug Configuration. C lick on that button and the Debug Configuration window appears.



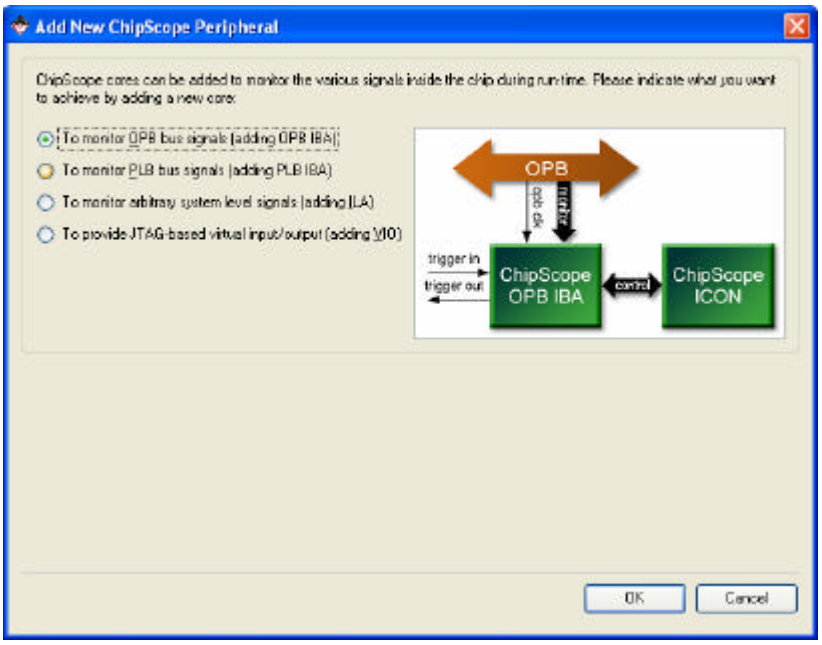
From that window, click on the button Add Chipscope Peripheral. The following window will appear:



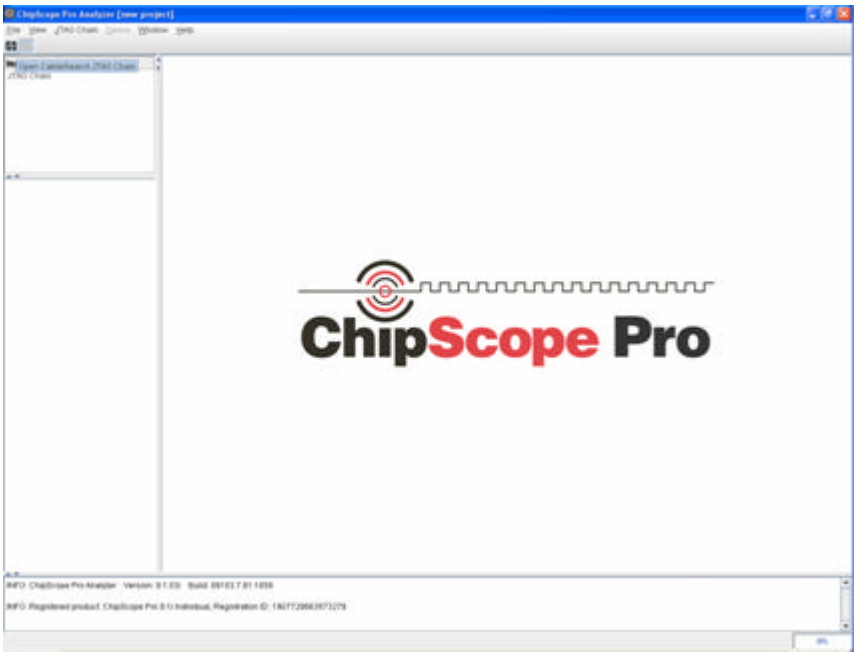
Click on To monitor arbitrary system level signals (adding ILA) and click OK to confirm the selection. From the following window, I can choose the signals which I want to monitor by the Chipscope:



