ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF TECHNOLOGY

ELECTRICAL AND COMPUTER ENGINEERING

DEPARTMENT

USB Interface for Communication between FPGA and Personal Computer

By

Abiy Tadesse

A thesis submitted to the school of Graduate studies of Addis Ababa University in partial fulfillment of the requirements for the degree of Masters of Science in Electrical Engineering

March 2009

Addis Ababa, Ethiopia

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL AND COMPUTER

ENGINEERING

USB INTERFACE FOR COMMUNICATION BETWEEN FPGA AND PERSONAL COMPUTER

By

Abiy Tadesse

Advisor

Prof. Dr. Gerald Higelin

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF TECHNOLOGY

ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

USB INTERFACE FOR COMMUNICATION BETWEEN FPGA AND PERSONAL COPMPUTER

By
Abiy Tadesse

FACULTY OF TECHNOLOGY

APPROVAL BY BOARD OF EXAMINERS

_____	_____
Chairman Dept. of Graduate Committee	Signature
Prof. Dr. Gerald Higelin	_____
Advisor	Signature
_____	_____
Internal Examiner	Signature
_____	_____
External Examiner	Signature

Acknowledgments

I wish to thank my advisor, Prof. Dr. Gerald Higelin, for all of his thoughtful guidance, invaluable assistance and encouragement throughout the thesis work. He also provided valuable materials (FPGA and accessories) which are not easily obtainable locally. With out his support, sound advice, good teaching and correcting my thesis draft work patiently, this research would never have been completed.

I would like to acknowledge the members of Technology Faculty of AAU, specially the Department of Electrical and Computer Engineering, for their cooperative in providing necessary supports.

I also wish to thank my fellow graduate student, Abraham Berhanu, who supported in providing invaluable sources.

Finally my thanks go to address everyone who helped directly or indirectly for the accomplishment of the thesis work.

Indescribable thanks are to my God for his countless helps throughout my life!

Table of Contents

List of Figures

Figures Page

List of Tables

Tables Page

List of Acronyms

ACK	Acknowledgment
ASIC	Application Specific Integrated Circuit
CLB	Configuration Logic Block
CPLD	Complex Programmable Logic Device
CRC	Cyclic Redundancy Check
DAS	Data Acquisition System
DCM	Digital Clock Manager
DMA	Direct Memory Access
EDA	Electronic Design Automation
EOP	End of Packet
FPGA	Field Programmable Gate Array
FS	Full Speed
HDL	Hardware Description Language
HID	Human Interface Device
HS	High speed
I/O	Input Output
IC	Integrated Circuit
ID	Identification
IOB	Input Output Block
IRQ	Interrupt Request
ISE	Integrated Software Environment
JTAG	Joint Test Action Group
LED	Light Emitting Diode
LS	Low speed
LUT	Look up Table
NAK	Negative Acknowledgment
NRZI	Non-Return to Zero, Inverted
PACE	Pinout and Area Constraints Editor
PC	Personal Computer
PID	Packet Identifier
RTL	Register Transfer Level
SE0	Single Ended Zero
SoC	System on a chip
SOF	Start of Frame
SOP	Start of Packet
Sync	Synchronization
USB	Universal Serial Bus
VHDL	Very High speed Integrated Circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XOR	Exclusive or Gate
XST	Xilinx Synthesis Tool

Abstract

The USB Interface provides capability of utilizing the new bus standard to interface programmable device designs to personal computers. This is useful for real time applications such DAS. The USB Interface is described in VHDL which is synthesizable, implement-able and programmable on FPGA. The Xilinx ISE 8.2i is the design software suite which allowed taking the design from design entry through Xilinx device programming. The design is implemented on the Xilinx Spartan 3 (XC3S200) FPGA. The Xilinx Spartan-3 Starter Kit is used for testing the FPGA with PC using USB cable.

The outshined performance of USB as compared to the previous interfaces and the reprogrammable feature and the wide use of FPGAs are the motivations for this thesis.

Chapter 1

Introduction

FPGAs are reprogrammable devices that can integrate large amount of logic in a single IC, SoC development. Their programmability enables to avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs [8]. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs. Programmable logic devices are now becoming the trend in electronic designs because of their small size, high performance, and programmability.

In order to interface a programmable device or any peripheral to a PC, USB is a highly popular bus standard currently. USB overcomes the problems related with the traditional interfaces. To design such a complex interface system for programmable devices, EDA tools are greatly needed to accelerate the design and provide it to market quickly [1]. VHDL is one of such valuable HDLs for designing even high end electronic devices.

1.1 Background of the problem

USB is emerged as a result of the difficulties associated with the cost, configuration, and attachment of peripheral devices in the personal computer environment. The older interfaces require system resources such as I/O addresses and IRQs hereby making the system resources scarce while complicating device configuration. Also, since the serial and parallel ports support single device only, the number of peripherals that can be attached are limited. Further more, end users are faced with a variety of problems when connecting peripherals to their PCs using the older interfaces. These concerns include:

- Too many connector/cable types.
- System must be shut down to attach and detach most peripherals (no hot attachment and detachment).
- System must be restarted to install/load software.
- Manual configuration requirement (I/O address, IRQ etc.).
- One port for one device only (no expandability or not shareable Interfaces).

The traditional ports are limited to certain devices that require specific configuration. The non-standards of parallel ports, the limited bandwidth and ports of serial ports, and also their inconveniences for the user are some of the reasons to replace them with USB. Before USB, most computers came with only one parallel port, and one or two serial ports hereby limiting the number of ports on PCs. Therefore, only few numbers of peripherals could be connected to a PC. Manual configuration was required to connect a new peripheral for the first time. Dedicated cables are required for the mouse, keyboard, printer, external modem, plotter etc., which are completely different.
Figure 1-1 shows backplane of a typical PC before USB. The variety of different connectors and cables required to connect particular peripheral devices is inconvenient and confusing.

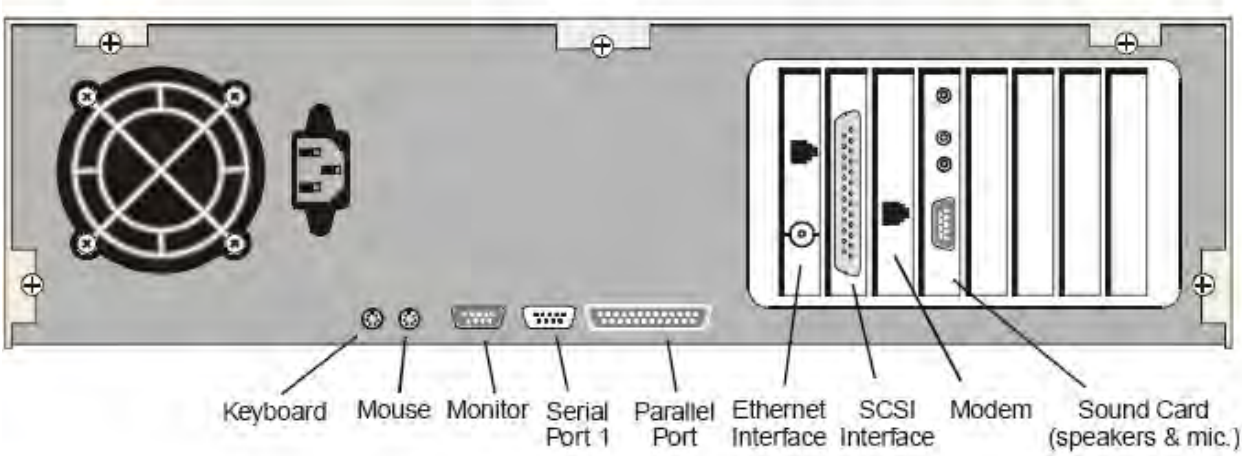

Figure 1-1: Connectors at Backplane of PC before USB [2]

USB creates a method of attaching and accessing peripheral devices that reduces overall cost, simplifies the attachment and configuration from the end-user perspective, and solves several technical issues associated with old style peripherals. Its performance outshines the traditional serial and parallel ports on a computer. USB provides a way of standard and fast connection between peripheral devices and computer. The shortcomings of the traditional interfaces have been surmounted by the USB port. Consequently, USB is now very popular and usable in almost all kinds of applications.

Almost each peripheral made these days comes in a USB standard. For example, printers, scanners, mice, keyboards, joysticks, digital cameras, DAS devices, modems, speakers, mobile phones, telephones, videophones, storage devices etc. This is because modern desktop and lap top computers appeared equipped with USB ports. Using USB, users install and use peripherals without trouble. Unlike the traditional interfaces, connection and removal of USB devices happen at any time without shutting down the entire system. In general, USB gives a single, standardized, easy-to-use way to connect up to 127 devices to a computer. The design goals of a new peripheral standard should overcome the existing shortcomings perceived by manufacturers and users, while providing for further growth, performance, and expansion. The design goals of USB include [2]:

- A single connector type to connect any PC peripheral.
- Ability to attach many peripheral devices to the same connector.
- A method of easing the system resource conflicts.
- Hot plug support.
- Automatic detection and configuration of peripheral devices.
- Low-cost solution for both system and peripheral implementations.
- Enhanced performance capability.
- Support for attaching new peripheral designs.
- Low-power implementation.

Hot Plug and Automatic configuration is crucial to satisfying end user requirements. USB can detect the attachment of a new peripheral and automatically install the relevant software needed to access the device. This process also eliminates the need to set switches and jumpers when configuring a peripheral device and eliminates the need to restart the system

when the device is attached. In short, the peripheral can simply be attached by the user and be ready for immediate use.

USB breaks away from the resource problems associated with legacy PC I/O implementations. The resource constraints related to I/O address space, IRQ lines, and DMA channels no longer exist with the USB implementation. Each device residing on the USB is assigned an address known only to the USB subsystem and does not consume any system resources. USB supports up to 127 device addresses that limit the number of USB devices supported in a single USB implementation. USB devices typically contain a number of individual registers or ports that can be indirectly accessed by USB device drivers. These registers are known as USB device endpoints. Each transaction begins with a packet transmission that defines the type of transaction being performed along with the USB device and endpoint addresses. This addressing is managed by USB software. Other non USB devices and related software within the system are not impacted by these addresses. Every USB device must have an internal default address location (called endpoint zero) that is reserved for configuring the device. Via endpoint zero, USB system software reads standard descriptors from the device. These descriptors provide configuration information necessary for hardware and software initialization. In this manner, system software can detect the device type (or class information) and determine how the device is intended to be accessed.

1.2 Objective of the thesis

The purpose of this thesis is to describe a USB Interface Hardware in VHDL, simulate, synthesize and implement it on a target FPGA.

1.2.1 Specific Objectives:

- Specifying the general and subcomponents of the device
- Developing VHDL code for the device
- Simulation, synthesis and implementation on FPGA
- Programming the FPGA using JTAG cable and testing it with PC using USB cable so that the operating system detects it as a USB device.

1.3 Methodology

To achieve the goal of the thesis, different methods were followed such as:

- Literature review.
- General block, subsystems and key components of subsystems, and I/O ports and signals specifications.
- Writing the VHDL code based on the above specifications, the functionalities of the components and also by considering the USB protocol.
- Compiling and functional simulation test of the VHDL code.
- Synthesis and Implementation of the designed hardware on target FPGA.
- Programming the FPGA and testing the programmed FPGA with PC so that the PC detects it as a USB device.

- Mainly, the technologies used in this thesis are VHDL, xilinx xc3s200 spartan3 FPGA board, JTAG cable, USB cable, and the Xilinx ISE software.

The target of the USB Interface hardware is the programmable device (FPGA). The FPGA consists of programmable blocks, interconnect wires and I/O pins for programmable interface connections.

VHDL is an IEEE standard hardware description language (HDL), used for creating electronic systems. It can be used in developing, verifying, synthesizing, and testing of hardware designs. It is particularly well suited for designing with programmable logic devices, and is gaining in popularity. Designing with large capacity CPLDs and FPGAs of 500 to more than 100,000 gates, engineers can no longer use Boolean equations or gate-level descriptions to quickly and efficiently complete a design [1]. VHDL provides high-level language constructs that enable designers to describe large circuits and bring products to market rapidly. In this thesis it is used to provide a description of the hardware for the Xilinx ISE 8.2i software for simulation, synthesis, implementation on FPGA and programming the FPGA.

1.4 The Outline of the Remaining Chapters of the Thesis

In this section, the outline of the remaining chapters is provided. Chapter 2 describes literature review (overview of USB and FPGA) giving more emphasis on USB hardware architecture and protocol which provides some background theory required to understand the rest of the thesis. Chapter 3 provides the USB Interface Hardware Design concerning general and subsystem specifications, I/O ports and signals specifications, and the VHDL implementation of the specified system. Chapter 4 discusses the results of the thesis describing how well the final implementation meets the design specifications. describes the VHDL implementation of the specified system. The final chapter, Chapter 5, provides the conclusion, application of the work, and future development that provides insight into possible future developments.

Chapter 2

Literature Review

2.1 USB Overview

USB seems a simple bus structure because of its convenience and ease of use from the end user perspective. But from the design viewpoint, it is not so. It is a complex bus architecture that requires a thorough study to well understand its nature and to design USB device interface. In order to show the intention of the thesis, the general idea of the USB architecture including USB hardware and USB protocol should be explained first. This is only the overview of USB architecture; and for more detail, it is possible to refer the references presented at the end of this document.

2.1.1 USB Hardware Architecture

2.1.1.1 USB Cable

A USB cable has four wires as can be seen in table 2.1 [13]:

Power (VCC), Ground (GND), Data Plus (D+) and Data Minus (D-).
The D+ and D- lines usually operate in differential mode (one is high while the other is low). A differential “1” is signaled when D+ is greater than D- and a differential “0” is signaled when D- is greater than D+.

Table 2.1: Pins and Signals of USB cable

Pin	Name	Cable color	Description
1	VCC	Red	+5V
2	D-	White	Data-
3	D+	Green	Data+
4	GND	Black	Ground

The maximum cable length is 5 meters between a hub and a device. With up to five levels of hubs we reach a length of 30 meters from the PC host to the device [3].

A USB Cable transports both power supply and data signals. The power supplied by the USB cable is an important benefit of the USB specification. A simpler I/O device can rely on the USB cable for all its power needs and will not require the traditional power adapter plugged into the wall. The power resource is carefully managed by the USB, with the hub device playing the major role. A hub or an I/O device can be self-powered or bus-powered.

- *Self-powered* - is the traditional approach in which the hub or I/O device has an additional power cable attached to it

- *Bus-powered* - is when a device relies solely on the USB cable for its power needs and is often less expensive.

2.1.1.2 USB Connectors

USB connectors are designed to permit any USB peripheral device to be attached to a hub port. Hub ports will be located at the back of the computer or may be associated with other peripheral devices such as printers, or are available on stand-alone hub devices. Many USB peripherals have the USB cable permanently attached, while others have detachable USB cables. If the same connector were used on both ends of a USB cable, it would be possible to connect the cable between two USB ports. To prevent a detachable cable from being plugged into two USB ports at the same time, a separate connector has been designed for the peripheral cable connection (see Figure 2.1). The two connector types are [12]:

- *Series A connectors* - provide the USB port connection to the USB peripheral cable. The series A receptacle is implemented as the hub port connector, while the series A plug is attached to the peripheral cable, permitting attachment of a USB peripheral device.

- *Series B connectors* - provide the cable connection to the USB peripheral device when a detachable cable is implemented. The series B receptacle is implemented at the peripheral and the series B plug is attached to the cable.

Each connector has four contacts: two for carrying differential data and two for powering the USB device. Note that the power contacts are longer than the data contacts to ensure that a USB device receives power prior to the data contacts mating. The connector contacts are numbered and the cable conductors are color coded for easy identification, as listed in Table 2-1.

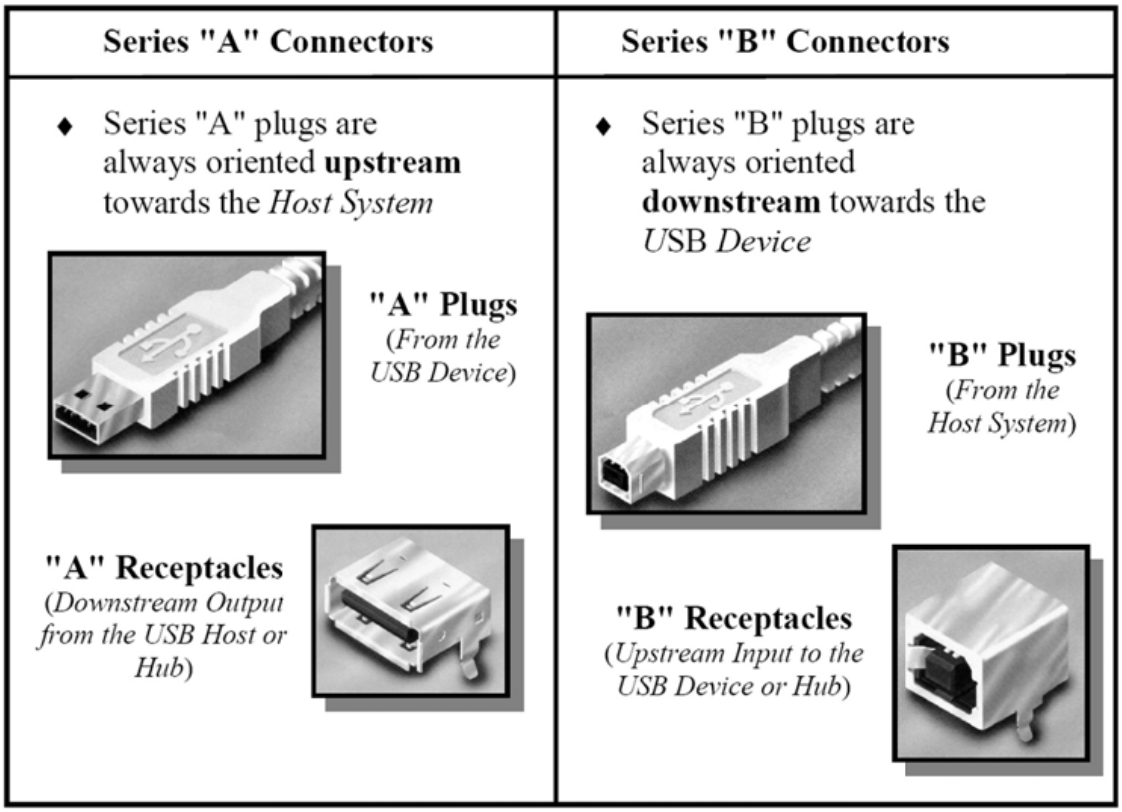

Figure 2-1: USB cable connector types

2.1.1.3 USB Speeds

Currently, USB supports three transmission rates:

- 1.5Mb/s - Low-speed devices
- 12Mb/s - Full-speed devices
- 480Mb/s - High-speed devices

A device's speed is detected when it is attached to the hub port (see section 2.1.1.5 on page 8). The 1.0 and 1.1 (1.x) versions of USB support only the 1.5 Mb/s and 12Mb/s speeds. These transmission rates were intended to support low and medium speed peripherals, while the 2.0 version of the USB specification defines a 480Mb/s rate that can support selected high-speed devices, and permits a larger number of low or full-speed devices to operate on a single bus.

2.1.1.4 USB Hub

All USB devices attach via a USB hub that provides one or more ports. Hub ports may support only full and low-speed or may support all three speeds. A device's speed is detected when it is attached to the hub port. Some devices such as keyboards and mice typically operate at low speed, while other devices such as digital telephones must operate at either full or high speed. However, depending on the device speed and the hub port capability, the following connection issues can exist:

- *Full-Speed hub ports* (*1.x hubs*) - support for LS and FS devices only
- *High-Speed hub ports* (*2.0 hubs*) - support for LS, FS, and HS devices

Hub devices provide additional ports for attaching other USB devices. Hubs can be stand-alone devices, or can be integrated into other USB peripherals such as printers or keyboards as shown in Figure 2-2. Physical USB devices that contain hubs and that have one or more internal devices attached to the hub ports are called compound devices.

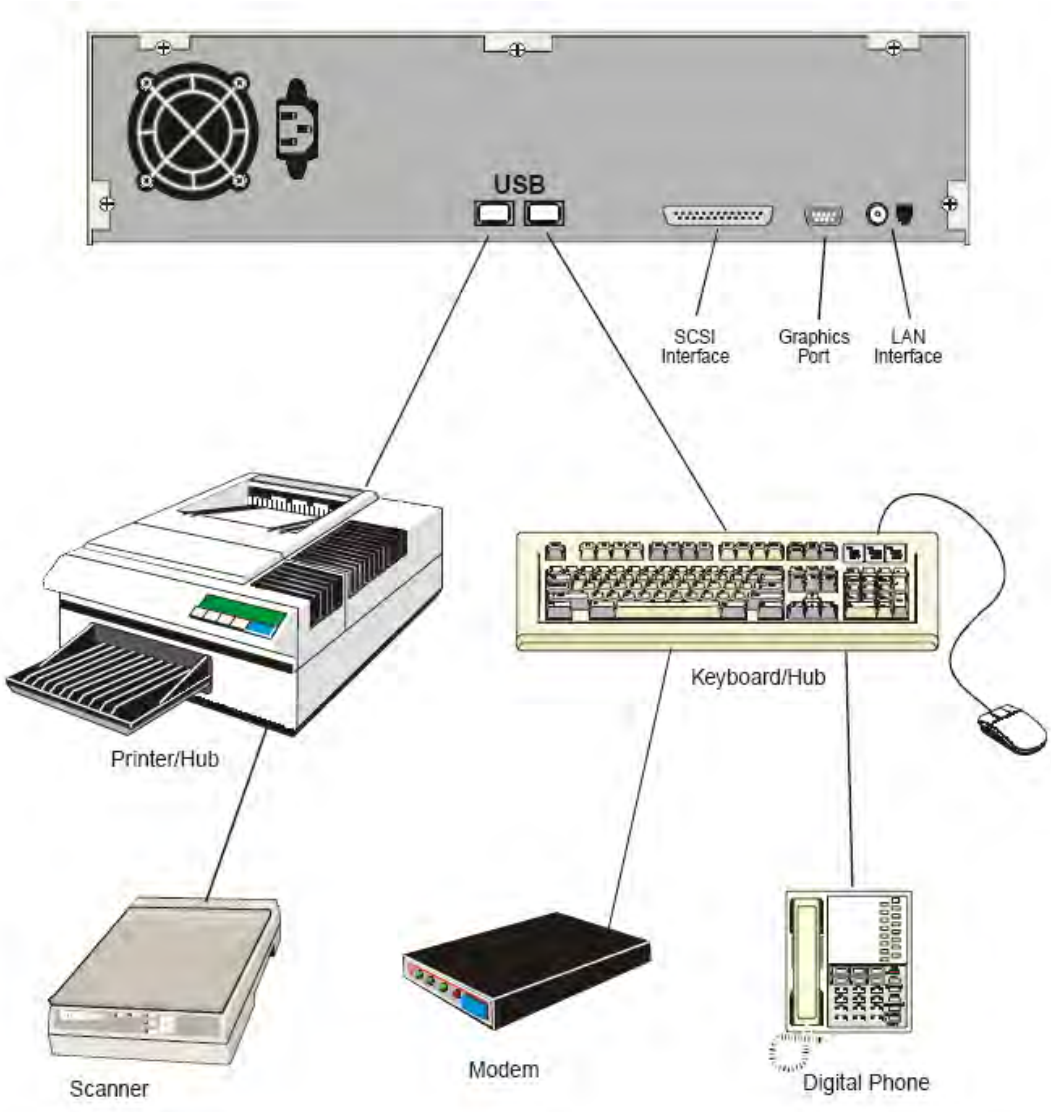

Fig. 2-2: USB Device Connections [2]

2.1.1.5 Detecting Device Attachment and Speed Detect

Before transferring information to or from a given USB device, host software must first detect its presence. USB is designed to detect the attachment of devices to the USB, after which transactions may be initiated by host software to configure the device for normal operation. USB hubs monitor each port to observe a connect or disconnect event. Device attachment can only be detected when power has been applied to the port. The pull-down

resistors on the D+ and D- lines ensure that both data lines are near ground (see Figures 2-3). When no device is attached, an electrical low on both data lines is detected. USB devices must include a pull-up resistor on either D+ or D- (depending on its speed) to enable connect detection.