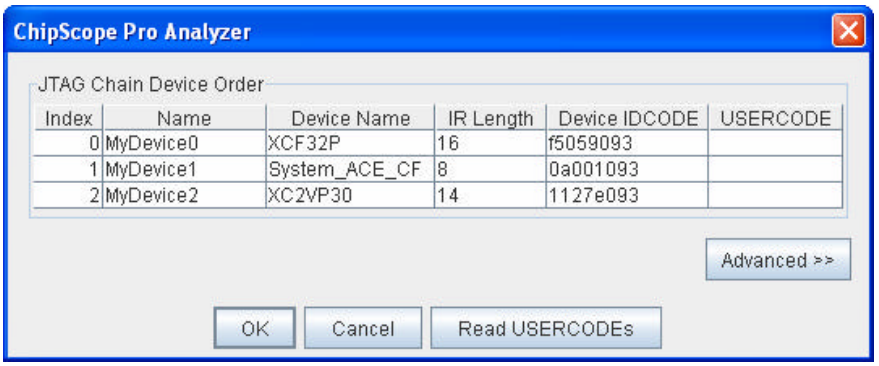
Then, add another monitor to monitor the OPB/PLB bus signals (this depends on the signal you want to monitor):



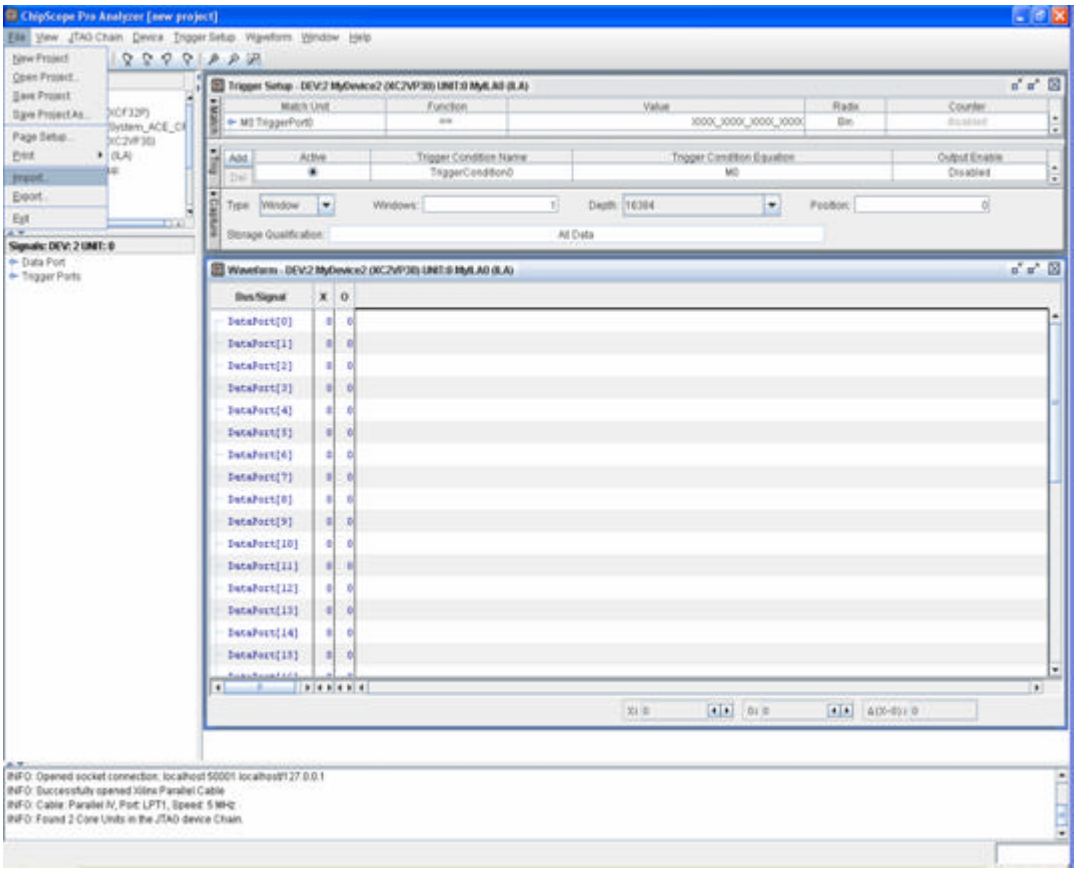
After adding all the Chipscope IP cores to the EDK project, I can compile the project and download the project to the target boards. Then open the Chipscope Pro Analyzer and click on the Open Cable/Search JTAG Chain button:



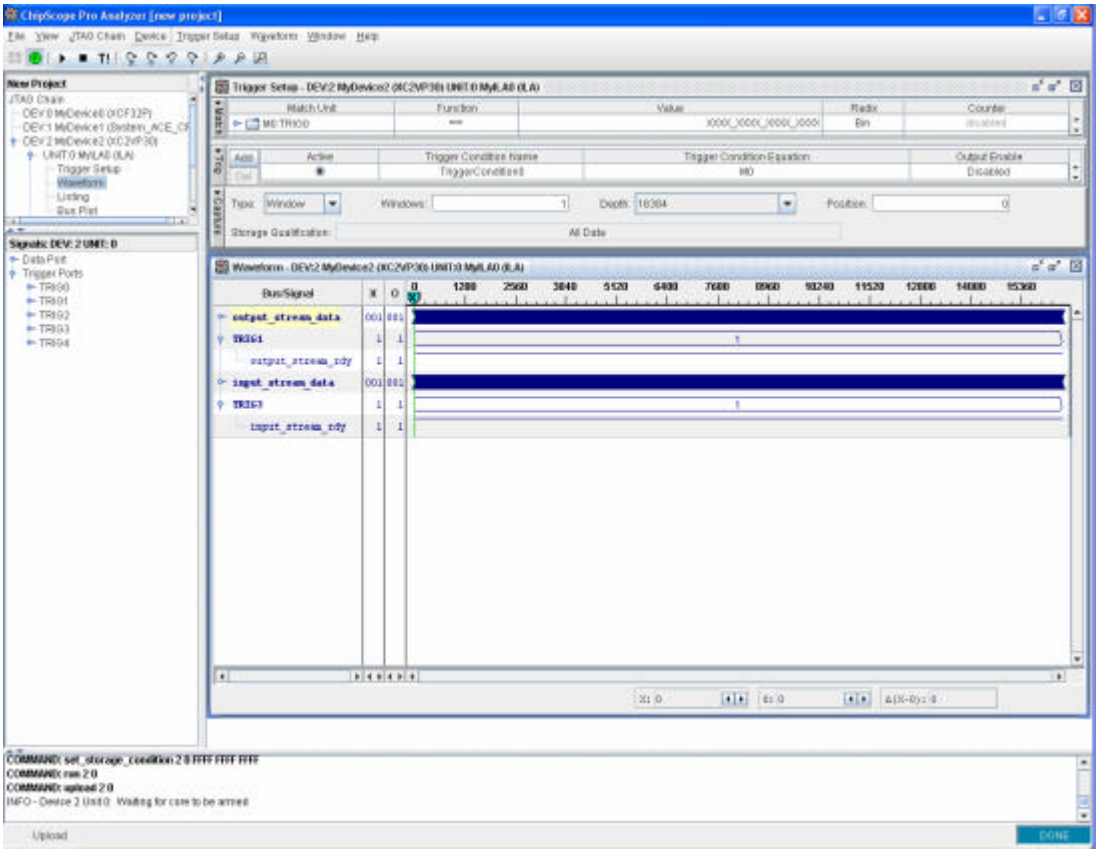
If all things are configured correctly, the Chipscope Analyzer will find the system devices and connect them automatically. If the devices listed are correct, then click OK:



The next step is to import the compiled .cdc file which includes the settings of the EDK project to the Chipscope:



After selecting the correct file and importing it to the Chipscope, the selected signals will appear in the Chip scope Analyzer waveform window. Then I can monitor these real time signals from this window:



BIBLIOGRAPHY

[1] Zbyn?ek Vymazal, "Serial communication of FPGA chips using RocketIO", Diploma Thesis, Dept. Eng., Czech Technical University in Prague, May 2006.