2.1.1.5.1 Full-Speed Device Connect

Figure 2-3 illustrates a full-speed device connected to a hub port. When a full-speed device is attached to the port, current flows across the voltage divider created by the hub's pull-down resistor and the device's pull-up resistor on D+. Since the pull-down resistor value is 15KΩ and the device's pull-up resistor is a value of 1.5KΩ, D+ will raise to approximately 90% of Vcc. When the hub detects that D+ approaches Vcc while the other remains near ground, it knows that a full-speed device has been attached.

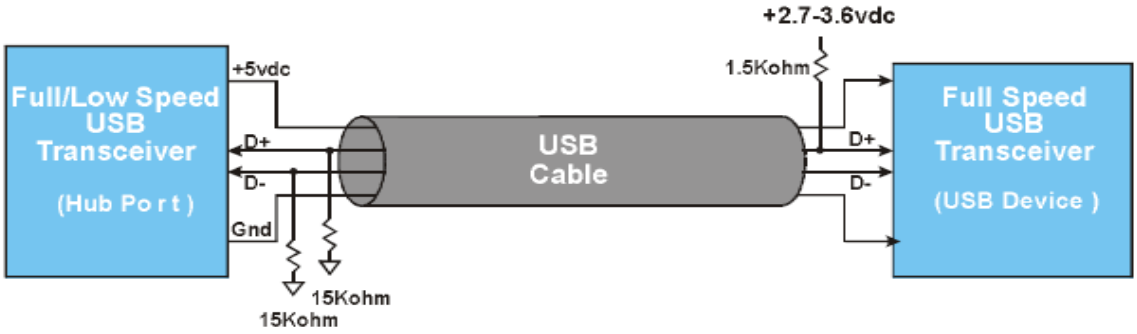

Figure 2-3: Full-Speed Device Detection

Host software polls each hub periodically to check for port events including device connection. Software then resets the device and performs device configuration.

2.1.1.5.2 Low-Speed Device Connect

Designers of low-speed devices place a pull-up on the D- line versus the D+ line used for full-speed devices. Since the pull-down resistor value is 15KΩ and the device's pull-up resistor is a value of 1.5KΩ, D- will raise to approximately 90% of Vcc. When the hub detects that D- approaches Vcc while the other remains near ground, it knows that a low-speed device has been attached.

2.1.1.6 Detecting Device Disconnect

The hub also monitors each port that is currently connected for the possibility of the device being removed. A hub detects the disconnect when it observes a single-ended zero (when D+ and D- fall below VIL).

2.1.1.7 Bus Idle

The idle state of the bus depends on whether a low or full-speed device is attached. Bus idle is valid when D+ (D-) is pulled up to the 2.7V minimum to 3.6V maximum level and D- (D+) is pulled below 0.8V.

Table 2-2: USB Bus States

Bus State	Signaling Levels	
	From Originating Driver	At Receiver
Differential "1"	D+ > V_{OH} (min) and D- < V_{OL} (max)	(D+) - (D-) > 200mV and D+ > V_{IH} (min)
Differential "0"	D- > V_{OH} (min) and D+ < V_{OL} (max)	(D-) - (D+) > 200mV and D- > V_{IH} (min)
Single-ended 0 (SE0)	D+ and D- < V_{OL} (max)	D+ and D- < V_{IL} (max)

2.1.2 USB Protocol

This section describes the USB protocol and how a USB connection is established with a host computer (regarding windows operating system).

USB devices are divided in classes. Each device class has its own standardization of the data format. For each device class a standard driver is available for example in Windows XP. Some of the main device classes are:

Power device
Audio
Communication
Mass storage
Printer
Monitor
HIDs

When a device is detected on the USB, system software must access the device's descriptors to determine the device type and operational characteristics. The device descriptor gives information about the entire USB device, such as the USB version, maximum packet size, vendor ID and product ID. Microsoft Windows will then try to enumerate the device. Enumeration is the process of determining what device has just been connected to the bus and what parameters it requires such as power consumption, number and type of endpoint(s), class of product etc. The host will then assign an address to the device and enable a configuration allowing the device to transfer data on the bus.

2.1.2.1 USB Transfers

The following four transfer types have been defined by the USB specification, each of which reflects the nature of transfers that may be required by a USB device endpoint [2, 3, and 5]:

- *Interrupt transfers* - are used to poll devices to determine if they have data that needs to be transferred (i.e., if they have an interrupt request pending). If a device does not currently have data to send (i.e., no interrupt is pending), then the device returns negative acknowledgement, indicating that no data is available to send at this time. Interrupt transfers are also used to send data to a device on a scheduled basis. Interrupt transfers are used for devices that regularly transmits a small amount of data such as a keyboard or a mouse.

- *Bulk transfer* - a bulk transfer is used for transferring large blocks of data that have no periodic or transfer rate requirement. An example of a bulk transfer is a print job being transferred to a USB printer. No problems will be incurred if the transfer occurs at a slow rate, other than impatience of the user who is waiting for the print job to emerge.

- *Isochronous transfer* - an isochronous transfer requires a constant delivery rate. Devices that use isochronous transfers must ensure that rate matching between the sender and receiver can be accomplished. For example, a USB microphone and speaker would use isochronous transfers to ensure that no frequency distortion results from transferring data across the USB. Isochronous data delivery is characterized by the need to provide data on a timely basis and is used where timeliness is more important than verifying accurate delivery of data. For this reason valid data delivery is not guaranteed during isochronous transfers.

- *Control transfers* - control transfers are used to transfer specific requests to USB devices and are most commonly used during device configuration. Each device must implement a default control endpoint (always endpoint zero) used for configuring the device, controlling device states, and other aspects of the device's operation.

2.1.2.2 USB Packets

Packets are the basic building blocks of USB transactions. A sequence of packets exchanged between the host and device is called transaction. USB transactions typically consist of three packets as illustrated in Figure 2-4 [2].

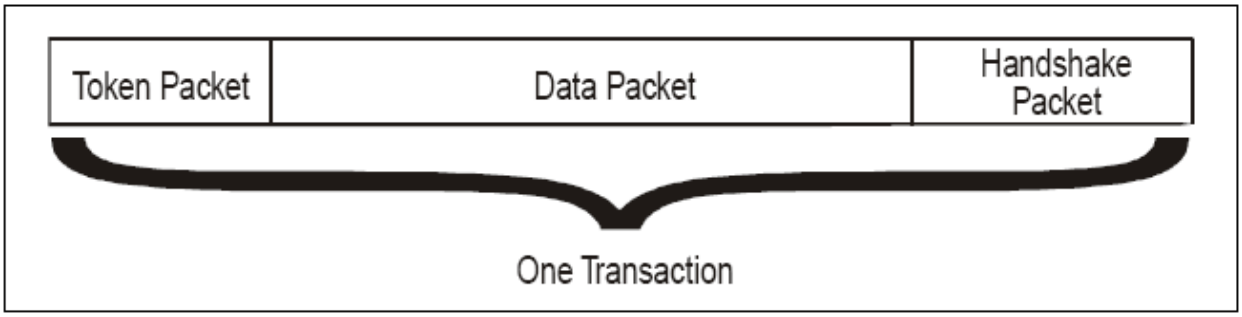

Figure 2-4: The Three Phases of USB Transaction

Figure 2-5 illustrates the basic format of a USB packet. Immediately preceding each packet is a synchronization sequence that permits USB devices to synchronize to the data rate of the incoming bits within the packet. The type of packet is defined by a bit pattern called a PID. Following the PID is packet-specific information (e.g., an address or data) that varies depending on the packet type. Finally, each packet ends with a sequence of CRC bits, used to verify correct delivery of the packet-specific information. The end of each packet is identified by an EOP. Each type of packet is detailed in the following sections.

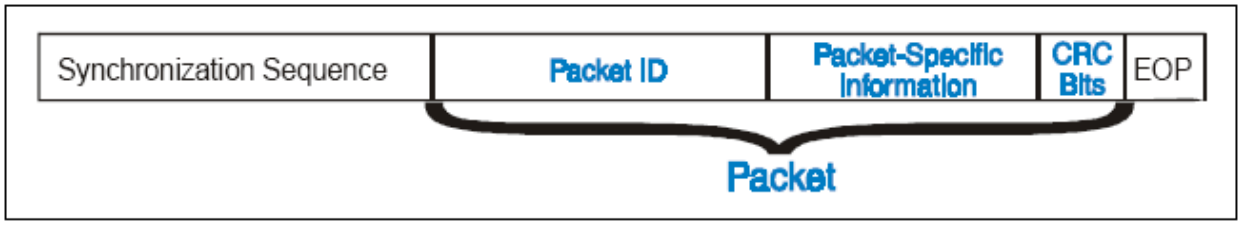

Figure 2-5: USB Packet Format

2.1.2.2.1 Token Packets

Token packets define the type of transaction that is to be broadcast over the USB. All transactions begin with a token packet that defines the target device, endpoint number, and the direction of the data transfer. Four types of token packets are defined by the USB specification:

- *SOF* - indicates start of the next 1ms frame.
- *IN* - specifies a USB transaction used to transfer data from a target USB device to the system.
- *OUT* - specifies a USB transaction used to transfer data from the system to a target USB device.
- *SETUP* - indicates the start of a control transfer. SETUP is the first stage of a control transfer and is used to send a request from the system to the target USB device.

Table 2-3 lists the token types supported. It also describes the packet-specific information embedded within the packet and the action specified. Each token is identified by its packet ID, as shown in column three.

Table 2-3: USB Tokens

PID Type	PID Name	PID[3:0]	Description of Token Packet
Token	SOF	0101b	Contains start of frame (SOF) marker and frame number. The SOF token is used by isochronous endpoints to synchronize its transfers.
Token	SETUP	1101b	Contains USB device address and endpoint number. Transfer is from host to function for setting up a control endpoint (e.g., configuration).
Token	OUT	0001b	Contains the USB device address and endpoint number. The transfer is from host to function.
Token	IN	1001b	Contains the USB device address and endpoint number. The transfer is from function to host.

2.1.2.2.1.1 IN Packet

When software requires reading information from a given device, an IN token is used. The IN packet let know the target USB device that data is being requested by the system. As illustrated in Figure 2-6, an IN token packet consists of the ID type field, the ID check field, the USB device and endpoints addresses, and five CRC bits. An IN transaction starts with an IN packet broadcast by the root hub, followed by a data packet returned from the target USB device, and in some cases concluded with a handshake packet sent from the root hub back to the target device to confirm receipt of the data.

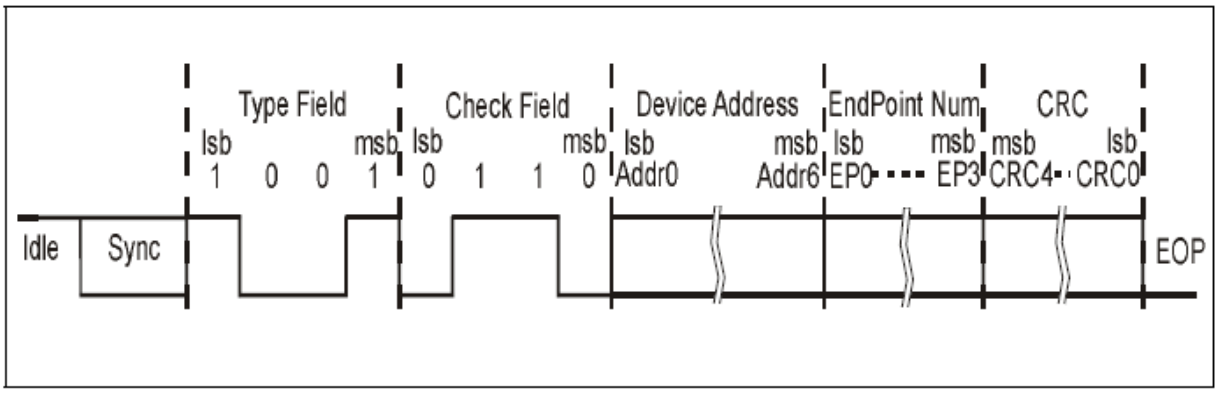

Figure 2-6: IN Token Packet Format

2.1.2.2.1.2 OUT Packet

System software specifies an OUT transaction when data is to be transferred to a target USB device. An OUT token packet consists of the packet ID or type field, the type check field, the USB target device and endpoint ID, and a 5-bit CRC as shown in Figure 2-7. The OUT token packet is followed by a data packet and a handshake.

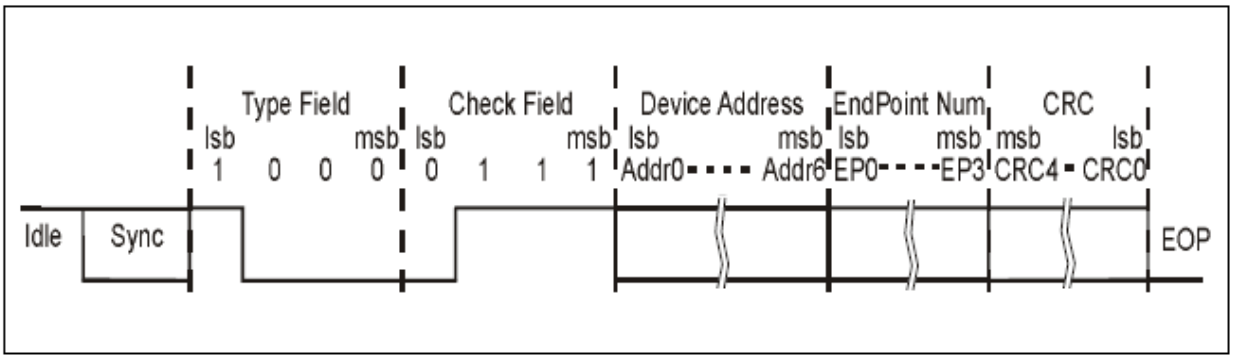

Figure 2-7: OUT Token Packet Format

2.1.2.2.1.3 SETUP Packet

SETUP packets are used only during the setup stage of control transfers. It starts a control transfer, and is defined as the setup stage. A setup transaction is similar in format to an OUT transaction (see Figure 2-8) The SETUP packet is followed by a DATA0 packet, and an acknowledge packet. The SETUP packet transfers a request to be performed by the target device.

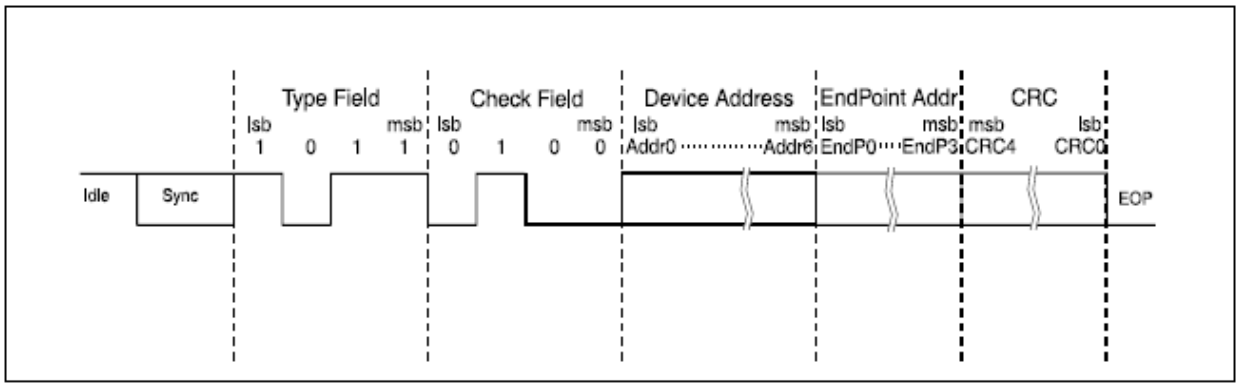

Figure 2-8: SETUP Token Packet Format

2.1.2.2.2 Data Packets

The data consists of a data packet that carries the payload associated with the transfer. A data packet can carry a maximum payload of 1023 bytes of data (isochronous transactions) during a single transaction; while the other transfer types have maximum data payloads of 64 bytes at full speed. Direction of a data packet transfer is specified by the transaction type and may be used to transfer data either to or from a target USB device as listed in Table 2-4 below.

Table 2-4: Direction of Data Packets

Transaction Type	Direction of Data Packet
IN transaction	from USB device
OUT transaction	to USB device
SETUP transaction	to USB device

Two types of data packets (DATA0 and DATA1) are defined to support synchronization of long transfers between the sender and receiver. For example, if a long transfer is being sent from the host to a printer, the transfer will be performed in small blocks, usually over large number of frames. To verify that a data transaction is not missed during a long transfer, a technique called data toggle can be employed. Consider the case where a data packet is sent, the acknowledgement is lost, and the sender of the original packet decides to resend that packet. The receiver must be able to distinguish between a new packet and a retransmitted old packet. This is possible if consecutive data packets toggle between the DATA0 and DATA1 PIDs. Figure 2-9 shows DATA0 packet format.

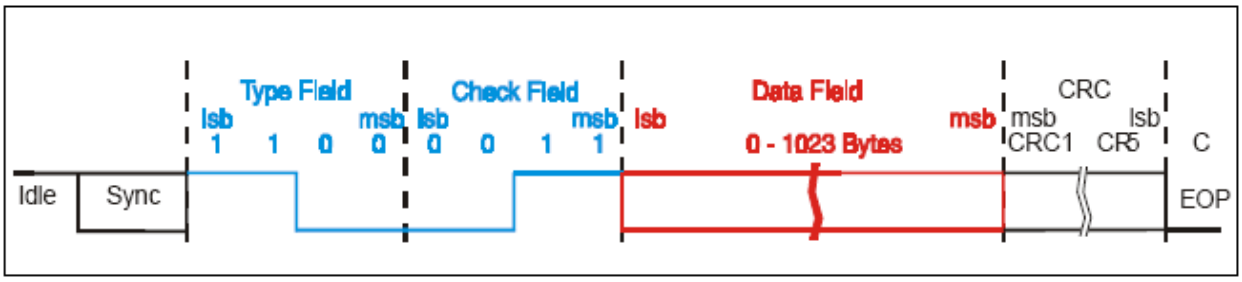

Figure 2-9: DATA0 Packet Format

2.1.2.2.3 Handshake Packets

All USB transfers (except isochronous) are implemented to guarantee data delivery, and include a handshake packet to verify a successful data transfer. USB devices use handshake packets to report the completion status of a given transaction. The receiver of the data payload (either the target device or the root hub) is responsible for sending a handshake packet back to the sender. Three possible results can be reported via different handshake packets (see Figure 2-10):

- *ACK* - this acknowledges error-free receipt of the data packet.
- *NAK* - reports to the host that the target is temporarily unable to accept or return data. During interrupt transactions NAK signifies that no data is currently available to return to the host (i.e., no interrupt request is currently pending).
- *STALL* - used by the target to report that it is unable to complete the transfer and that software intervention will be required for the device to recover from the stall condition.

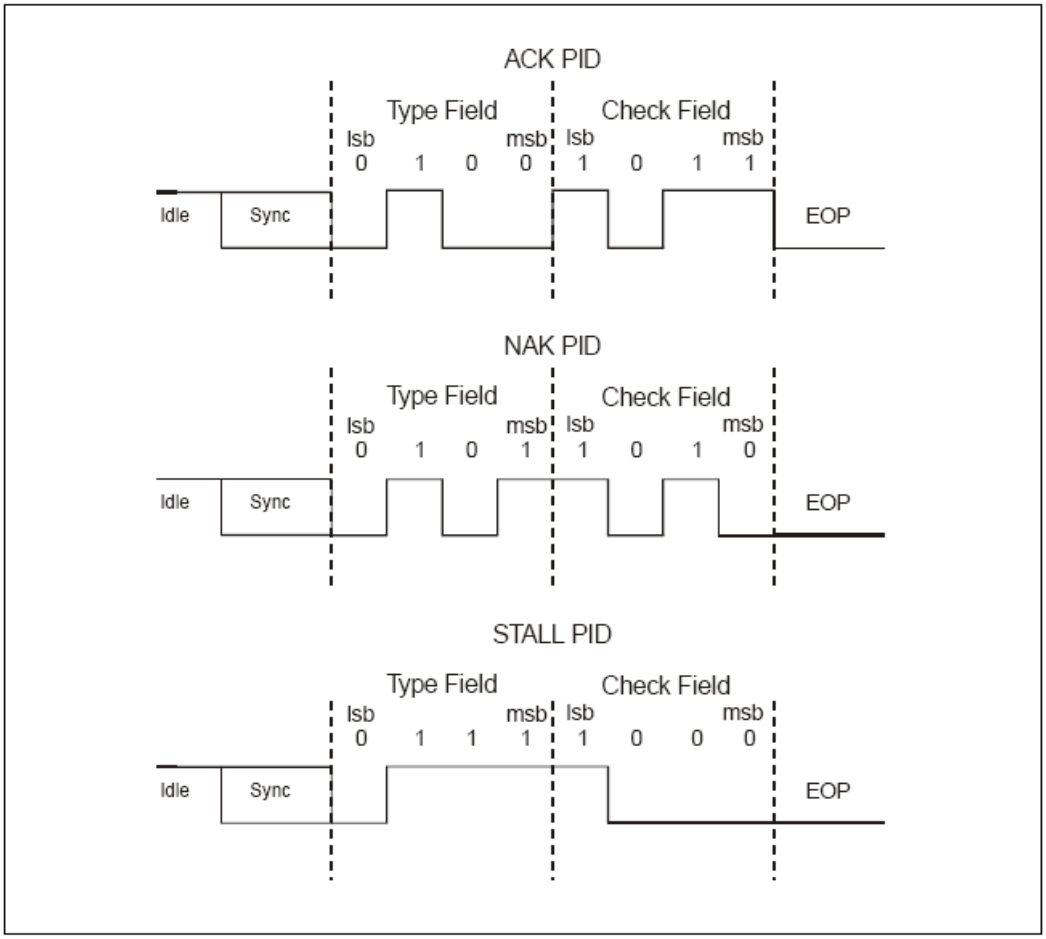

Figure 2-10: Handshake Packet Formats

2.1.2.3 NRZI Encoding

Data transferred via the USB is encoded using NRZI encoding to help ensure integrity of data delivery, without requiring a separate clock signal be delivered with the data. Figure 2-11 illustrates the steps involved in transferring information across a USB cable segment. Table 2-5 shows NRZI encoded bit stream example of Figure 2-11. NRZI encoding is performed by the USB transmitter. Zeros in the NRZI data stream are represented by transitions while 1s are represented by the absence of a transition. Every NRZI encoded bit depends on the previous encoded bit. The NRZI encoder must maintain synchronization with the incoming data stream to correctly sample the data.

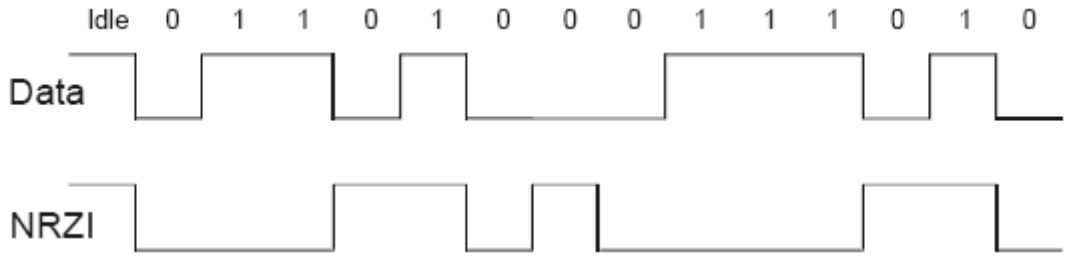

Figure 2-11: NRZI Encoding

Table 2-5: NRZI Encoded bit stream example

Idle	1													
Data	0	1	1	0	1	0	0	0	1	1	1	0	1	0
NRZI	0	0	0	1	1	0	1	0	0	0	0	1	1	0

2.1.2.4 Synchronization Sequence

Figure 2-12 illustrates the synchronization sequence. The synchronization sequence consists of eight bits starting with seven consecutive logic 0s and ending with a logic 1. Since zeros are encoded with transitions of the differential data lines, the seven zeros each create a transition during each bit time, thus, providing a clock that can be synchronized to. The synchronization sequence also alerts USB receivers that a packet is being sent, which will immediately follow the 8-bit synchronization sequence.

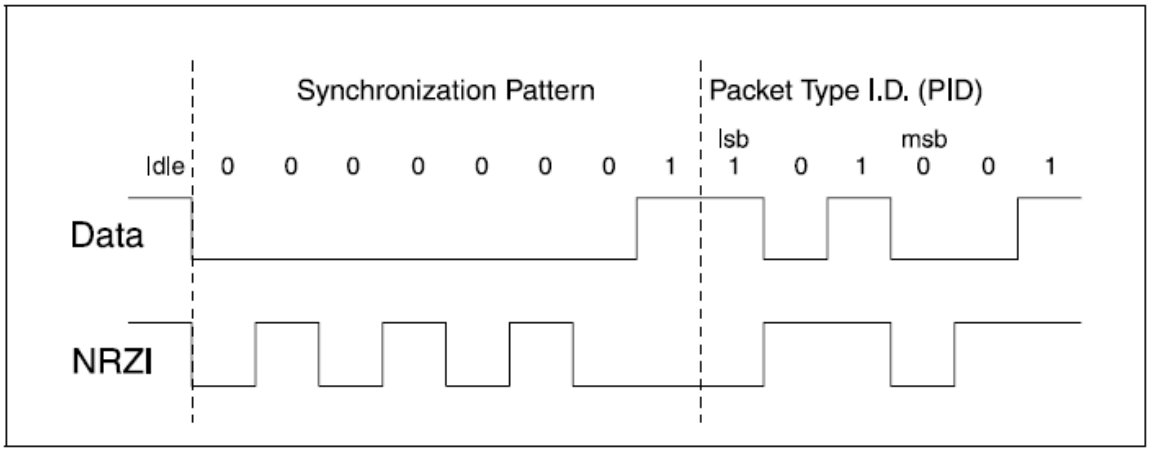

Fig. 2-12: Synchronization Sequence

Packets can be broadcast at a full speed (12Mb/s) bit rate or a low speed (1.5Mb/s) bit rate depending on the speed of the device being accessed. The USB receiver must detect the logic state of each bit value within the packet by sampling the data lines at the correct point during each bit time. The synchronization sequence is transmitted at the transfer speed being used, allowing the receiver to synchronize to either incoming data rate.

2.1.2.5 Packet Identifier

Packet identifiers define the purpose and thus the format and content of a given packet. Packets are grouped into three major categories: token packets, data packets and handshake packets (see USB Packets on page 11)

As shown in Figure 2-13, packet IDs consist of a 4 bit identifier field followed by a 4 bit check field. The check field contains the inverted value (1's complement) of the packet ID value.

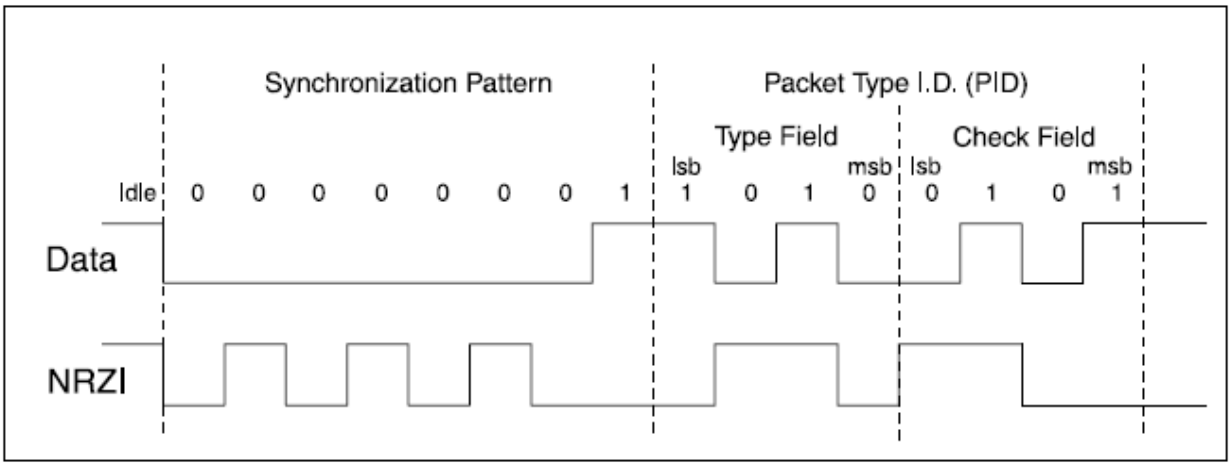

Figure 2-13: Packet Identifier Format

2.1.2.6 Bit Stuffing

Bit stuffing occurs when six consecutive 1's are transmitted; and it forces transitions into the NRZI data stream in that event as shown in figure 2-14. This ensures that the receiver detects a transition in the NRZI data stream at least every seventh bit time. This enables the receiver to maintain synchronization with the incoming data. The transmitter of NRZI data is responsible for inserting a 0 (stuffed bit) into the NRZI stream. The receiver must be designed to expect an automatic transition following six consecutive 1s and discard the 0 bit that immediately follows the sixth consecutive 1.

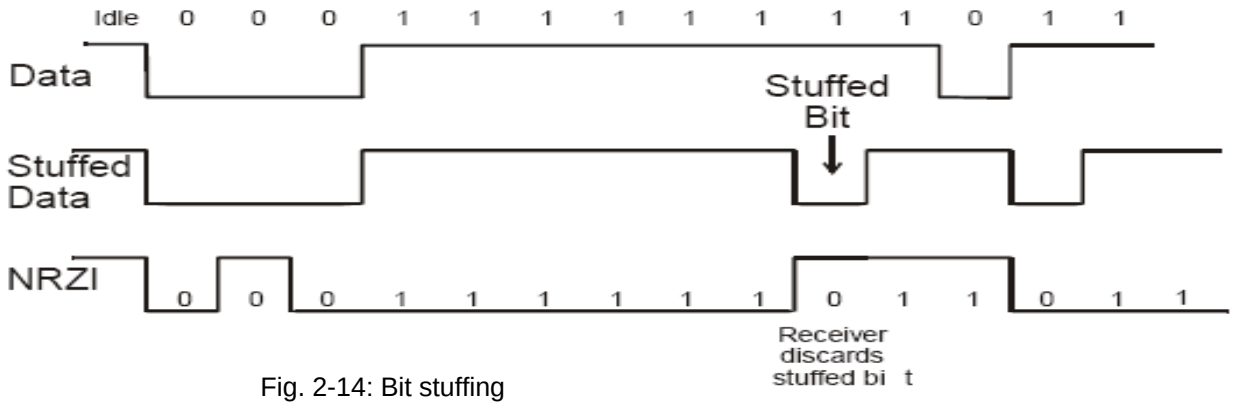

Fig. 2-14: Bit stuffing

2.1.2.7 Packet-Specific Information

Each packet contains information that is related to the job it performs. The information may consist of a USB device address, a frame number, data to be transferred to or from a USB device, etc. This information is crucial to the success of a given transaction and is verified at the end of the packet with CRC bits.

2.1.2.8 Cyclic Redundancy Checking

The USB agent that sends a given packet computes either a 5-bit or 16-bit CRC depending on the packet type. Data packets use a 16-bit CRC, while all other packet types use the 5-bit CRC. The CRC is generated and checked for the packet-specific data only, since the packet ID has its own check bits.

2.1.2.9 End of Packet

The end of each packet is signaled by the sending agent by driving both differential data lines low for two bit times followed by an idle for one bit time. The agent receiving the packet recognizes the EOP when it detects the differential data lines both low for greater than one bit time.

2.2 FPGA Overview

2.2.1 General Overview

An FPGA is a reprogrammable chip. If it is programmed today for one specific application, then it could be reprogrammed later on for another special application instead. Unlike a normal computer program, a program for an FPGA is not a sequence of instructions; rather, it is a description of how the logic gates in the chip should be connected to each other and to I/O pins. An FPGA program is usually written in hardware description language and then translated into a physical design by special synthesis software.

An FPGA is an invaluable programmable device that has a kind of inherent parallelism. I.e., instead of executing the elements of one's program one by one, it implements all elements all the time concurrent with each other. FPGAs can be used in data acquisition, where it can be convenient to preprocess a signal before sending the data to a PC.

FPGAs consist of resources like logic blocks, I/O blocks for connecting to the pins of the FPGA packet and interconnection wires. Interconnect wires connect the logic block together. Programmable switches allow the logic blocks to be interconnected in many ways. Programmable connections also exist between the I/O blocks and the interconnect wires.

The target FPGA used for this thesis is the Xilinx spartan3 xc3s200 FPGA. The FPGA board shown in Figure 2-15a and 2-15b are the Xilinx Spartan-3 Starter Kit Board (bottom and top sides respectively). It has onboard LEDs and buttons and a segment display, and other accessories as shown in the figures; of course all connected to programmable I/O pins on the FPGA. The board can also be connected to other hardware through its RS-232, VGA or PS/2 ports. The block diagram of Xilinx Spartan-3 Starter Kit Board is shown in Figure 2-16.

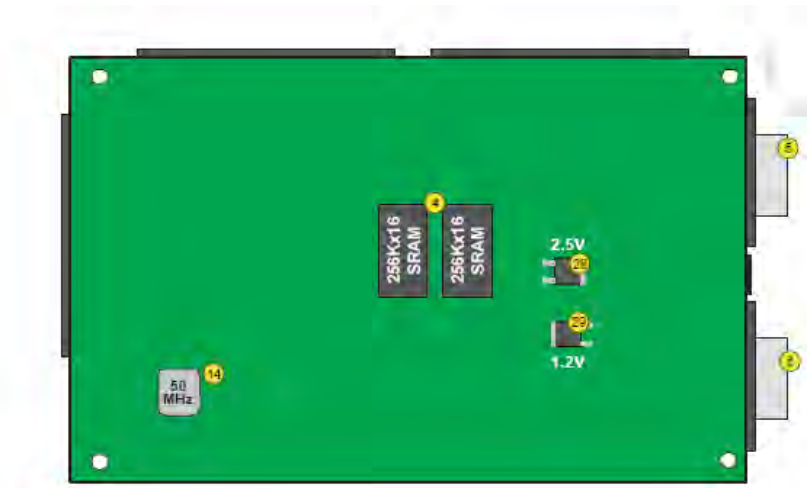

Figure 2-15a: Xilinx Spartan-3 Starter Kit Board (Bottom Side)

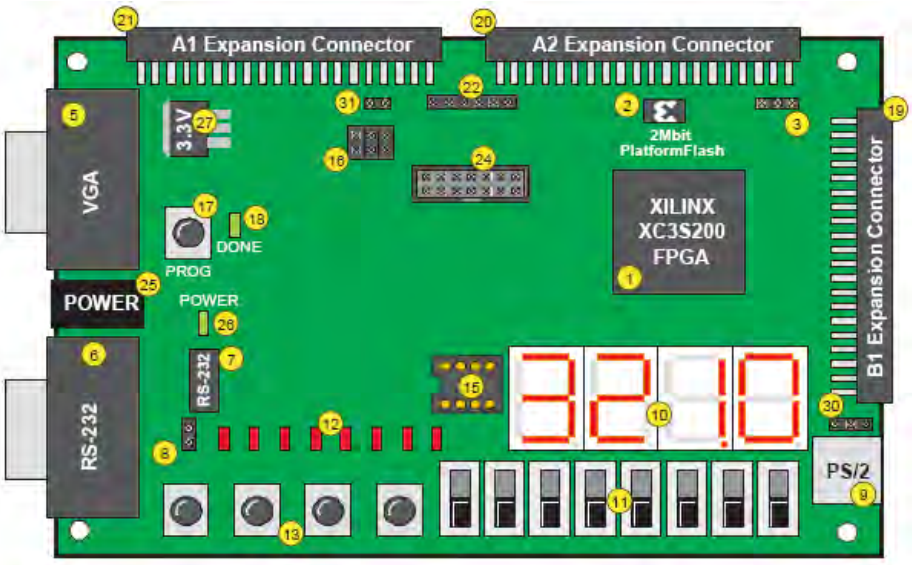

Figure 2-15b: Xilinx Spartan-3 Starter Kit Board (Top

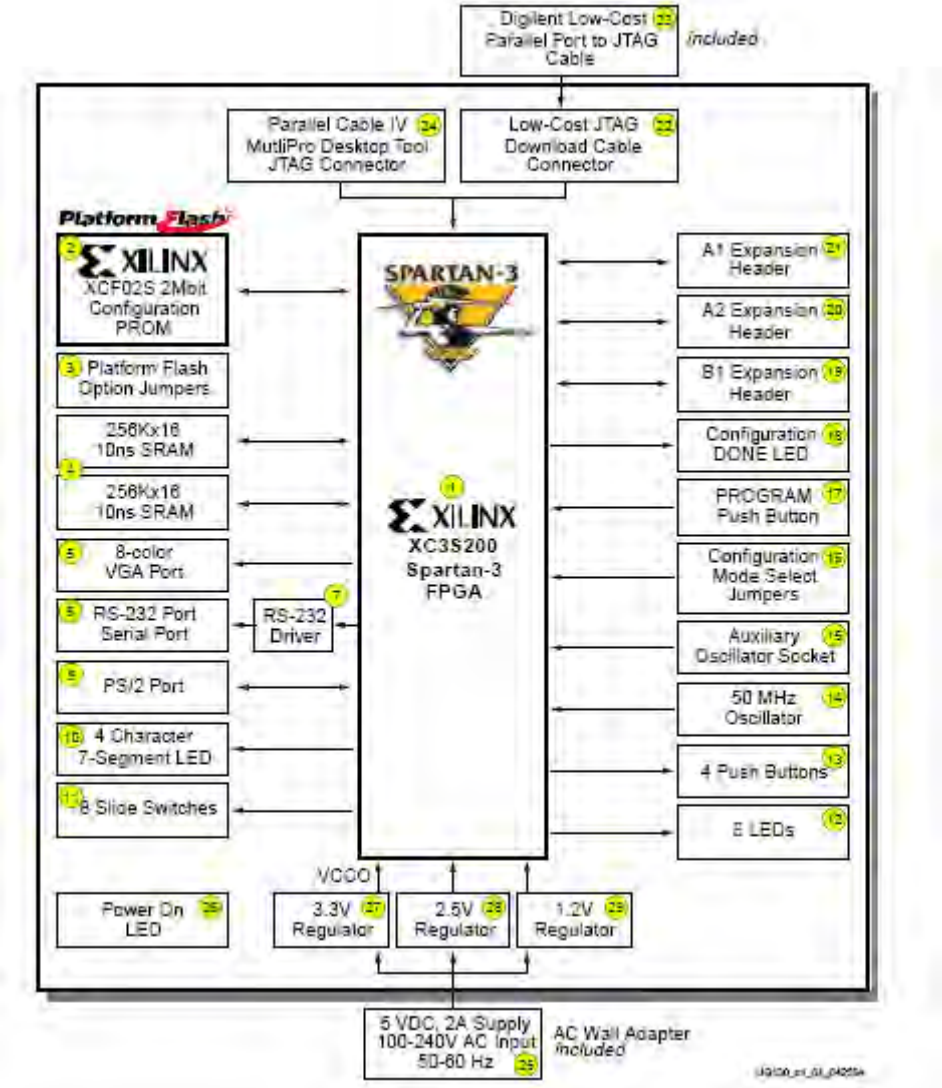

Figure 2-16: Xilinx Spartan-3 Starter Kit Board Block Diagram

The Spartan-3 FPGAs are the first platform among several within the family and are increasingly being used for many systems and efficient SoC designs. Please refer the web site http://www.xilinx.com/ for more detail.

2.2.2 Architectural Overview

The Spartan-3 Generation architecture consists of five fundamental programmable functional elements [8]:

- *CLBs* - Contain flexible LUTs that implement logic plus storage elements used as flip-flops or latches. CLBs perform a wide variety of logical functions as well as store data.

- *IOBs* - Control the flow of data between the I/O pins and the internal logic of the device. IOBs support bidirectional data flow plus 3-state operation.

- *Block RAM* - Provides data storage in the form of 18-Kbit dual-port blocks.

- *Multiplier Blocks* - Accept two 18-bit binary numbers as inputs and calculate the product.

- *DCM Blocks* - Provide self-calibrating, fully digital solutions for distributing, delaying, multiplying, dividing, and phase-shifting clock signals.

2.2.3 Configuration

Spartan-3 devices are configured by loading application specific configuration data into the internal configuration memory. Configuration is carried out using a subset of the device pins, some of which are "Dedicated" to one function only, while others, indicated by the term "Dual-Purpose", can be re-used as general-purpose User I/Os once configuration is complete. Depending on the system design, several configuration modes are supported, selectable via mode pins. The mode pins M0, M1, and M2 are dedicated pins. Spartan-3 supports the following five configuration modes:

- Master Serial (M1=M2=M3=0)
- Slave Serial (M1=M2=M3=1)
- Master Parallel (M1=M2=1, M3=0)
- Slave Parallel (M1=0, M2=M3=1)
- JTAG (M1=1, M2=0, M3=1)

The XC3S200 FPGA Attributes are summarized and listed as follows [8]:

Device:	XC3S200
System Gates:	200K
Logic Cells:	4,320
CLB Arrays (one CLB=Four Slices):	Rows 24
	Columns 20
	Total CLBs 480
Distributed RAM (bits):	30K
Block RAM (bits):	216K
Dedicated Multipliers:	12
DCMs:	4
Maximum User I/O:	173
Maximum Differential I/O pairs:	76

Chapter 3

USB Interface Hardware Design

3.1 General System Specifications

The general block diagram of how the USB interface will be used is shown in Figure 3-1. The USB Interface is connected to the host and the programmable device as shown in the figure. The programmable device can be microcontroller, FPGA or other programmable device that can generate and read digital signals. The programmable device has to let the USB Interface know when ever it is ready to send data. The device driver will search the USB Interface regularly to see if valid data is in a particular endpoint. If there is data in requested endpoint, the USB Interface will send its contents. To transfer data from the PC to the device, the host must send data to an endpoint on the USB Interface. The USB Interface then sends a signal to the programmable device indicating it that there is data for one of its endpoints that can be read. The programmable device communicates with the USB Interface through the endpoints.

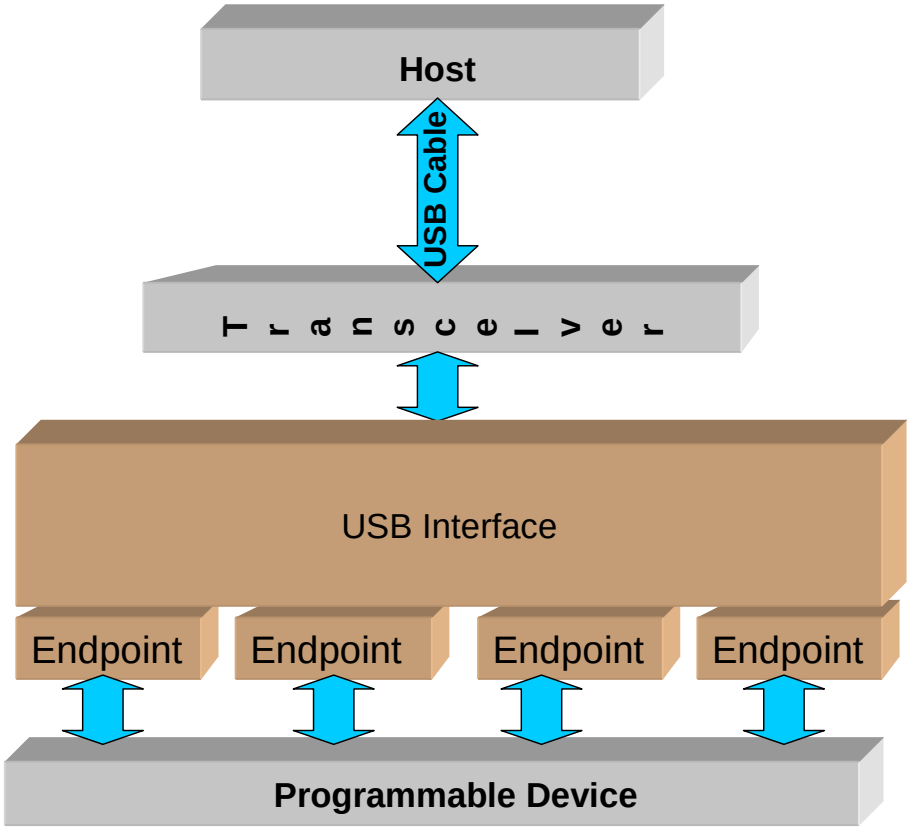

Figure 3-1 General block diagram of the USB Interface connected to the host and programmable device

The USB Interface has error checking capability. It performs CRC checks on the token and data packets that it receives. If there is error, it sends NAK signal back to the host.