[2] Sherief Reda, "Introduction to reconfigurable computing", Reconfigurable Computing lectures, Division of Engineering, Brown University, Spring 2007.

[3] Gerald Estrin, "Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer", *IEEE Ann. Hist. Comput.* 24, 4 (Oct. 2002), 3–9.

[4] Russ Duren, "Low Cost Reconfigurable Computing Cluster Brings Millions of Reconfigurable FPGA Gates to Students", Dept. Eng., Baylor University, December 2008.

[5] Raik Nagel and Thomas Rauber, "*RCM* - A Multi-layered Reconfigurable Cluster Middleware", *Parallel, Distributed, and Network-Based Processing, 2006. PDP 2006. 14th Euromicro International Conference on*, 15-17 Feb. 2006.

[6] Sass, R., Kritikos, W.V., Schmidt, A.G., Beeravolu, S. Beeraka, P.,"Reconfigurable Computing Cluster (RCC) Project: Investigating the Feasibility of FPGA-Based Petascale Computing", ield-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium, Page(s):127 – 140 on 23-25 April 2007.

[7] Schmidt, A.G.; Kritikos, W.V.; Datta, S.; Sass, R., "Reconfigurable Computing Cluster Project: Phase I Brief", Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on Page(s):300 – 301, 14-15 April 2008.

[8] W. V. Kritikos., "Feasibility of Serial ATA Cables for the Physical Link in High Performance Computing Clusters", Master's thesis, University of Kansas, May 2007.

[9] Shouqian Yu, Lili Yi, Weihai Chen, Zhaojin Wen, "Implementation of a Multi-channel UART Controller Based on FIFO Technique and FPGA", Industrial Electronics and Applications, 2007. ICIEA 2007. Page(s):2633 – 2638, 2nd IEEE Conference on 23-25 May 2007.

[10] Xilinx, INC, "EDK Concepts, Tools and Techniques"; EDK_CTT, September 18, 2008.

[11] Xilinx INC, "Xilinx University Program Virtex-II Pro Development System Hardware Reference Manual"; UG069 (v1.0), March 8, 2005.

[12] Xilinx INC, "LogiCORE FIFO Generator v3.2 User Guide"; UG175, September 21, 2006

[13] IDE cable pinout, http://pinouts.ru/DiskCables/IDE_pinout.shtml

[14] Carmen Li Shen, "Evaluating Impulse C and Multiple Parallelism Partitions for a Low-Cost Reconfigurable Computing System", M.S. Thesis, Dept. Eng., Baylor University, Waco, TX, December 2008.

[15] Impulse Accelerated Technologies, "Impulse C-to-FPGA Workshop", www.rssi2008.org/proceedings/tutorial/Impulse.pdf

[16] Impulse Accelerated Technologies, "CoDeveloper User Guide (V3.2)", 2008.

[17] David Pellerin, Scott Thibault, "Practical FPGA Programming in C", Prentice Hall PTR, April 22, 2005.

[18] Impulse Accelerated Technologies, "CoDeveloper Xilinx PowerPC Platform Support Package (V3.3)", 2008.

[19] Xilinx INC, "Embedded System Tools Reference Manual"; UG111, 2009.

[20] Serial ATA Working Group, "Serial ATA: High Speed Serialized AT Attachment Revision 1.0"

[21] Xilinx INC, "RocketIO Transceiver User Guide", UG024 (V3.0), February 22, 2007

[22] Claude Shannon, "A Mathematical Theory of Communication", *Bell System Technical Journal*, 1948.

[23] Xilinx INC, "Aurora Protocol Specification", SP002 (V2.0), September 19, 2007.

[24] Xilinx INC, "LogiCORE Aurora User Guide", UG061 (V2.8), October 10, 2007.

[25] Xilinx INC, "Aurora Product Specification", DS128, October 10, 2007.

[26] Virtex-II Pro Resource, http://virtex2pro.blogspot.com/2008/03/create-auroratransceiver.html.