3.2 The Subsystems Specification

The USB Interface can be subdivided in to many sub systems. The main three components are the transmitter, receiver and controller as shown in Figure 3-2.

3.2.3 Controller

The controller is responsible to manage all activities of the USB transactions taking place under the USB Interface. The controller is basically a Mealy state machine. It keeps track of different states of the transactions. The detail will be discussed in section 3.4.3.

3.3 key functional elements of the transmitter and receiver

The three major parts of the USB Interface design are the transmitter, receiver and controller. The transmitter and the receiver form the data paths of the USB Interface. In Figure 3-3, the labels "Transmitter section" and "Receiver section" indicate the key components of the transmitter and the receiver (shaded blocks to the right and left of the Controller) respectively, which will be discussed in the subsequent sections. The arrows indicate the direction of information transmission between the blocks.

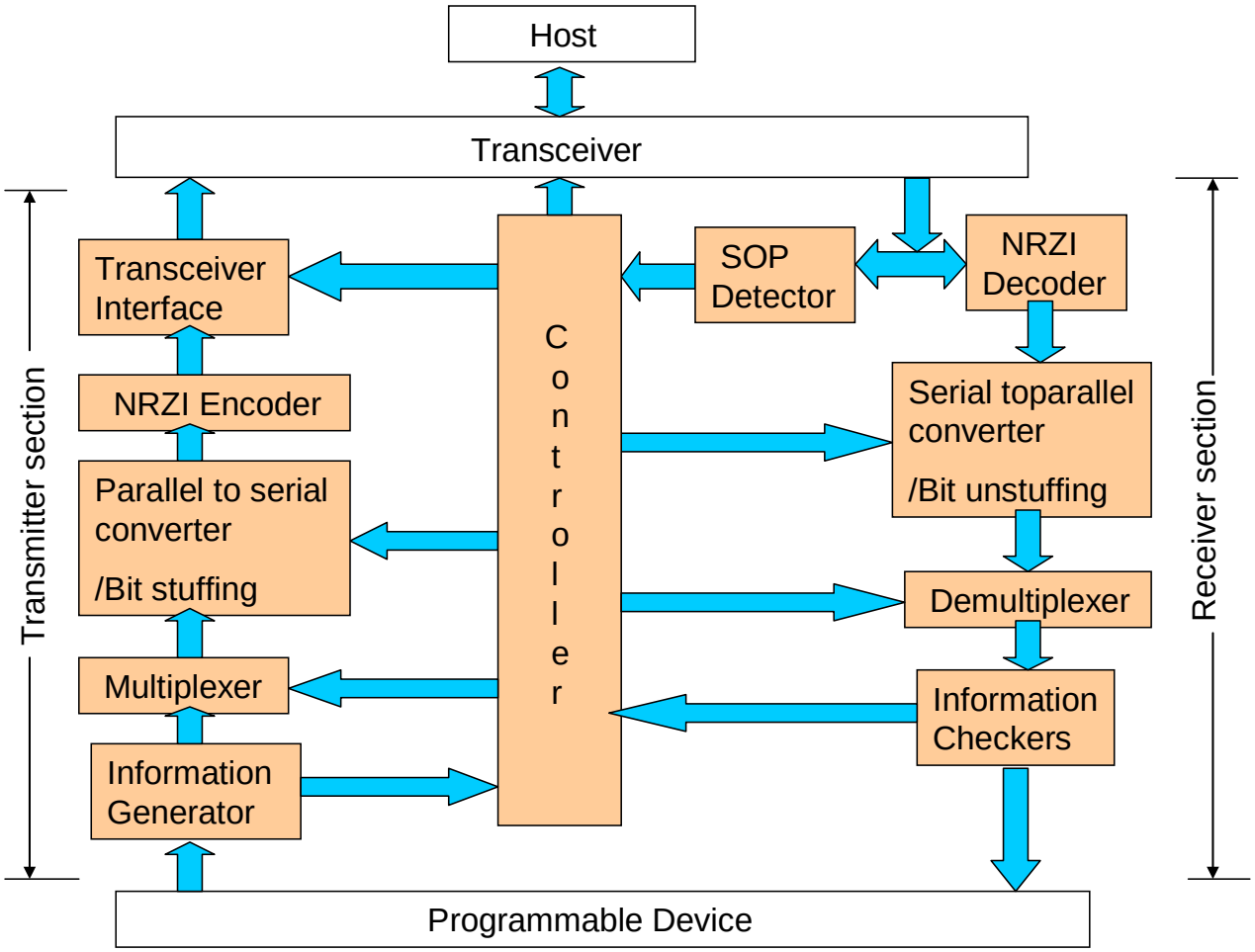

Figure 3-3: Key components of the transmitter and receiver and directions of data flow.

3.4 VHDL Implementation

Figure 3-4 shows the specifications of input and output ports and signals of the controller, the transmitter, the receiver and the system. The three main components of the USB

Interface are connected to each other, to the transceiver as well as to the USB bus. The function of the signals and the VHDL implementation of the system are explained step by step. The VHDL source code can be found in the Appendix section of this document.

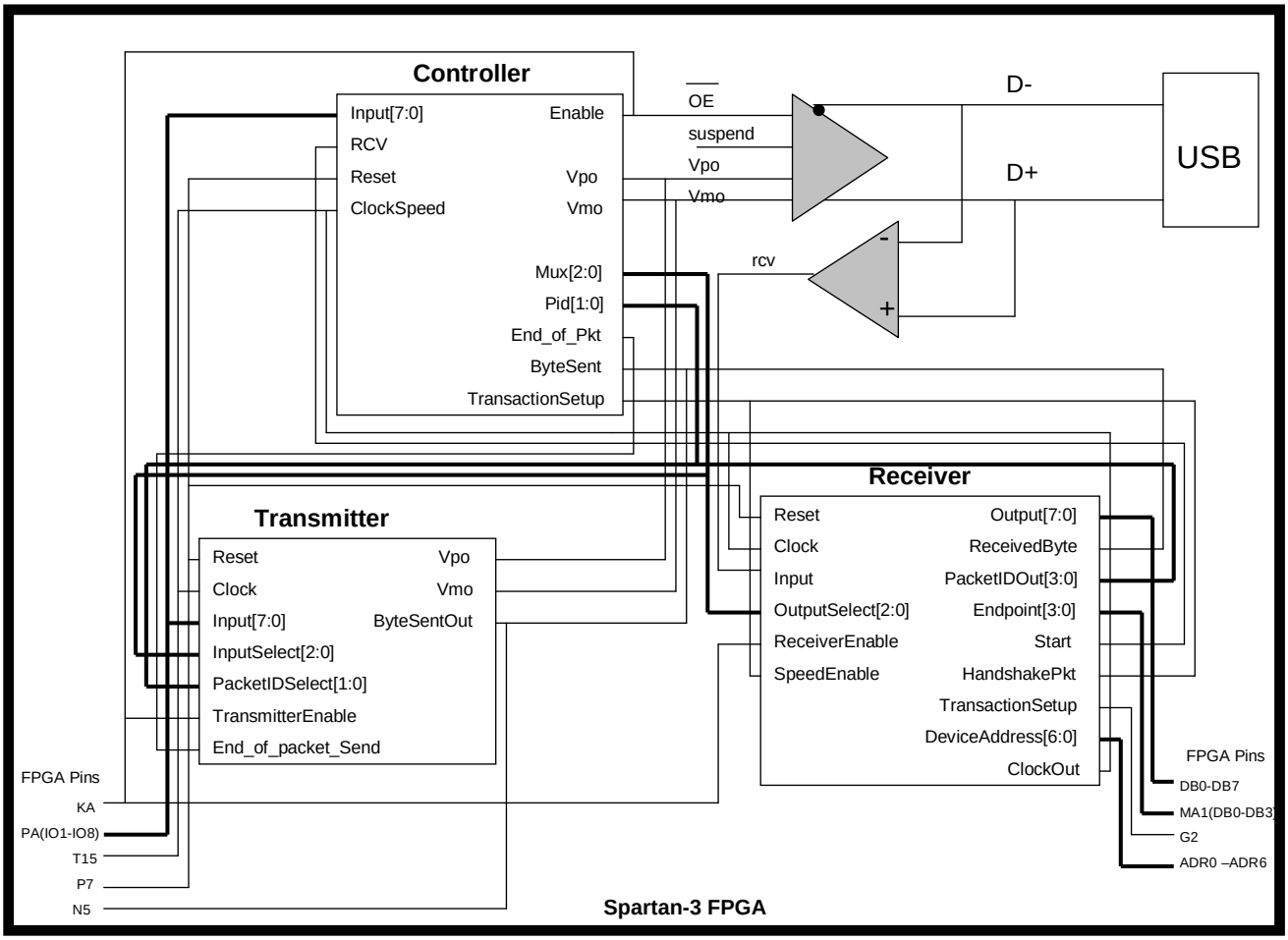

Figure 3-4: I/O signals specifications of the USB Interface

The USB Interface doesn't send data on the USB bus at the same time as the host. The controller sets /OE to low at the same time it enables the transmitter; and sets /OE to high when it enables the receiver. This avoids occurrence of error operations.

For data transmission and reception, the controller has to activate or deactivate the transceiver. The rcv is the pin on which serial data is received from the host. The vpo and vmo signals are used to transmit data on the USB bus (see section 3.4.1.5 for the detail).

3.4.1 VHDL Implementation of the Transmitter

The transmitter carries out the following tasks:

- NRZI encoding
- Parallel to serial conversion
- Bit Stuffing

- Sending start of packet sequence
- Calculating and sending 16-bit CRC for error checking
- Generate the correct PID as specified by the controller
- Letting the controller to decide the type of byte to send

The input and output ports of the transmitter are specified as shown in Figure 3-5. The purpose of all the signals used in this part is also listed. The step by step design process of the transmitter and the algorithmic approach for the VHDL implementation of the design are described in the corresponding topics.

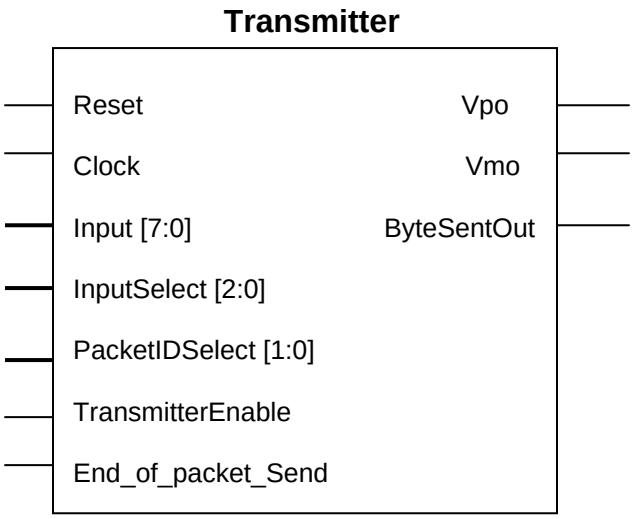

Figure 3-5: I/O ports of the transmitter

Signals used for the design of the transmitter and their purposes are described below:

PreviousOutput -used for keeping track of last output for NRZI encoding
DataStuffed -indicates stuffed bit
CountBit [0:6] -used to contain consecutive 1s
CountShift [0:8] -used to contain number of bits sent
ShiftRegister [7:0] -used as serial to parallel conversion shift register
Muxoutput [7:0] -used to contain selected output of multiplexer
SentByte -used for indicating that byte is sent
Output -triggers the changes of Vpo and Vmo
CRCState [15:0] -contains present state of CRC
NextCRCState [15:0] -contains next CRC state
CRC16Out [15:0] -contains the CRC result
PacketID [7:0] - contains type of packet

3.4.1.1 Information Generator

Information Generator is one of the key components of the transmitter as shown in Figure-3-3. It is responsible for generating correct PID and calculating 16-bit CRC.

The PID generation is performed after the controller specified the type of PID to be generated through the **PacketIDSelect** port. The controller determines what type of PID to send after making sure that the data has been received correctly. If data has been received correctly, an ACK (Acknowledgement) packet is sent. Otherwise, a NAK (Negative acknowledgement) packet is sent. The piece of VHDL code shown below describes PID generation process.

```
Process (PacketIDSelect)
begin
    case PacketIDSelect is
        when "00"=> PacketID<="01001011";      ---ACK
        when "01"=>PacketID<="01011010";       ---NAC
        when others=> PacketID<="11000011";    ---Data0
    end case;
    end process;
```

The 16 bit CRC calculation is performed in this section but not 5 bit CRC. 5 bit CRC is used for token packets only; but the transmitter doesn't send token packets. The 16-bit CRC is applied to data packets. CRC calculations are performed using multi-sectional shift registers that are fed into EXCLUSIVE-OR gates. If the present input is a 1, then the value of the CRC register is XORed with the polynomial 1100 000000000100 (CRC_{16}= X^{16}+X^{15}+X^{2}+1), which has binary bit pattern representation of 1000000000000101b (Where, X stands for base 2). If the input bit is a 0, then the value of the register only shifts left 1 bit. For the details on CRC, please see the references [2, 7, 11, and 13] provided in this document.

Based on the concept of CRC generator algorithm, the VHDL code is described. To calculate a 16-bit CRC, signals are declared for calculating, swapping and holding the result of the CRC calculation. These signals are **CRC16Out** which contains the result, **CRCState** which contains present state and **CRCNexeState** which contains next CRC state. These values are initialized to 0. The sensitivity list of the process which performs the CRC calculation contains **Clock**, **Reset** and **DataStuffed** signals. Part of the VHDL code for the process that performs this task is described as:

```
process(Clock, Reset, DataStuffed)
    begin

        if Clock'EVENT and Clock = '1' then
            CRC16Out <= NextCRCState;
            CRCState <= NextCRCState;
        end if;

        if Reset = '1' then
            CRCState <= "0000000000000000";
            CRC16Out <= "0000000000000000";
            NextCRCState <= "0000000000000000";
        else
            NextCRCState (15) <= CRCState (14) XOR DataStuffed XOR CRCState (15);
            NextCRCState (14) <= CRCState (13);
            NextCRCState (13) <= CRCState (12);
            NextCRCState (12) <= CRCState (11);
```

```
        NextCRCState (11) <= CRCState (10);
        NextCRCState (10) <= CRCState (9);
        NextCRCState (9) <= CRCState (8);
        NextCRCState (8) <= CRCState (7);
        NextCRCState (7) <= CRCState (6);
        NextCRCState (6) <= CRCState (5);
        NextCRCState (5) <= CRCState (4);
        NextCRCState (4) <= CRCState (3);
        NextCRCState (3) <= CRCState (2);
        NextCRCState (2) <= CRCState (1) XOR DataStuffed XOR CRCState (15);
          NextCRCState (1) <= CRCState (0);
          NextCRCState (0) <= DataStuffed XOR CRCState (15);
end if;
end process;
```

3.4.1.2 Multiplexer

One of the key components of the transmitter is the multiplexer. It is responsible for selecting the type of byte to be sent by the transmitter. The type of byte to be sent is specified by the controller. The sensitivity list of the process which performs the work of the multiplexer contains **InputSelect**, **Input** and **CRC16Out**. The output of the multiplexer is **MuxOutput**. Part of the VHDL code for the process that performs this task is written as:

```
process (InputSelect, Input, CRC16Out)

  begin

      case InputSelect is

        when "000" => MuxOutput <= PacketID;
        when "100" => MuxOutput <= CRC16Out(7 downto 0);
        when "101" => MuxOutput <= CRC16Out(15 downto 8);
        when "011" => MuxOutput <= Input;
        when "111" => MuxOutput <= "00000001";   ---Synchronization sequence
        when others =>MuxOutput <= Input;
      end case;
  end process;
```

3.4.1.3 Parallel to Serial Converter and Stuffed Bit Adder

The parallel to serial converter is a shift register in which data are loaded in to the register in parallel. Serial data is taken out from the last flip-flop as shown in Figure 3-6.

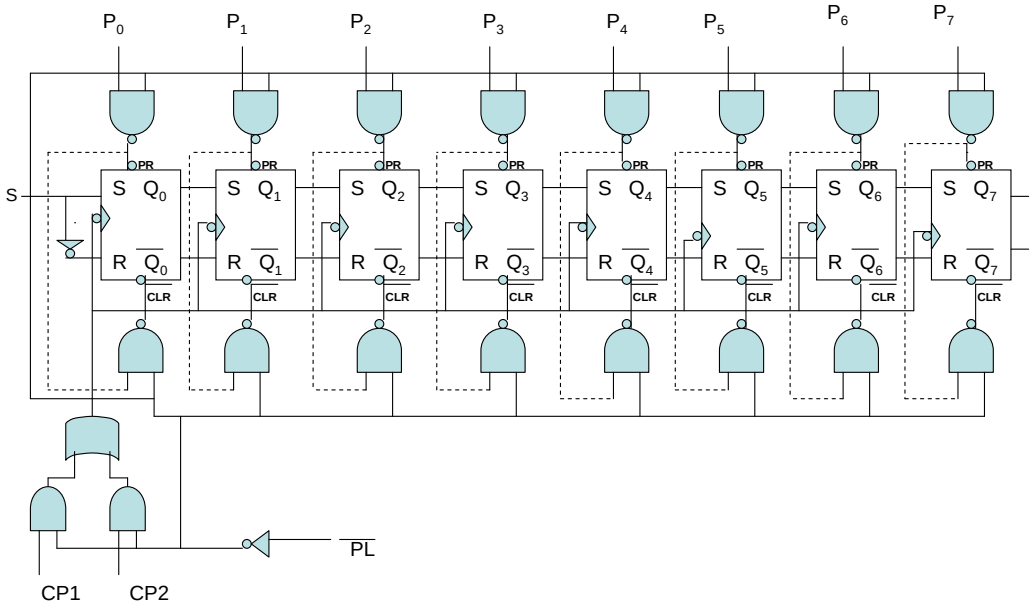

Fig. 3-6: 8 bit Parallel in Serial out shift register (logic diagram of 74165)

The concept of parallel to serial conversion is used to describe the VHDL code for this part of the transmitter. To write the VHDL code for parallel to serial conversion and Stuffed Bit adding, in addition to signal declarations, variables are also declared. The variables are used here as temporary storage of data and for computational purposes. This is because they can be manipulated several times before a run of the process that performs the task completes. But signals are updated only at the end of a process. Finally the values of the variables are assigned back to the signals at the end of the process so that they can be used the next time the process is run and by other processes.

The variables used are:

ShiftRegisterTemp [7:0], DataStuffedTemp, CountShiftTemp [0:8], and CountBitTemp [0:6]

After sending a byte, the transmitter latches the data on the output of the multiplexer. It performs this by making the process latch the data from input whenever **CountShiftTemp** (which counts the number of bits sent to the host) is 0 as the piece of VHDL code below shows. This indicates that no bits have been sent.

```
if CountShiftTemp=0 then
      ShiftRegisterTemp:=MuxOutput;
end if;
```

When the controller enables the transmitter, the next task will be counting the number of consecutive 1s. This is done for the purpose of adding a stuffed bit. Whenever a '1' is coming in consecutive order, the counter signal **CountBit** is incremented. If the stuffed data contains a '0', the value of bit **CountBit** is reset to 0 as shown in the VHDL code fragment below:

```
if DataStuffedTemp='1' then
      CountBitTemp:=CountBitTemp+1;
else
```

```
        CountBitTemp:=0;
end if;
```

The process keeps checking and counting the number of 1s until it gets six consecutive 1s. If there have been six consecutive 1s, then a stuffed bit is added. But, if there have not been six consecutive 1s, data is shifted right by one and the most significant bit of the latched register is sent first. The variable **CountShiftTemp**, which counts the number of bits sent, is only incremented when a valid bit, as compared to a stuffed bit, is sent as described in the following piece of VHDL code.

```
if TransmitterEnable='1' then;
   if CountBitTemp=6 then
         DataStuffedTemp:='0';
         CountBitTemp:=0;

   else
          DataStuffedTemp:=ShiftRegisterTemp(7);
          ShiftRegisterTemp:=(6 downto 0) & '1';
          CountShiftTemp:=CountShiftTemp+1;
end if;
```

If eight bytes have been sent, the transmitter has to indicate this to the controller and the programmable device. The transmitter sets the **ByteSentOut** signal to 1 for one clock cycle indicating that it has completed sending a byte. This happens when **CountShiftTemp=8**, otherwise, **ByteSentOut** is set to 0. During this condition (when **CountShiftTemp=8**), the transmitter must also latch the next output of the multiplexer to send on the bus. At this time, **CountShiftTemp**, is set to 0 as shown in the following piece of VHDL code.

```
If CountShiftTemp=8 then
         ByteSentOut<='1';
         CountShiftTemp:=0;
         ShiftRegisterTemp:=MuxOutput;
else
          ByteSentOut<='0';
end if;
end if;
```

3.4.1.4 NRZI Encoding

The data that is prepared to be sent by the transmitter, whether it is bit stuffed or not, must be NRZI encoded before being transmitted through USB bus. Every NRZI encoded bit depends on the previous encoded bit. The value of the first encoded bit depends on some initial value. In USB, idle state is defined to be logic high '1'. This means the USB packets are separated by streams of '1's. Because of this, the NRZI encoding of the first bit is based on the assumption that the previous bit before the start of reception of that byte was '1'. Let the previously encoded data has a last bit of 0 or 1 and the coming input is 0, then NRZI encoding scheme requires transition from 0 to 1 for the last bit 0 and transition from 1 to 0 for last bit 1. But if the input is 1, then the last bit is taken as it is without bit transition. Therefore, data bit stream 00000001 is NRZI encoded to 01010100, for instance, assuming idle state as 1.

Following this technique, to implement the NRZI encoding in VHDL, signals **DataStuffedTemp**, and **PreviousOutput** and Output are declared. The NRZI encoding technique considers both the previous data and the coming input data. In this technique, whenever the coming input data is a '1', then the previous data is taken as it is .However, if the coming input is a '0', then the inverted value of the previous data is taken as the piece of VHDL code below describes.

```
if DataStuffed='0' then
        PreviousOutput<=not(PreviousOutput);
        Output<=PreviousOutput;
end if;
```

Then the value of the Output forces the values of **Vpo** and **Vmo** to change reflecting its new values as explained in the next section.

3.4.1.5 Transceiver Interface

The transmitter has an interface to the transceiver at the end of its output section as shown in Figure 3-3. A transceiver interface is required to connect the USB system to a transceiver that converts the analog differential signal from the D+ and D- lines of the USB cable in to digital signals that can be used by the programmable device. The programmable device also uses the transceiver to convert digital signals to differential signals when sending information to the host. A functional diagram of the PDIUSBP11 transceiver from Phillips Semiconductors [12] is shown in Figure 3-7 and values and results of the vmo and vpo pins is listed in table 3-1.

Serial data is received from the host through the rcv signal. The conversion of the differential signal to a binary bit stream is accomplished through the use of comparator circuitry, which compares the D+ and D- lines to produce the signal outputted on rcv. When OE# (output enable) pin is active, then the transmitter can transmit data on the USB bus. When it is not, then the transceiver can receive data from the USB for the USB system. The signal pins Vmo and Vpo are used by the USB Interface to transmit data on the USB bus. The D- and D+ lines are connected to the vmo and vpo signal pins through the tri-state buffer. When OE# is active, the values from vmo and vpo can pass through the buffer. SE0 represents single ended zero (Vpo and Vmo are both zero). The Vp and Vm signals represent the gated versions of D+ and D- (Vp represents D+ and Vm represents D-) and are not required in this USB Interface design. The suspend input is active high and enables a low power state while the USB bus is inactive.

Table 3.1: VPO and VMO inputs values and results

Vpo	Vmo	RESULT
0	0	SEO
0	1	Logic '0'
1	0	Logic '1'
1	1	Undefined

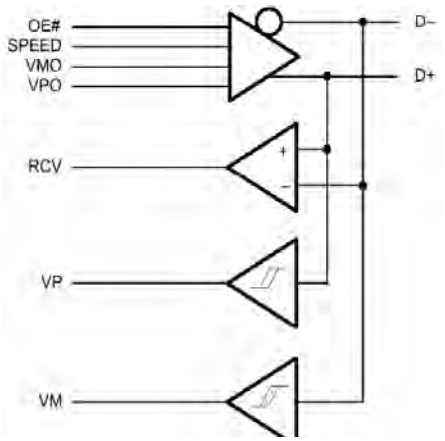

Figure 3-7: Transceiver functional diagram

VHDL implementation of this section is described in a process known as **TranInterface**. This process forms an interface for the USB Interface to the transceiver. The sensitivity list for this process is **Output** and **End_of_pkt_Send**. When the **End_of_pkt_Send** signal is activated, both **Vpo** and **Vmo** are set low. The controller then needs to use end of packet signal on the transmitter to send an EOP signal on the USB bus. The controller holds the EOP signal for two bit times to indicate an end of packet. This process also changes the values of **Vpo** and **Vmo** when the value of **Output** changes. When **Output** is 1, **Vpo** should be set to 1 and **Vmo** to 0. When **Output** is low, **Vpo** is set to 0 and **Vmo** to 1 as shown in the VHDL code fragment below.

```
TranInterface: process(Output, End_of_pkt_Send)

begin
    if End_of_pkt_Send='1' then
                Vpo<='0';
                Vmo<='0';
        else
         if Output='1' then
                  Vpo<='1';
                  Vmo<='0'
         else
                  Vpo<='0';
                  Vmo<='1';
  end if;
```

3.4.2 VHDL Implementation of the Receiver

The responsibility of the receiver is receiving the incoming data from the host .The receiver performs tasks including:

- Detecting the SOP
- NRZI Decoding
- Removing Stuffed Bits
- Serial to Parallel Conversion
- 16-bit and 5-bit CRC calculations
- Indicating to the controller about the type of transaction taking place

- Taking out data from the transaction and sending it to the programmable device.
- Indicating the programmable device about address and endpoint of the target.

The input and output ports of the receiver are specified as shown in figure 3-8. The purpose of all the signals used in this part is also listed. The step by step design process of the receiver and the algorithmic approach for the VHDL implementation of the design are described in the corresponding topics.

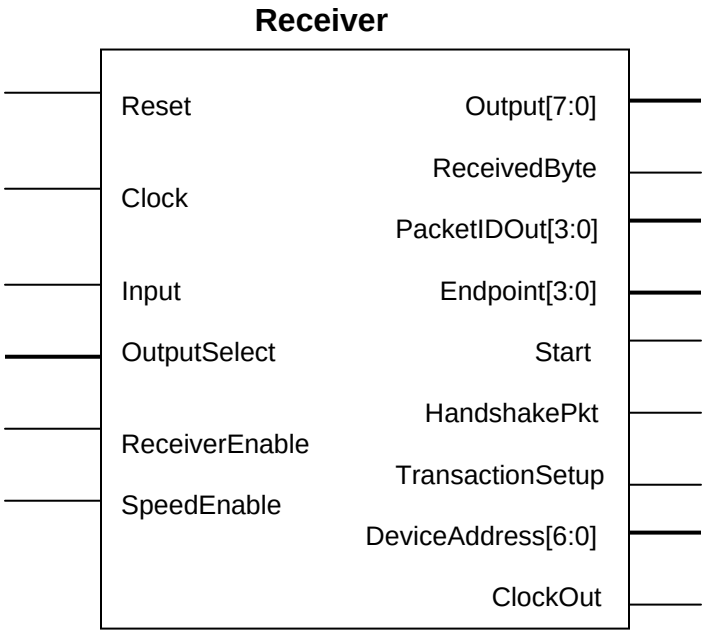

Figure 3-8: I/O ports of the receiver

Signals used for the design of the receiver and their purposes are listed below:

PreviousInput - used by NRZI decoder to hold the last input
DataStuffed - used to hold the stuffed input
CountBit - Counts number of 1s
Data [7:0] - used as a data register
CountShift - used to count number of bit shifted
PreviousPllInput - used to hold last pll input
UserClock - holds the input clock
PreviousOutputSignal - used as the last output signal
CountPll -used to count when to toggle the output clock signal
ByteOut [7:0] -used to indicate that a byte is received
PacketID [7:0] -used to hold packet identification
SOPState [7:0] - used as current start of packet state
NextSOPState [7:0] - used as next start of packet state
SOPStateType [7:0] - user defined type, used contain all the SOP states
NextCRCState [15:0] - used as the next 16-bit CRC states
CRCState [15:0] -used as the current 16-bit CRC state
CRC16Out [15:0] -used as 16-bit CRC register
NextCRC5State [4:0] -used as next 5-bit CRC state

CRCR5State [4:0] -used as current 5-bit CRC state
CRC5Out [4:0] -used as 5-bit CRC register

Variables declared and used in manipulation of the VHDL algorithm for the receiver are:

DataStuffedTemp, DataTemp [7:0], CountBitTemp, CountShiftTemp

3.4.2.1 SOP Detector

Start of packet detector is one of the key components of the receiver as shown in Figure 3-3. The signals from the rcv pin of the transceiver are fed in to the SOP detector and NRZI decoder. Whenever a start of packet has been detected, the SOP detector will indicate it to the controller. The SOP detector is basically a Mealy state machine as shown in Figure 3-9). The reason is that the output of the state machine is dependent on not only the current state but also on the present input as well.

The state diagram shown in Figure 3-9 can be translated easily to a high level VHDL description with out having to perform the traditional design methodology such as state assignment, generate the state transition table, or determine the next-state equations based on the types of flip flops available. In VHDL, each state can be translated to a case in a "case-when" construct. The state transition can then be specified in "if-then-else" statements. For example, to translate the state flow diagram of Figure 3-9 in to VHDL, we begin by defining an enumeration type, consisting of the state names and declaring signals of that type as:

```
type SOPStateType  is (S0,S1,S2,S3,S4,S5,S6,S7,S8);
signal SOPState,NextSOPState:SOPStateType;
```

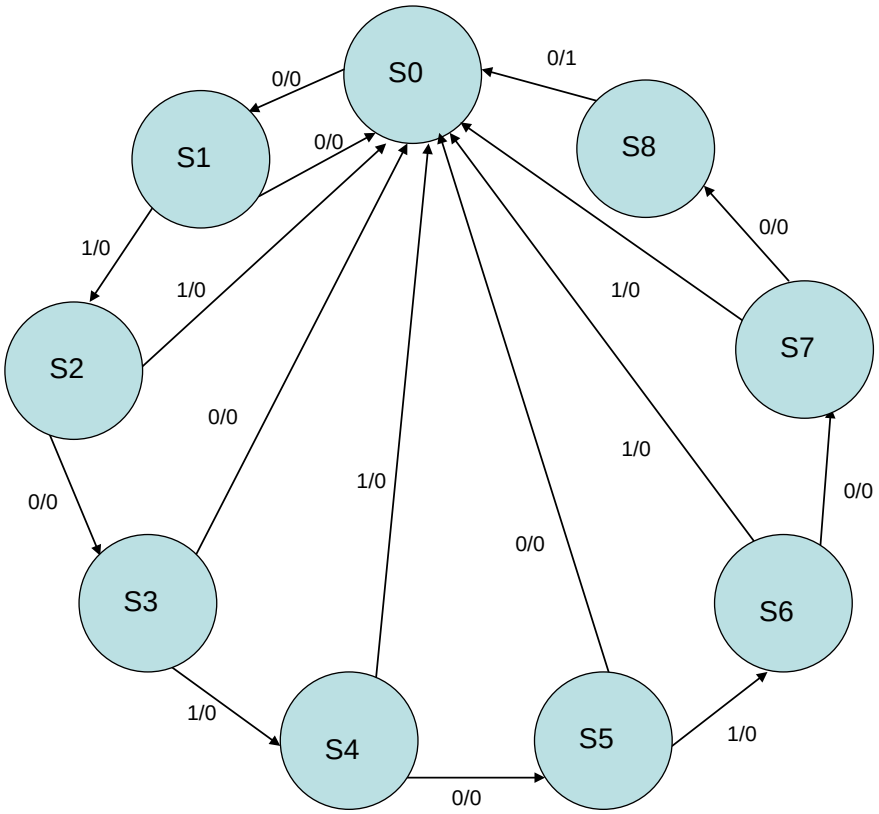

Figure 3-9: Mealy State machine for start of packet detector

The process in the receiver which detects the start of a packet is labeled **StartOfPacketDetector**. The SOP detector of the receiver seeks the initial encoded 8-bit NRZI sequence "01010100".This is the packet that must be sent before all packets are sent in the USB system for synchronization (see Figure 2-12 page 17).

Once a 0 is detected, it moves into state S1. If the next bit is a 1 it moves back to the start state S0. If the next bit were a 0 instead, it would move up a state to S1. If there were six consecutive 1's the controller would eventually reach state S7. If the next bit after S7 had been reached were a 1, the SOP detector process would move to state S8 on the next clock cycle sending the start signal high. This signal will remain high for one clock cycle. This indicates to the controller that a start of packet has been detected. The controller will then prepare the system to handle the rest of the transaction.

The following part of the VHDL code shows how the state variables and signals for SOP detector are declared and how the process **StartOfPacketDetector** performs SOP detection.

```
StartOfPacketDetector:   process(SOPState, Input)

    begin

        case SOPState is

            when S0=>if Input = '0' then NextSOPState <= S1;Start <= '0';
                    else NextSOPState <= S0;Start <= '0';end if;
            when S1=>if Input = '1' then NextSOPState <= S2;Start <= '0';
```

```
            else NextSOPState <= S0;Start <= '0';end if;
        when S2=>if Input = '0' then NextSOPState <= S3;Start <= '0';
            else NextSOPState <= S0;Start <= '0';end if;
        when S3=>if Input = '1' then NextSOPState <= S4;Start <= '0';

            else NextSOPState <= S0;Start <= '0';end if;
        when S4=>if Input = '0' then NextSOPState <= S5;Start <= '0';
            else NextSOPState <= S0;Start <= '0';end if;
        when S5=>if Input = '1' then NextSOPState <= S6;Start <= '0';
            else NextSOPState <= S0;Start <= '0';end if;
        when S6=>if Input = '0' then NextSOPState <= S7;Start <= '0';
            else NextSOPState <= S0;Start <= '0';end if;
        when S7=>if Input = '0' then NextSOPState <= S8;Start <= '0';
            else NextSOPState <= S0;Start <= '0';end if;
        when S8=>NextSOPState <= S0;Start <= '1';
    end case;
end process;
```

3.4.2.2 NRZI Decoder

NRZI Decoder is also one of the key components of the receiver as indicated in Figure 3-3. The receiver decodes the NRZI encoded data sent from the sender using the NRZI decoder. Signals from the rcv pin of the transceiver are also fed in to the NRZI decoder. The decoder must decode the data to obtain the required actual information. The NRZI decoder performs the inverse function of the NRZI encoder. It considers the last value of previously accepted input. This is recorded by the signal **PreviousInput**. If the input has changed from the last value, a '0' has been received. But, if the input is the same as the previous clock cycle, then the data received is a '1'. The decoded data is placed on a signal called **DataStuffed** so that it can be used by the next system of the receiver. Therefore, it performs the inverse of the encoder as the following piece of the VHDL code describes:

```
if(input=PreviousInput) then
    DataStuffedTemp:='1';
else
    DataStuffedTemp:='0';
end if;
```

3.4.2.3 Serial to Parallel Conversion and Bit Unstuffing

Serial to parallel converter and stuffed bit remover is part of the receiver section. The data the receiver receives is serial data. The NRZI decoded signal is fed to the serial to parallel converter. Here, serial to parallel conversion and bit unstuffing are performed. The bit stuffed by the sender must be removed by the receiver at this stage.

Data input serially and shifted to the right or left on every clock pulse transition. Information entered serially, by shifting, can be taken out in parallel (P0 to P7) as shown in Figure 3-10. The concept of serial to parallel conversion and stuffed bit removing is implemented in VHDL as described in the following section.

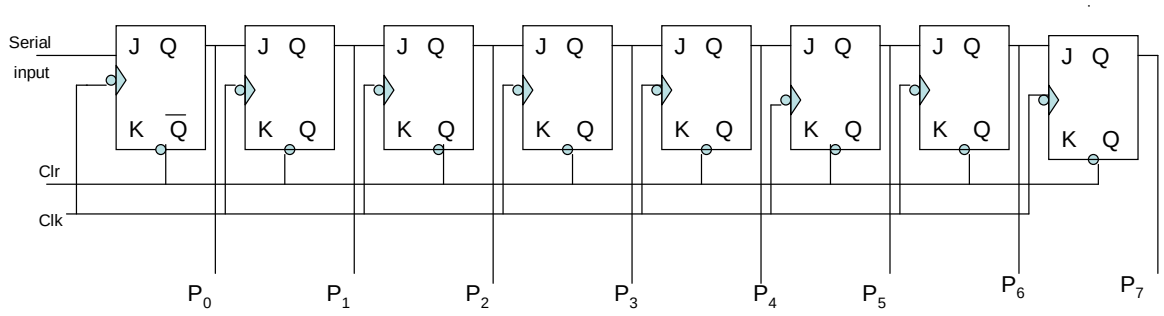

Fig. 3-10: Serial in Parallel out shift register (logic diagram of 74164)

For the purpose of removing stuffed bit, the number of 1s is counted. Whenever consecutive 1s are received, the counter is incremented; otherwise, the counter is set to 0 as described in the following VHDL code fragment:

```
if DataStuffedTemp='1' then
        CountBitTemp:=CountBitTemp+1;
else
        CountBitTem:=0;
end if;
```

If there have been six consecutive 1s, it is known that the next bit will be a stuffed bit. In this case, the next bit is ignored. Adding the input to the data register and shifting from parallel to serial is done only if it is not a stuffed as described in the following VHDL code fragment.

```
if not(CountBitTemp = 6) then
        DataTemp(7) := DataTemp(6);
        DataTemp(6) := DataTemp(5);
        DataTemp(5) := DataTemp(4);
        DataTemp(4) := DataTemp(3);
        DataTemp(3) := DataTemp(2);
        DataTemp(2) := DataTemp(1);
        DataTemp(1) := DataTemp(0);
        DataTemp(0) := DataStuffedTemp;
        CountShiftTemp := CountShiftTemp + 1;
else

        CountBitTemp := 0;
end if;
```

Once the data has been shifted, the variable **CountShift** is incremented. This register continues counting the number of shifted bits in order to know when to make the **ReceivedByte** signal go high. When this signal becomes high, it indicates to the controller and the programmable device that a byte has been received as described in the following VHDL code fragment:

```
If CountShiftTemp=8 then
        ReceivedByte<='1';
        ByteOut<=DataTemp;
        CountShiftTemp:=0;
else
      ReceivedByte<='0';
end if;
```

3.4.2.4 Demultiplexer

The demultiplexer section of the receiver is responsible to contain the out put of the serial to parallel converter. The demultiplexer performs selection of the byte it received if the controller specifies and instructs it which byte to be selected. The selection made by the controller is provided to the receiver through **OutputSelect** port. Figure 3-11 shows the output of the demultilpexer.

Figure 3-11: Demultiplexer outputs

Not all the fields of the demultiplexer are utilized by the receiver but are present to be consistent with the multiplexer of the transmitter. The packet ID field represents the type of packet being transmitted. This value is obtained from received packets and used by the PID checker process to indicate the controller about the type of packet being received. The "combination" consists of two bytes which represents the destination endpoint of the packet, address of the destination and CRC 5 value of the token packet. The endpoint value is passed to the programmable device through the port **Endpoint**. The process which performs the task of demultiplexer is described in the VHDL code fragment below:

```
process (OutputSelect, ByteOut)

    begin

        case OutputSelect is

            when "000" => PacketID <= ByteOut;
            when "001" =>
                    Endpoint(3) <= ByteOut(0);
                    DeviceAddress <= ByteOut(7 downto 1);
            when "010" =>
                    Endpoint(2 downto 0) <= ByteOut(7 downto 5);
            when others => Output <= ByteOut;
        end case;
end process;
```

3.4.2.5 Information Checkers

In the receiver section, the information checkers are responsible for checking the PID, comparing the CRC with that calculated by the receiver and informing the programmable device about the target endpoint.

3.4.2.5.1 PID Checker

The PID checker process informs the controller which type of packet has been received, including handshake packet. The following VHDL code fragment shows the ways how some of the PIDs are checked:

```
PacketIDChecker: process (Reset, PacketID)

    begin

        if Reset = '1' then
            HandshakePkt <= '0';
            PacketIDOut <= "0000";
            TransactionSetup <= '0';

        elsif PacketID = "10100101" then        ---Start of frame (SOF) transaction
            PacketIDOut <= "0000";
            HandshakePkt <= '0';

        elsif PacketID = "10110100" then        ---SETUP Transaction
            PacketIDOut <= "0001";
            HandshakePkt <= '0';
            TransactionSetup <= '1';

        elsif PacketID = "10000111" then        ---OUT  Transaction
            --EnableReceiver <= '1';
            PacketIDOut <= "0010";
            HandshakePkt <= '0';

        elsif PacketID = "10010110" then        ---IN  Transaction
            PacketIDOut <= "0011";
            HandshakePkt <= '0';

        elsif PacketID = "11000011" then        ---DATA0 transaction
            PacketIDOut <= "0100";
            HandshakePkt <= '0';
        elsif PacketID = "11010010" then        ---DATA1 transaction
            PacketIDOut <= "0101";
            HandshakePkt <= '0';

        elsif PacketID = "01001011" then        ---ACKNOWLEDGE MENT
            PacketIDOut <= "0110";
            HandshakePkt <= '1';

        elsif PacketID = "01011010" then        ---NEGATIVE ACKNOWLEDGEMENT
            PacketIDOut <= "0111";
            HandshakePkt <= '1';

        elsif PacketID = "01111000" then        ---STALL
            PacketIDOut <= "1000";
            HandshakePkt <= '1';
        end if;
    end process;
```

3.4.2.5.2 5-bit CRC Calculation

The receiver, as stated before, performs 16- bit and 5-bit CRC check. Here, only 5-bit CRC check is explained. If the calculated CRC matches that of sent with the packet, the receiver knows that the packet has been received correctly. If the input to the CRC calculator is 1, then the current value of the CRC register is XORed with the polynomial X^5+X^2+1 before being shifted left [2]. The bit pattern representing this polynomial is 00101b Otherwise; the CRC register is shifted left without being XORed.

The 5-bit CRC calculator process calculates the CRC for incoming bits as described in the VHDL code fragment below:

```
process(Clock)

    begin

        if Clock'EVENT and Clock = '1' then
                CRC5Out <= NextCRC5State;
                CRC5State <= NextCRC5State;
        end if;

        if Reset = '1' then
                CRC5State <= "00000";
                CRC5Out <= "00000";
                NextCRC5State <= "00000";
        else
                NextCRC5State(4) <= CRC5State(3);
                NextCRC5State(3) <= CRC5State(2);
                NextCRC5State(2) <= CRC5State(1) XOR DataStuffed XOR CRC5State(4);
                NextCRC5State(1) <= CRC5State(0);
                NextCRC5State(0) <= DataStuffed XOR CRC5State(4);
        end if;
        end process;
```

3.4.3 VHDL Implementation of the Controller

The controller is responsible to manage every activity of the USB transactions. It controls the data paths of the transmitter and the receiver.

The input and output ports of the controller are specified as shown in figure 3-12. The purpose of all the signals used in this part is also listed. The step by step design process of the controller and algorithmic approach for the VHDL implementation of the design are described accordingly.

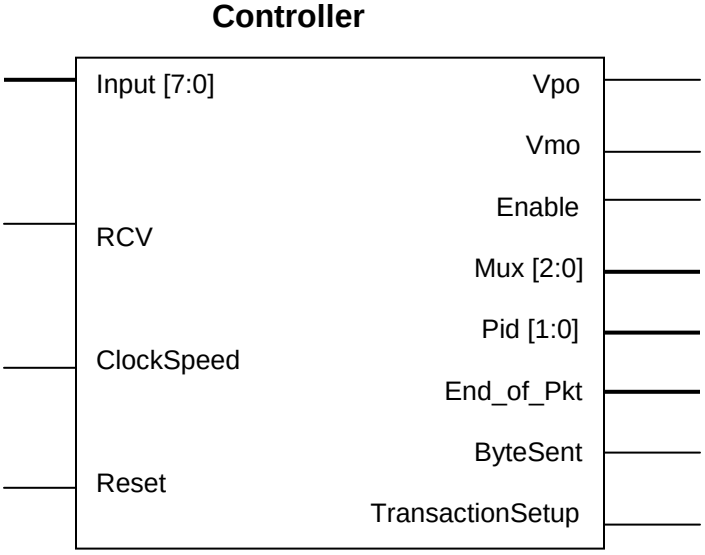

Figure 3-12: I/O ports of the controller

Note that the signals **Mux[2:0]**, **Pid[1:0]**, **End_of_Pkt**, **ByteSent**, and **TransactionSetup** are inout ports, meaning that they are bidirectional signals which allow for internal feedback.

Signals used for the design of the controller and their purposes are listed below:

SpeedEnable -used to indicate the receiver to start receiving
MuxSelect [2:0] -used to provide condition for moving from one state to another
EnableTransmitter -condition for enabling or disabling the transmitter
EnableReceiver -condition for enabling or disabling the receiver
PacketIDToSend [1:0] -specifies type of PID to send
End_of_pkt -indicates end of packet
ByteOut -indicates that byte is sent
ReceivedByte -indicates that byte is received
TransmiterMuxSelect[2:0] -specifies type of byte to be selected by multiplexer/demultiplexer
TransactionType [3:0] -condition for indicating type of transaction
ShiftRegister1 [9:0] -contains all the states of the controller
Resume -condition to indicate the start of controller
HandshakePkt -used for indicating handshake packets
PacketIDType [3:0] -provides the type of Packet ID
Clock -clock signal

Variable declared is: **EndCount**

Constants declared for holding values that can't be changed in the code are:

```
TransactionIn [3:0]:="0011";
TransactionOut [3:0]:="0010";
TransactionSOF[3:0]:=0000";
```

The controller is basically organized as a Mealy state machine as shown in figure 3-13. The state machine of the controller is only enabled when a byte is sent or received by the transmitter or receiver data paths. As shown in Figure 3-13, the first state of the controller is the **start** state. The next three states are used to receive the token packet from the host. Then the next four states divide the data packet or make a data packet based on the type of transaction. The last two states are used to either send or receive a token based on the type of transaction.

The state diagram shown in Figure 3-13 can be translated easily to a high level VHDL description with out having to perform the traditional design methodology. In VHDL, each state can be translated to a case in a "case-when" construct. The state transition can then be specified in "if-then-else" statements. To translate the state flow diagram shown in Figure 3-13 in to VHDL, we begin by defining an enumeration type, consisting of the state names and declaring signals of that type as:

```
type ShiftRegister1Type is (Start, PacketIDToken, FirstByte, SecondByte, SyncronData, PacketIDData, CRCData,
PacketIDHsk, Data, SyncronHsk);

signal ShiftRegister1:ShiftRegister1Type;
```

Therefore, the signal **ShiftRegister1** contains all the states of the controller and is used in the VHDL code in different algorithms and processes as required.

State machine for the controller

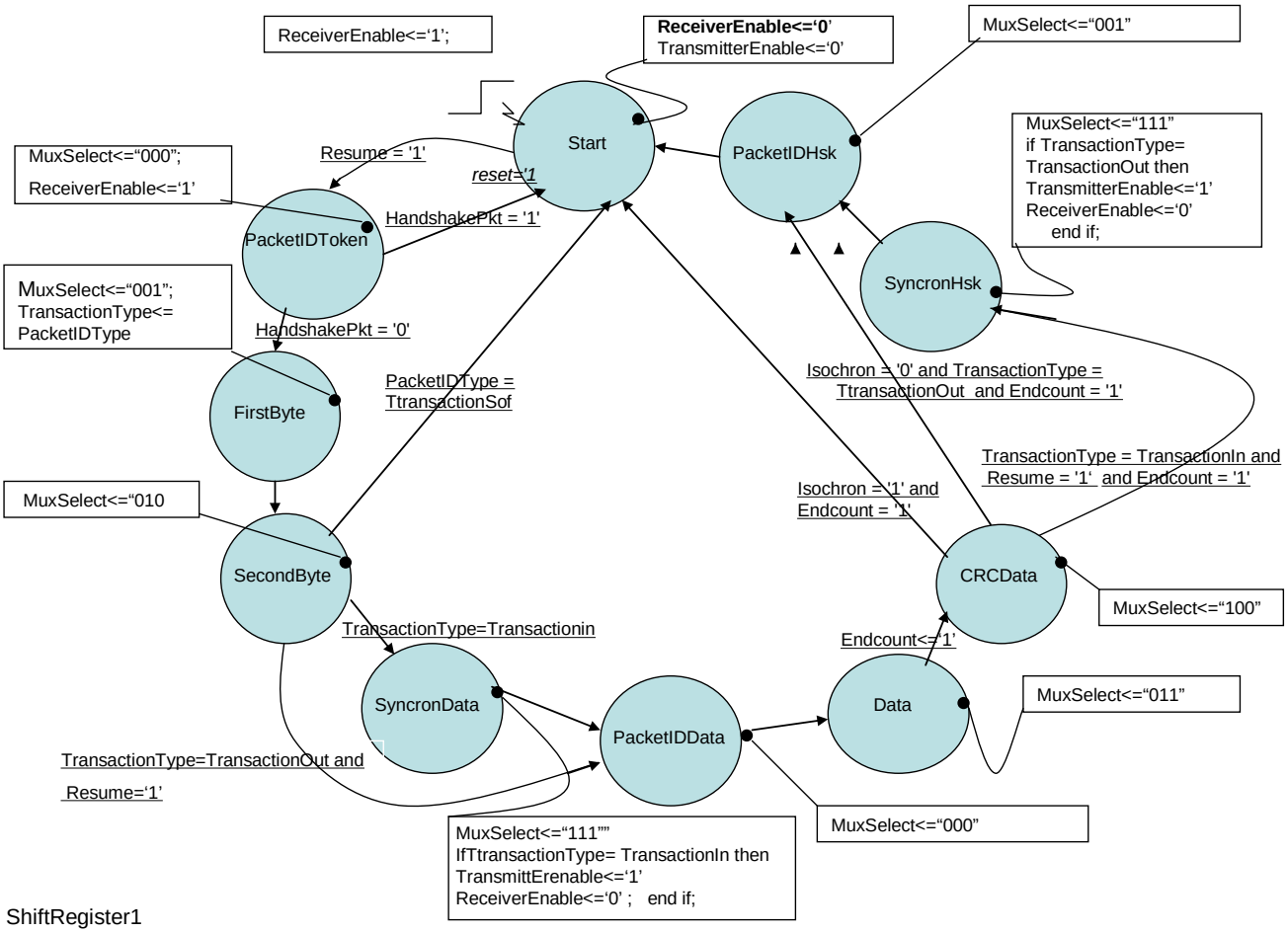

Fig. 3-13: Mealy state machine for the controller

The controller is indicated about packet arrival when the receiver detects a packet. When this occurs, the process called **SpeedEnable**, enables the receiver. Then the receiver will be ready to accept the next incoming data. The piece of VHDL code for this process is described as:

```
SpeedEnable: process(Resume)

    begin

        if Resume = '1' and ShiftRegister1 = Start then
                SpeedEnable <= '1';
        else
                SpeedEnable <= '0';
        end if;
    end process;
```

The controller then goes to the next state, **PacketIDToken** state, letting the receiver to decode the **PacketID** of the incoming token packet. After the receiver received the **PacketID**, the

controller then goes to **FirstByte** and **SecondByte** states indicating the receiver to receive the endpoint and address of the target device. In addition, the receiver is also ordered to check that the calculated CRC is matched with the received one with the packet once it is in the **SecondByte** state. If there is a match, the transmitter is ordered by the controller to send ACK signal after the controller is reached to the corresponding state. Or, NAK signal is sent if no match.

If the type of transaction is **TransactionIn,** then the controller goes to **SyncronData** state and the transmitter is indicated to transmit the synchronization sequence by the controller. If the transaction type is OUT or SETUP, the **SyncronData** state is not executed (no need to send synchronization sequence for them). Therefore, the controller goes to the **PacketIDData** state for the transaction of **PacketID**. Next state is the **data** state where transaction of data byte is performed. Then the controller sends or receives handshake packet based on the transaction type, after sending the data packet.

The process called **StateControl** contains the piece of VHDL codes that describes how the controller moves from one state to another for its controlling activities. It performs this task based on different conditions and the type of transaction as shown below:

```
StateControl: process (Clock, Reset)

    variable EndCount: std_logic;
        begin

            if Reset='1' then
                ShiftRegister1 <= Start;

                 ---default values setting
                EndCount := '1';
                EnableReceiver <= '0';
                EnableTransmitter <= '0';
                Enable <= '0';
                PacketIDToSend <= "00";
                TransmitterMuxSelect <= "111";
            elsif Clock'event and Clock = '1' then
            if ReceivedByte = '1' OR Resume = '1' OR ByteSentOut = '1' then
                EndCount := '1';

---working of the controller, how it moves from state to state based on different conditions and type of transaction

                    case ShiftRegister1 is
                        when Start =>
                            EnableReceiver <= '0';
                            EnableTransmitter <= '0';
                        if Resume = '1' then
                            ShiftRegister1 <= PacketIDToken;
                            EnableReceiver <= '1';
                        end if;
                        when PacketIDToken =>
                        if HandshakePkt = '1' then
                            ShiftRegister1 <= Start;
                        elsif HandshakePkt = '0' then
                            ShiftRegister1 <= FirstByte;
                        end if;
                        when FirstByte =>
                           TransactionType <= PacketIDType;
                           ShiftRegister1 <= SecondByte;
                        when SecondByte =>
```

```
 if PacketIDType = TransactionSOF then
         ShiftRegister1 <= Start;
 elsif TransactionType = TransactionOut and Resume = '1' then
         ShiftRegister1 <= PacketIDData;
 elsif TransactionType = TransactionIn then

         ShiftRegister1 <= SyncronData;
         TransmitterMuxSelect <= "111";
         EnableReceiver <= '0';
         EnableTransmitter <= '1';
 end if;
 when SyncronData =>
 if TransactionType <= TransactionIn then
        EnableReceiver <= '0';
        EnableTransmitter <= '1';
 end if;
 if ByteSentOut = '1' then
        ShiftRegister1 <= PacketIDData;
 end if;
        TransmitterMuxSelect <= "000";
        PacketIDToSend <= "11";
 when PacketIDData =>
 if TransactionType = TransactionOut then
        ShiftRegister1 <= Data;
 elsif ByteSentOut = '1' then
        ShiftRegister1 <= Data;
 end if;
        TransmitterMuxSelect <= "011";
 when CRCData =>
 if Isochron = '1' and EndCount = '1' then
        ShiftRegister1 <= Start;
 elsif TransactionType = TransactionIn and Resume = '1'
 and EndCount = '1' then
        ShiftRegister1 <= PacketIDHsk;
 elsif Isochron = '0' and TransactionType = TransactionOut
 and EndCount = '1' then
        ShiftRegister1 <= SyncronHsk;
        EnableTransmitter <= '1';
        EnableReceiver <= '0';
        TransmitterMuxSelect <= "111";
 end if;
 when PacketIDHsk =>
 if ByteSentOut = '1' then
       ShiftRegister1 <= Start;
 end if;
 when Data =>
 if EndCount = '1' then
 if TransactionType = TransactionOut then
       ShiftRegister1<= CRCData;
 elsif (TransactionType = TransactionIn) and (ByteSentOut = '1') then
      ShiftRegister1 <= CRCData;
 end if;
 end if;
 when SyncronHsk =>
 if TransactionType = TransactionOut then
      EnableTransmitter<= '1';
      EnableReceiver <= '0';
      TransmitterMuxSelect <= "000";
 end if;
 if ByteSentOut = '1' then
     ShiftRegister1 <= PacketIDHsk;
end if;
when others =>
 null;
```

```
            end case;
            end if;
            end if;
end process;
```

Depending on the state where the controller is in, it provides the selection values of the multiplexer and demultiplexer for transmitter and receiver respectively. As an example, when the controller is in the **syncrondata** state, the transmitter's multiplexer is set to select the sync sequence. The sync sequence is sent to the host controller ensuring that a start of packet. During transmission of data to the host controller, the transceiver pin is activated to low. The following VHDL code labled as **MuxSelectAssignment** shows how the controller provides the select values of the multiplexer and demultiplexer.

```
MuxSelectAssignment:
    MuxSelect  <= "000" when (ShiftRegister1 = PacketIDToken) else
            "001" when (ShiftRegister1 = FirstByte) else
            "010" when (ShiftRegister1 = SecondByte) else
            "111" when (ShiftRegister1 = SyncronData) else
            "000" when (ShiftRegister1 = PacketIDData) else
            "100" when (ShiftRegister1 = CRCData) else
            "000" when (ShiftRegister1 = PacketIDHsk) else
            "011" when (ShiftRegister1 = Data) else
            "111" when (ShiftRegister1 = SyncronHsk) else
            "000";
```

Chapter 4

Results of the thesis work

In this chapter, the outcomes of the thesis such as simulation results, synthesis results, and implementation results of the design will be discussed. The Xilinx ISE software has capability of design simulation, synthesis, fitting, place and route or implementation of the design on the target FPGA. It also has capability of downloading the design to program the FPGA as many times as required.

Before the three main components of the device under design have been integrated in to one system, they were simulated and synthesized separately. Then the controller has been made to incorporate the transmitter and the receiver inside its architecture.

4.1 Simulation Results

To verify that the USB Interface design functions as it is expected, behavioral simulation test has been done. A test bench waveform containing input stimulus has been created for verifying the functionality of the system. The simulator interpreted the VHDL code into circuit functionality and displayed logical results of the described HDL determining correct circuit operation as will be discussed next. The source code can be found in the Appendix section.

The simulation test was done for IN transaction and also for OUT transaction as explained just accordingly.

4.1.1 Waveform for IN transaction

Waveform for an IN transaction is generated by using a test bench VHDL module. Figure 4.1 contains the waveform for IN transaction which is explained as follows:

Data is sent using **vpo** to the transceiver. **vmo** is used to generate a differential signal for the transceiver i.e., it is inverted **vpo**.

The transmitter sends out a sync sequence while the controller is in **syncrondata** state. The synchronization sequence, which is sent before any packet, contains the value 00000001 (see section 2.1.2.4 on page 17). This data is NRZI encoded to 01010100. This NRZI encoded bit stream (01010100) matches with the waveform result at the output of **Vpo** corresponding to **syncrondata** state as shown in Figure 4-1 (waveform between the dashed lines as indicated by arrow line at the **Vpo** output in **syncrondata** state), verifying that the transmitter has NRZI encoded and sent the sync sequence through its output signal, **Vpo** correctly.
.
Once the sync sequence has been sent, the controller then moves to the **packetiddata** state. The data packet sent out by the transmitter must have PID that shows what type of packet it is. In the USB protocol it has the value 11000011 (see Figure 2-9 on page 15; and the VHDL code fragment under section 3.4.1.1 "Information Generator" on page 28). It is selected by the multiplexer after being directed by the controller. While the controller is in **packetiddata** state, the transmitter sends out the DATA0 PID which is 11000011. This value is NRZI encoded to 00101000 (note that the previous bit stream is sync sequence and has '0' at last). Now, in

Figure 4-1, if the waveform result at the output of **Vpo** corresponding to the **packetiddata** state is observed, the NRZI encoded DATA0 PID (00101000) bit stream is seen to be matched with it. This verifies that the transmitter has generated the packet ID and NRZI encoded it correctly.

The actual data used for testing the IN transaction is 01000101 which is NRZI encoded to 11010011. If the waveform result at the output of **Vpo** corresponding to the **data** state in Figure 4-1 is observed, it could be verified that the NRZI encoded input (11010011) bit stream matches with the values of **Vpo**. This verifies that the transmitter has correctly NRZI encoded the given input and transferred that NRZI encoded data to the out put pin, **Vpo**, which is then sent out through the USB bus.

From the results described above, it can be seen that the USB Interface could correctly performed the IN transaction.

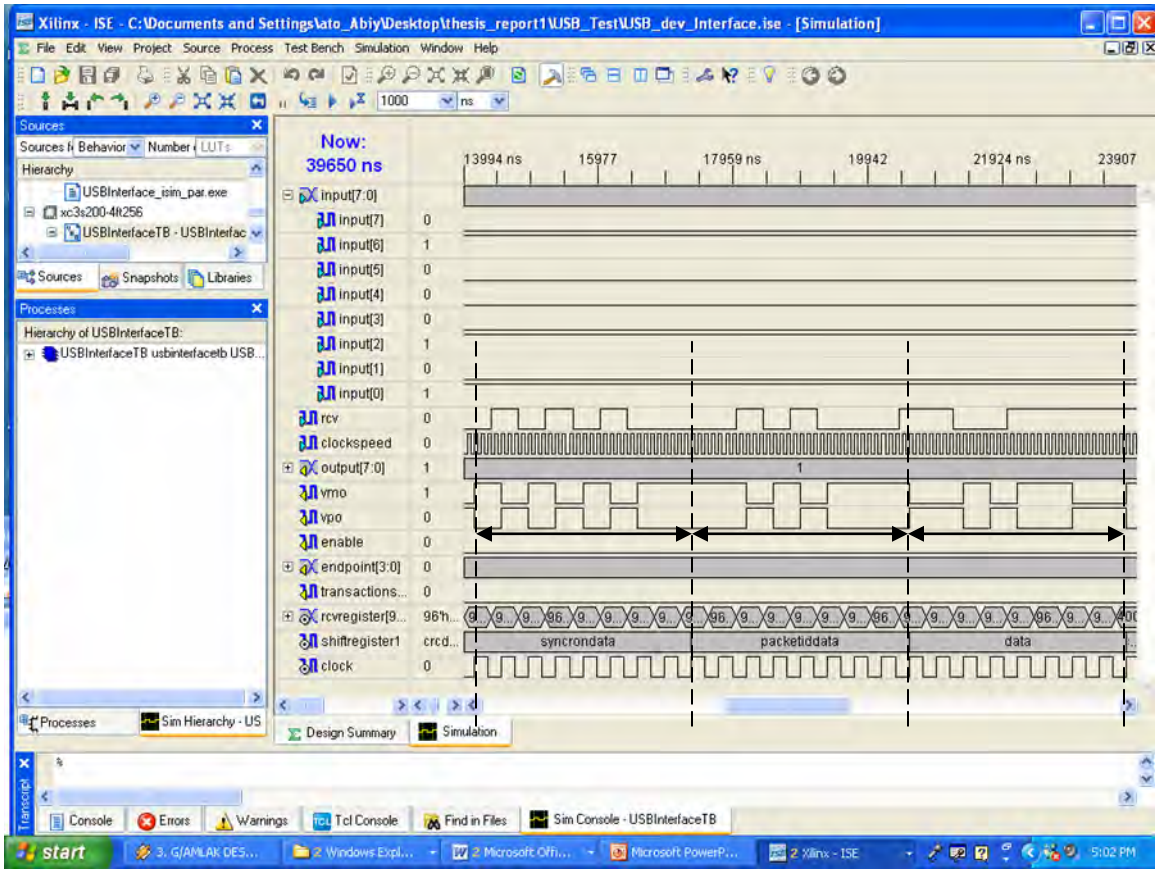

Figure 4-1: Waveform generated for an IN transaction

4.1.2 Waveform for OUT transaction:

For testing the OUT transaction, NRZI encoded bit streams of:

sync sequence (01010100), PID for OUT transaction (01010000), PID for data (00101000), and actual NRZI encoded data (11101011) are used. NRZI decoded bit stream of this data (11101011) is 01100001. In Figure 4-2, at the end of the **data** stage, the waveform output

values (indicated by dashed line and the bidirectional vertical arrow line) match with the bit stream (01100001). Such a result indicates that the data payload has been received correctly.

Further, it should be noted that after the USB Interface has received the data, it has to send out an acknowledgement handshake signal to the sender. To do this, it must send the synchronization sequence first and then the acknowledge signal next. The **syncronhsk** is the state in which the controller sends the SOP sequence for this case. This sequence is 00000001. Which is NRZI encoded to 01010100. In Figure 4-2, at **syncronhsk** state, the corresponding waveform result of **Vpo** shows that the two results are the same. After this sequence has been sent, packet ID (PID) for the acknowledge handshake must be sent. From the USB protocol, packet ID for ACK is 01001011 (see Figure 2-10, ACK PID on page 16 under section Handshake Packets); which is NRZI encoded to 11011000. Now if we observe the waveform result on the **Vpo** output, at the corresponding **packetIDhsk** state from Figure 4-2, it matches with the bit stream (11011000) verifying that the device is correctly sending the Acknowledgement packet ID ensuring that the data is correctly received and the device is acknowledging the sender for packet reception.

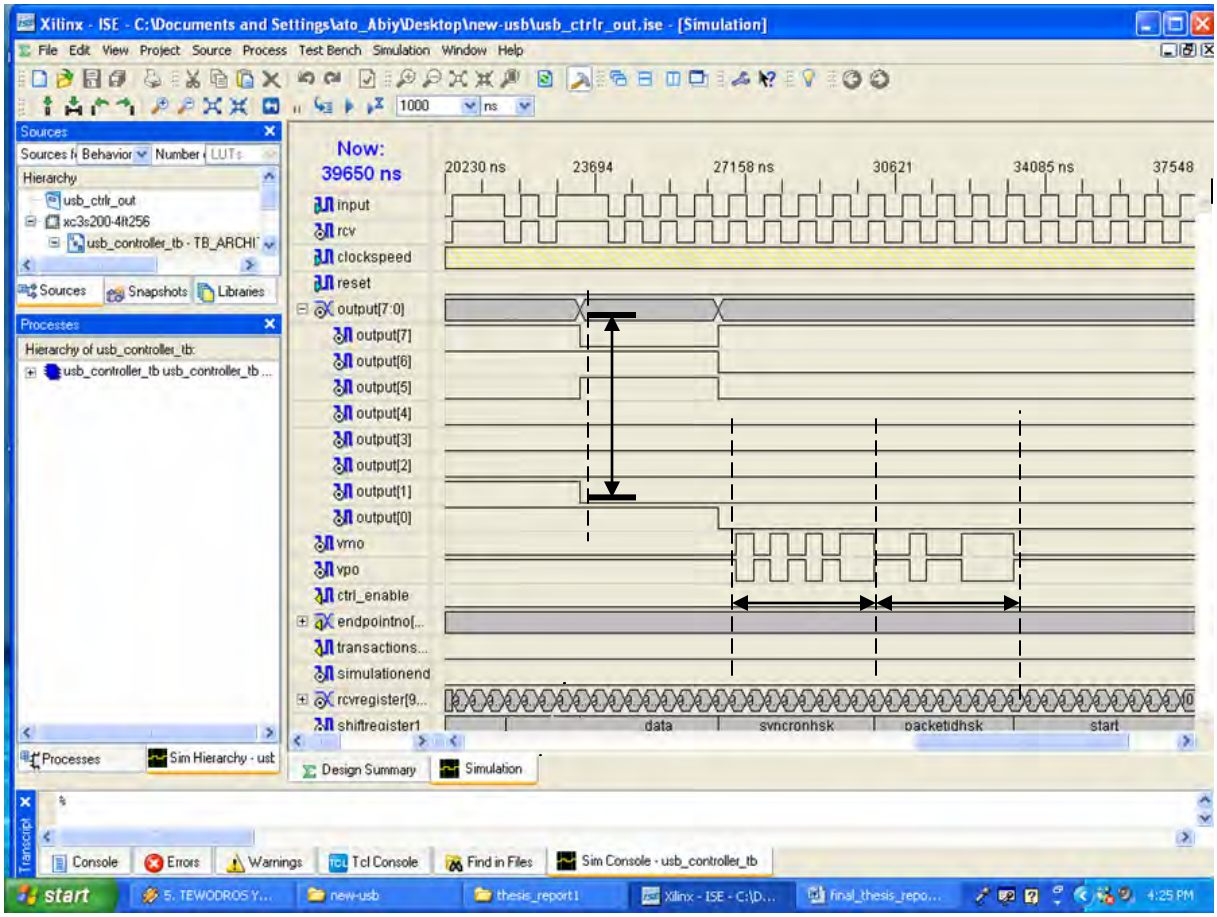

Figure 4-2: Waveform generated for an OUT transaction

In all these activities, the controller has been moved from one state to another depending on the type of the transactions. It has enabled the transmitter and the receiver high and low for data transmission and reception respectively. It has made the transmitter to send start of

packet and handshake signal once the receiver performed the reception; hereby carrying out its synchronizing and controlling activities.

From the simulation results, it can be observed that the USB Interface design performs as intended.

4.2 Synthesis and Implementation of the Design

4.2.1 Synthesis results

After design entry and simulation, synthesis has been run. The synthesis process has converted the VHDL description into a set of primitives or components that have been assembled in the target FPGA. Figures 4-3, 4-5 and 4-7 show the RTL views, Figures 4-4, 4-6 and 4-8 show the schematic RTL views of the transmitter, receiver and the top module, the controller respectively; and Figures 4-9 and 4-10 show the Technology Level schematic view of the synthesis process of the top module.

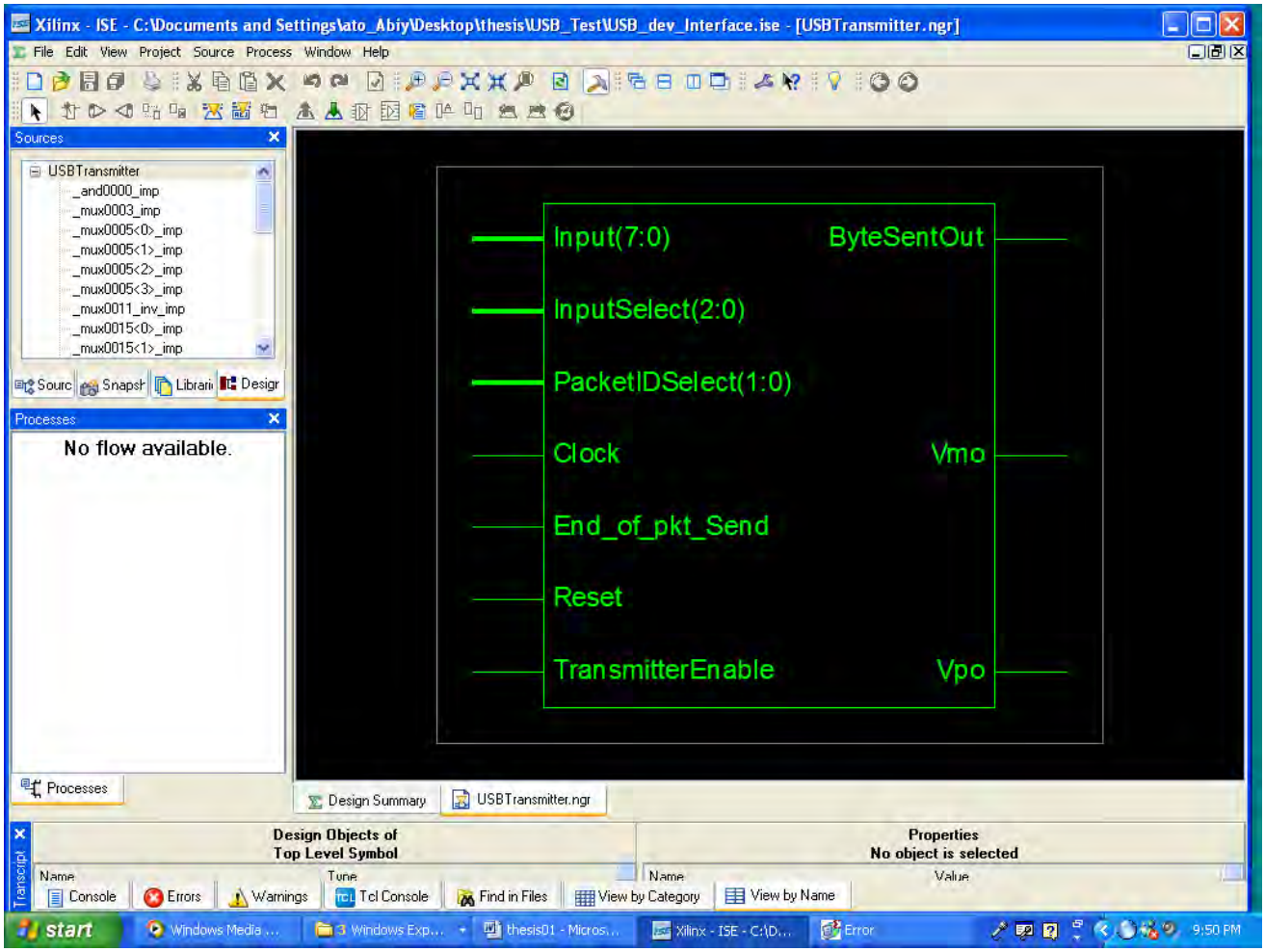

Figure 4-3: RTL View of the transmitter

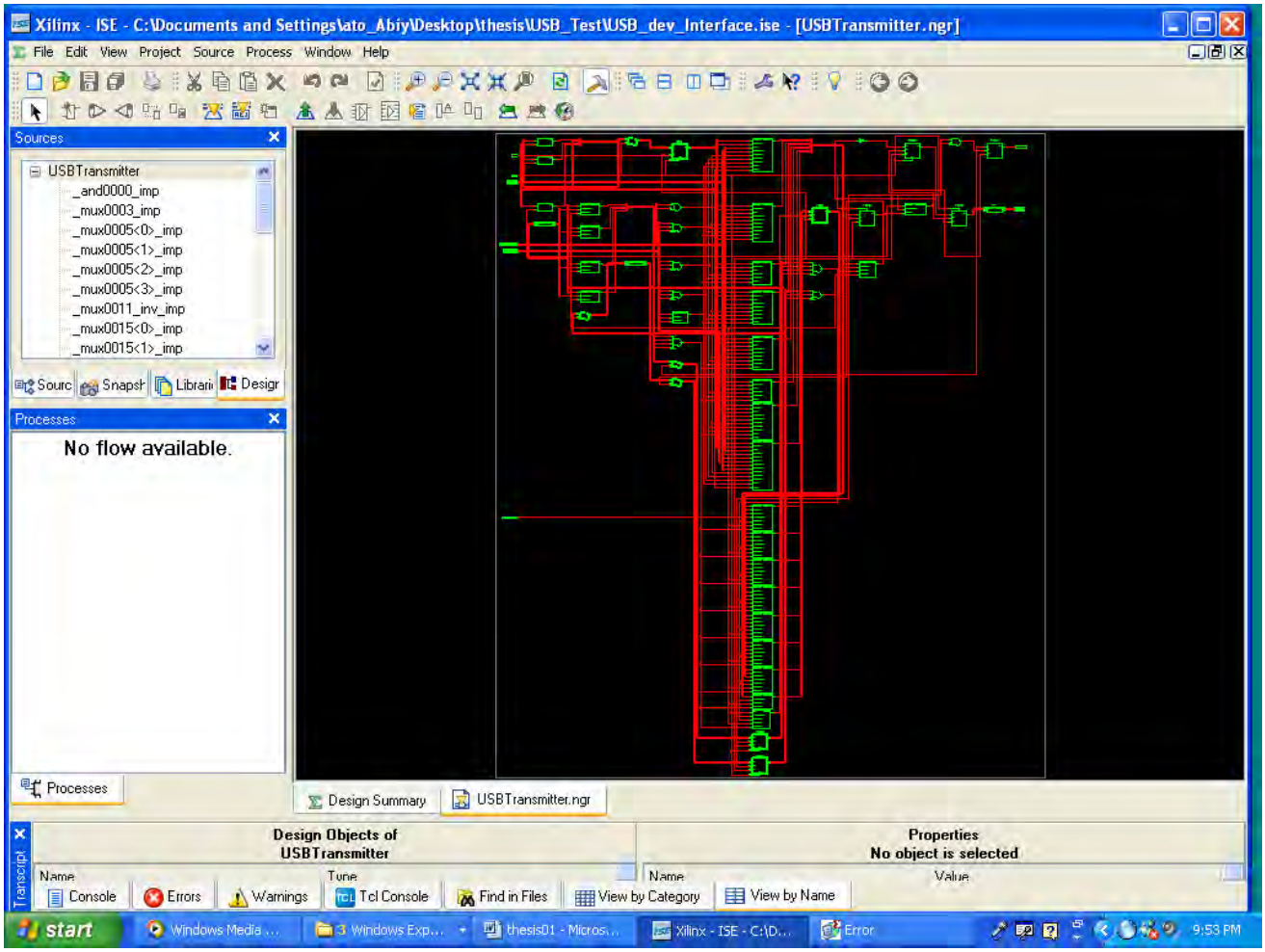

Figure 4-4: RTL Schematic view of the transmitter

After the HDL synthesis phase of the synthesis process, the RTL schematic representation of the synthesized VHDL source file is displayed as shown in Figures 4-4, 4-6 and 4-8. These schematic views show a representation of the pre-optimized design in terms of generic symbols, such as AND and OR gates. Viewing this schematic enabled to see a gate-level representation of the VHDL and helped to discover design issues early in the design process. Viewing the netlist in a graphical format enables to analyze how the synthesis tool inferred components in the design. This helped to identify problems and improve the design early in the design process.

The synthesis process has converted the design to internal data structures, translating its "behavior" to a register-transfer level (RTL) description. RTL descriptions specified registers, signal inputs, signal outputs, and the combinational logic between them. Depending on the specific features of the device architecture, the combinational logic is still represented by internal data structures.

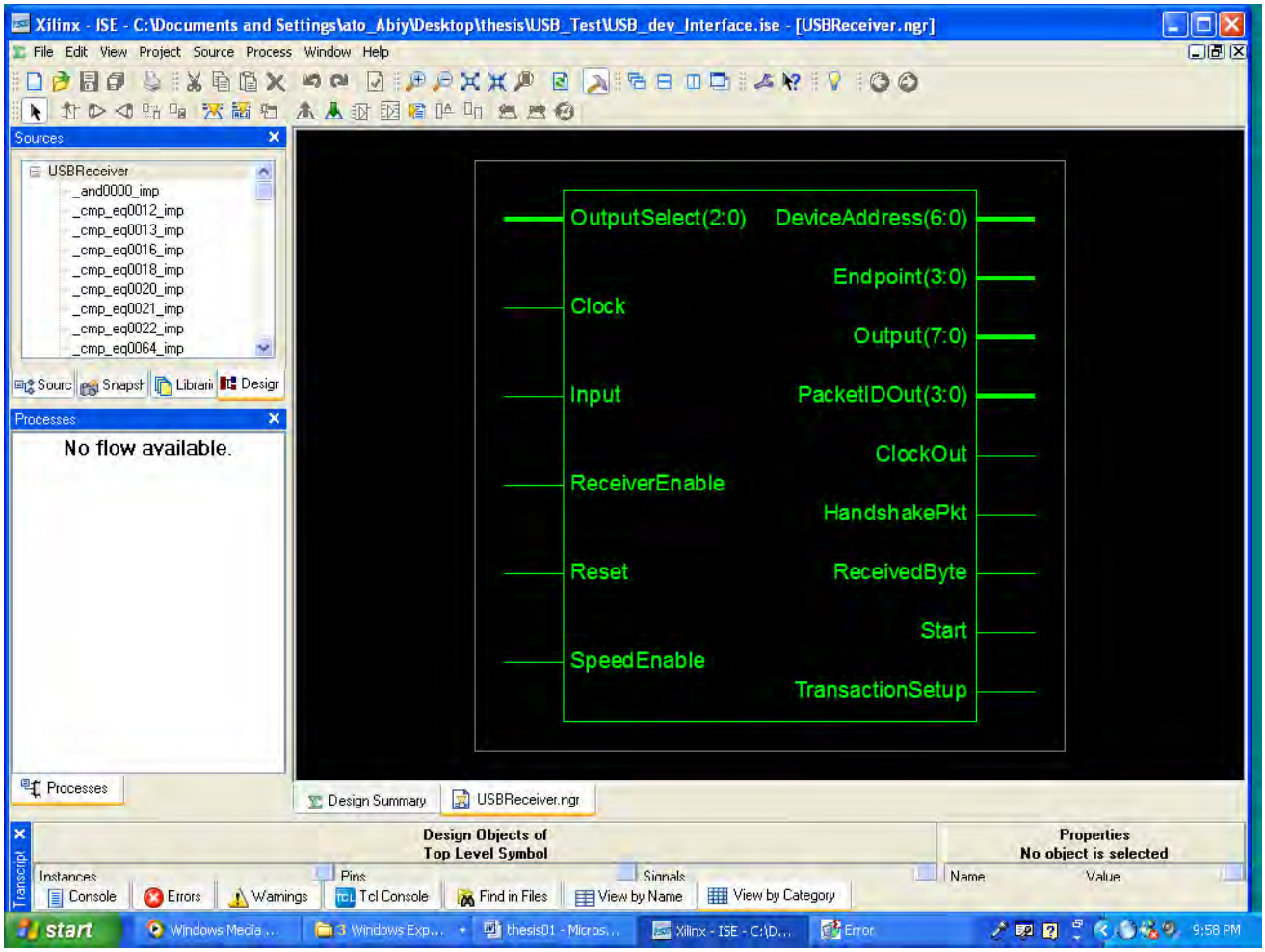

Figure 4-5: RTL View of the Receiver

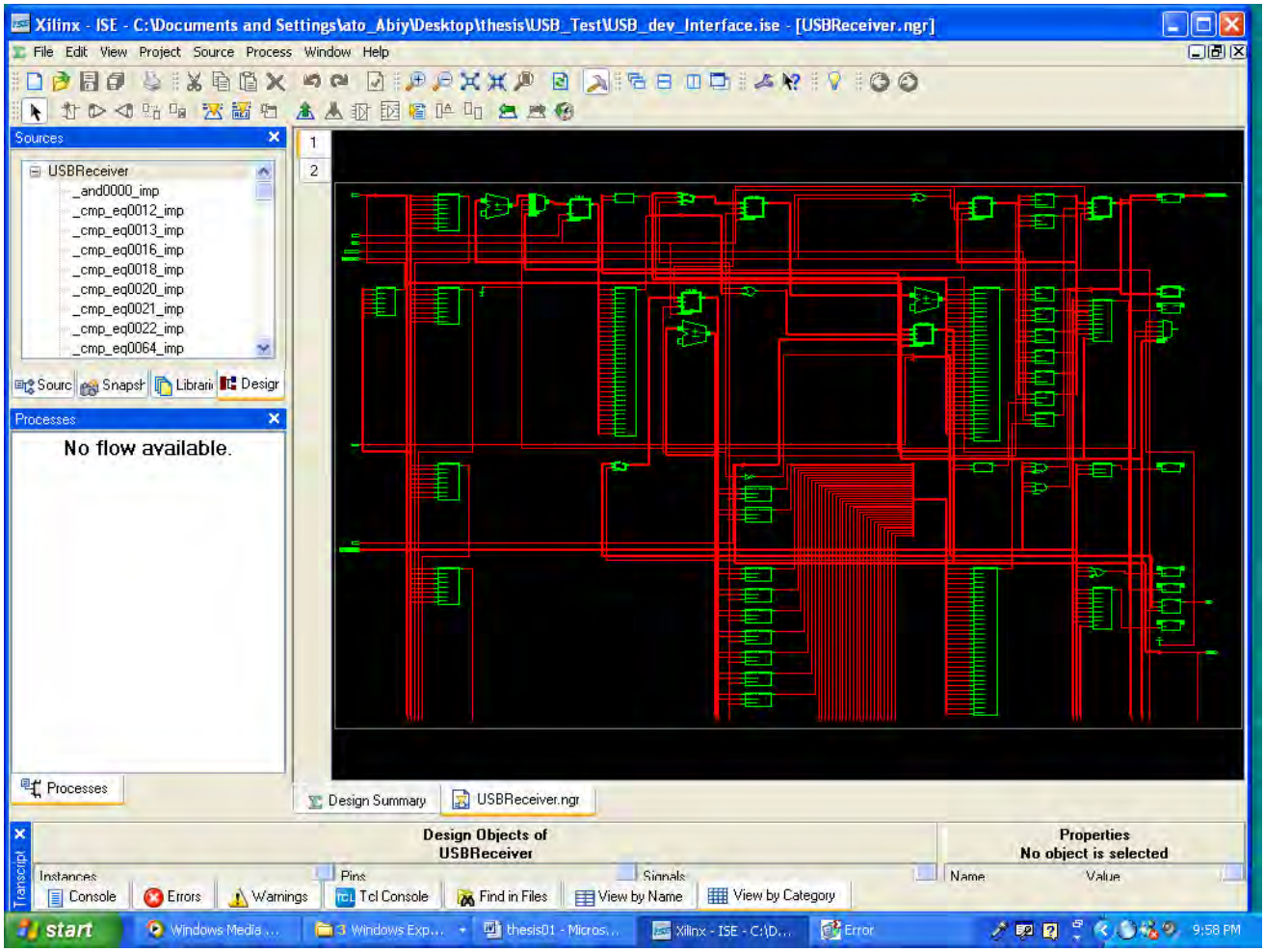

Figure 4-6: RTL Schematic view of the receiver

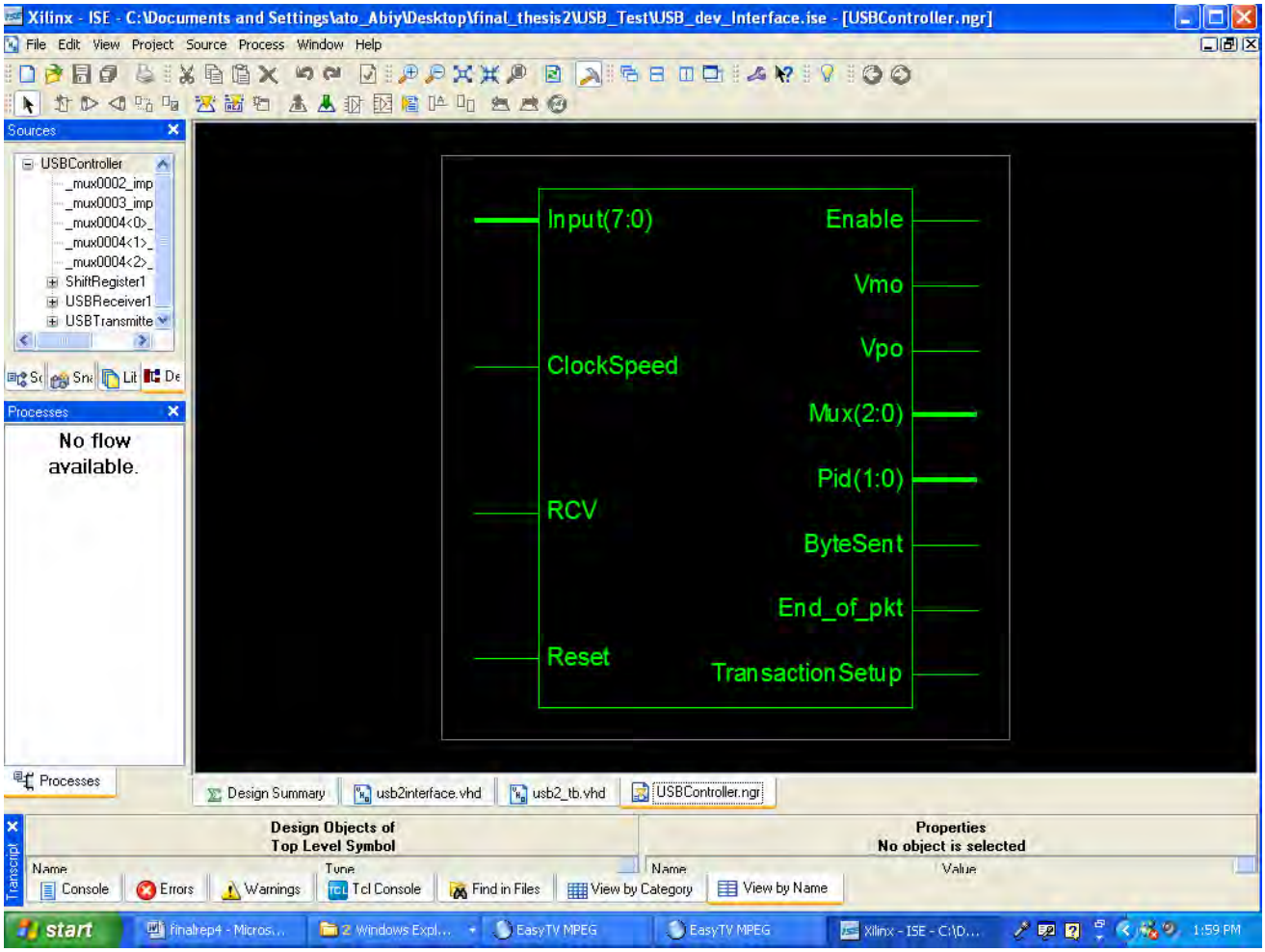

Figure 4-7: RTL View of the Controller (top module)

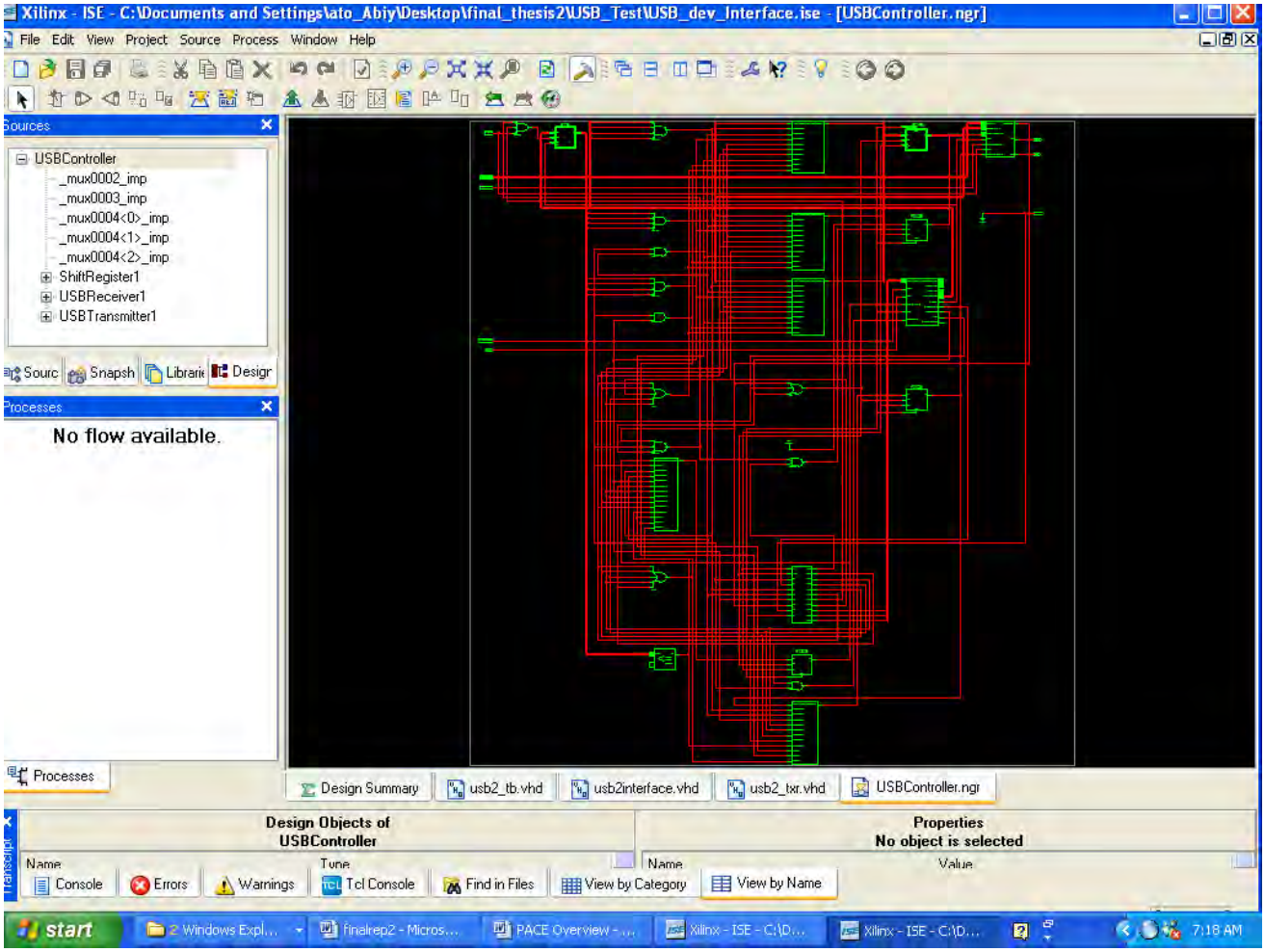

Figure 4-8: RTL Schematic View of the Top Module (controller)

As can be seen from the synthesis results, the RTL-level (behavioral) simulation enabled to verify or simulate the VHDL description at the system or chip level.

The Technology Viewer enabled to view a Technology Level netlist as a schematic as shown in Figures 4-9 and 4-10. This schematic shows the design in terms of logic elements optimized to the target architecture (for example, in terms of Lookup tables or LUTs) and has been generated after the optimization and technology targeting phase of the synthesis process. Viewing this schematic enabled to see gate-level representation of the VHDL optimized for the specific target (Technology).

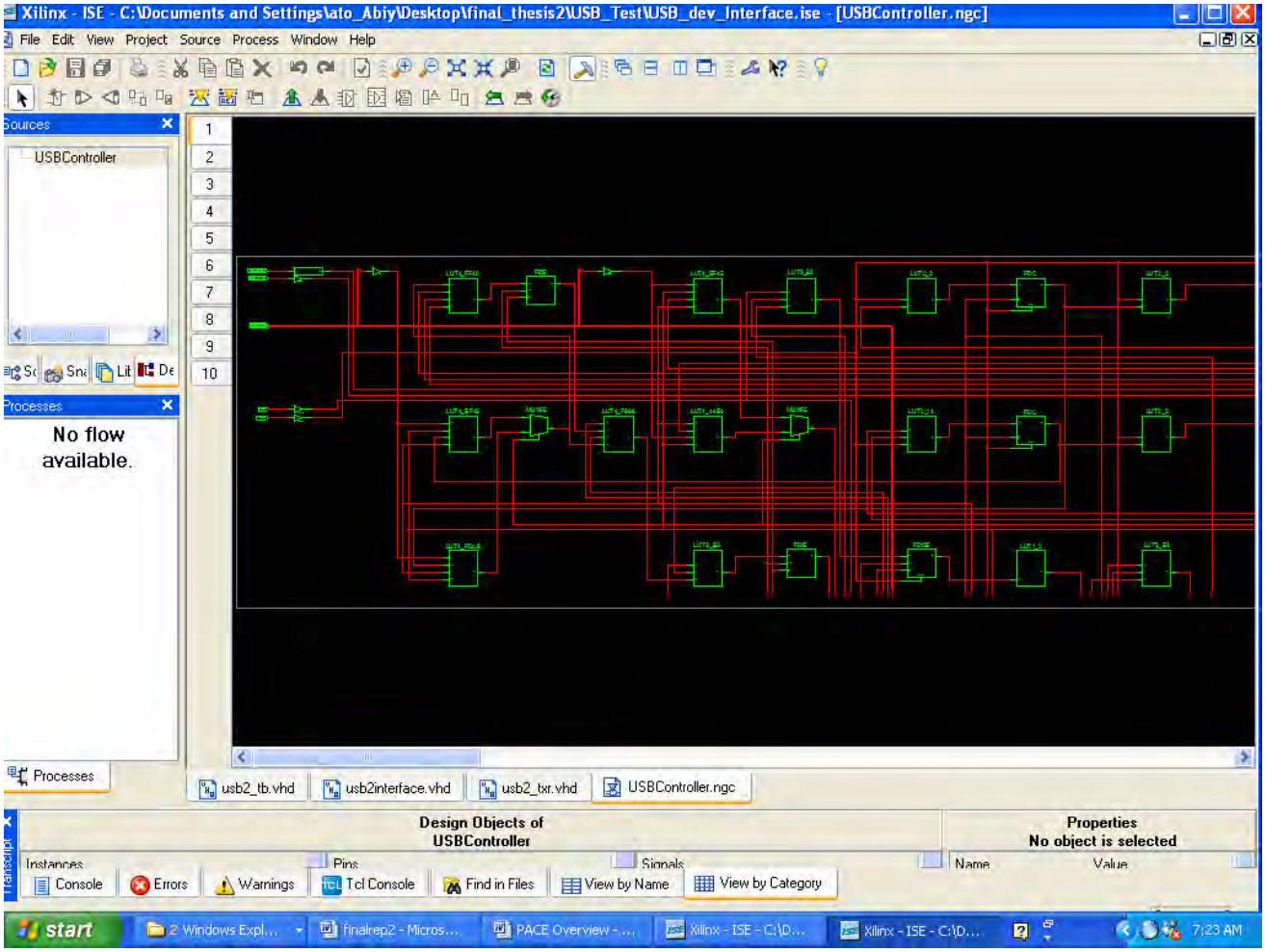

Figure 4-9: Technology schematic view of top module, controller (input side)

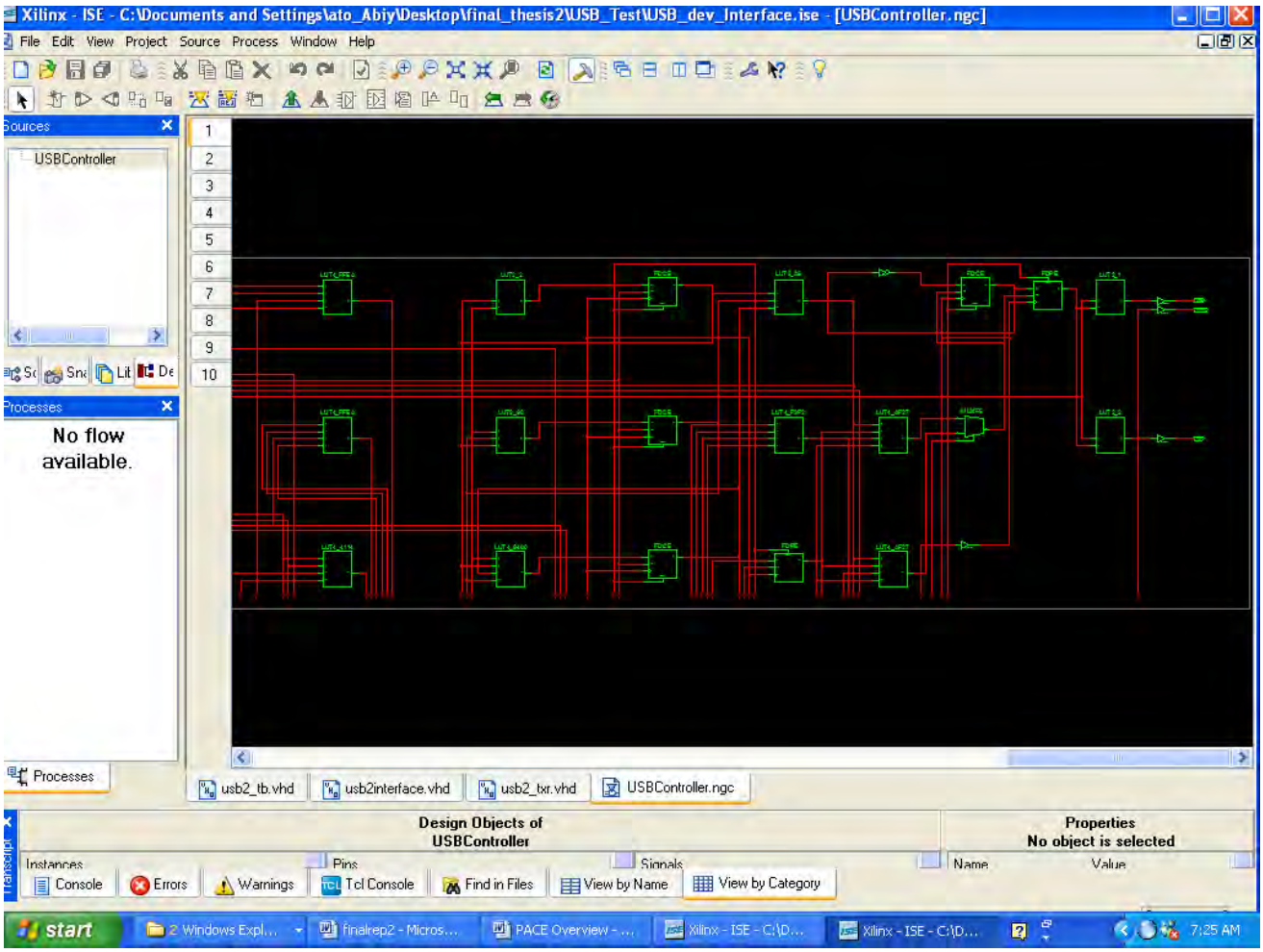

Figure 4-10: Technology schematic view of top module, controller (output side)

4.2.2 Implementation

After synthesis, design implementation has been run, which converted the logical design into a physical file format that has been downloaded to the selected target device. It includes the following steps:

- *Translate* - merges the incoming netlists and constraints in to Xilinx design file.
- *Map* - fits the design in to the available resources on the target device.
- *Place and Route* - places and routes the design to the timing constraints.
- *Programming file generation* - creates a bit stream file that can be downloaded to the device.

The target of the USB Interface designed for this thesis is Xilinx Spartan 3 FPGA (XC3S200). The design has been implemented on the target FPGA. After translation, mapping and placing and routing processes, the device utilization summary of the design was generated as shown in Figure 4-11. The summary shows the design's modular name, the target device and the device utilization summary in tabular format. As can be seen in the figure, the available logic to be utilized of the selected target FPGA, the logic used by the

implementation of the device under design, and utilization in percentage are summarized. From this data it can be seen that the design has taken small amount of logic as compared to the available resources implying the usage of optimal resources and the unused logic can be available for other designs.

Input and output pin assignments of the design on the target FPGA is done using PACE editor (see Figure 4-14 label number 2). Pinout and Area Constraints Editor (PACE™) is an interactive graphical application used to view and edit location constraints for I/Os, to create area constraints for logic in a design, and to determine resource requirements of a design.

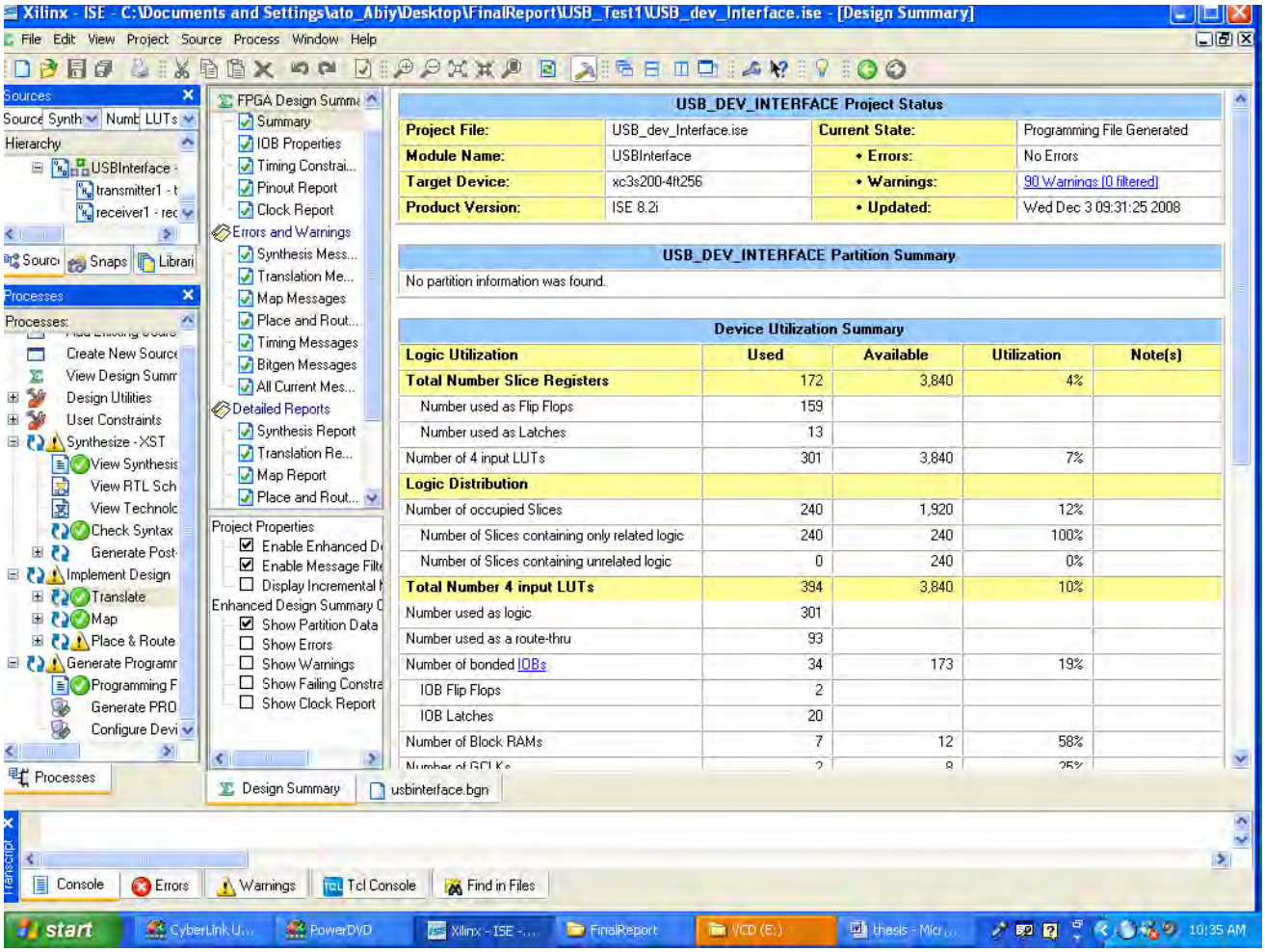

Figure 4-11: Device Utilization Summary

Figure 4-12 shows automatic placement result of the device under design in the floorplanner design view. This methodology can be used to view physical constraints and to improve the performance of the automatically placed and routed design. It helps to determine where to place logic in floorplan for optimal results.

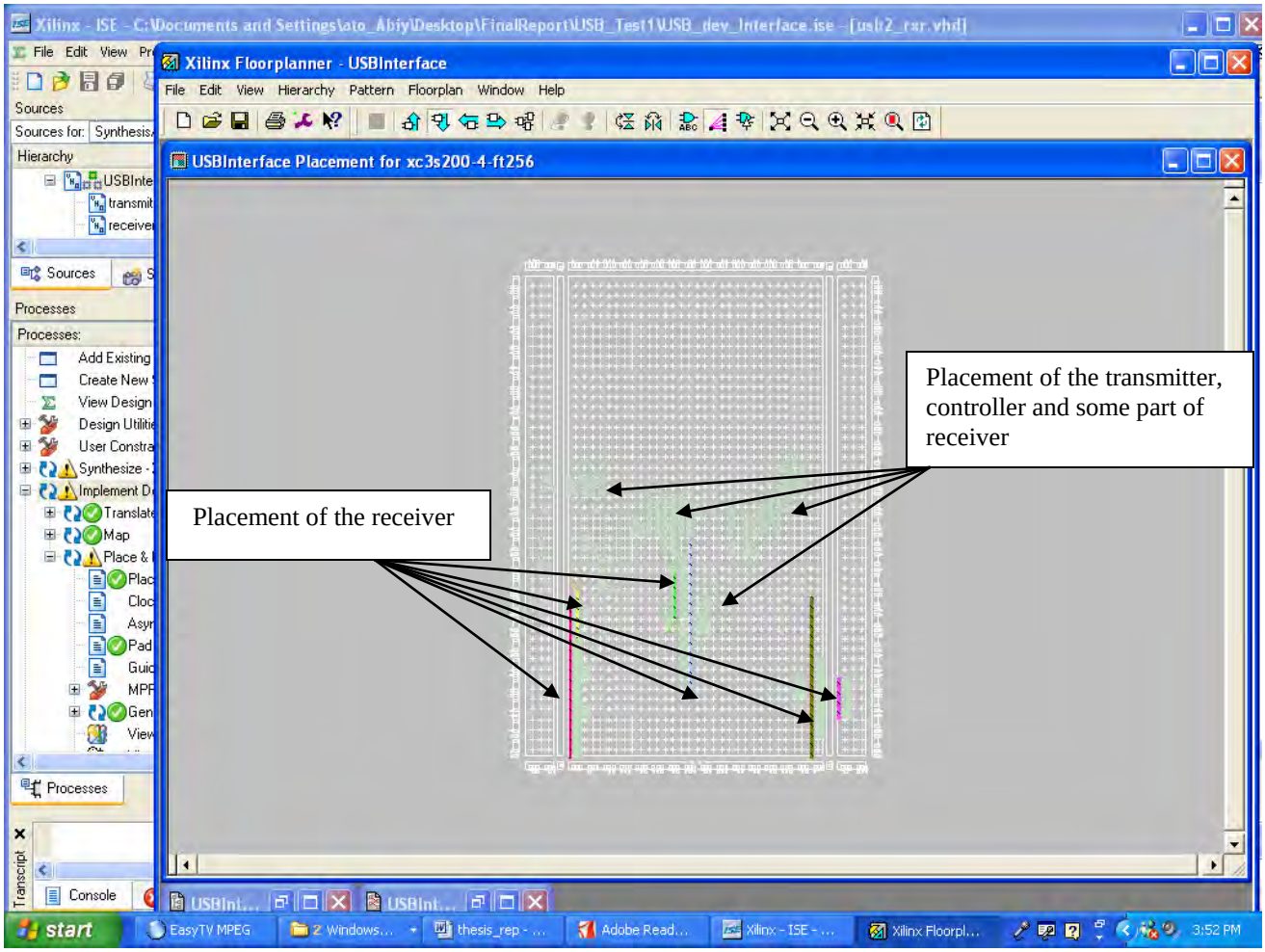

Figure 4-12: Floorplanning view after Place and Route

4.3 Programming the FPGA

4.3.1 Downloading the Design to the Spartan 3 Demo Board

After implementation of the design, programming the target FPGA has been done by using IMACT Device Configuration application software (see Figure 4-14, label 3), which is part of Xilinx ISE. Using the IMPACT software, programming file has been generated in the form of bit streams and this bit stream file was downloaded on to the target FPGA through JTAG cable. The JTAG cable has been connected to the parallel port of the PC at one end and to the JTAG port of the Spartan 3 demo board on the other end.

4.4 Testing the programmed FPGA and the result of the test

After the target FPGA was programmed, then, it has been connected to PC using USB cable as shown in figure 4-13. At this time the FPGA was seen to get power from the USB cable with out requiring external power adapter. And also the operating system detected the programmed FPGA and automatically displayed a message (see Figure 4-14, label 4). Even though the operating system detected the programmed FPGA as USB device, the displayed

message implied that it was not recognized as a USB device. This is because no descriptors have been available. Because of the lack of descriptors, the operating system doesn't enumerate it as one of the USB device classes to search for it the proper device driver. Descriptors and the important device attributes can be provided so that the operating system can enumerate and recognize the device. This is left for programmable device designer in order to maintain the complexity of this design.

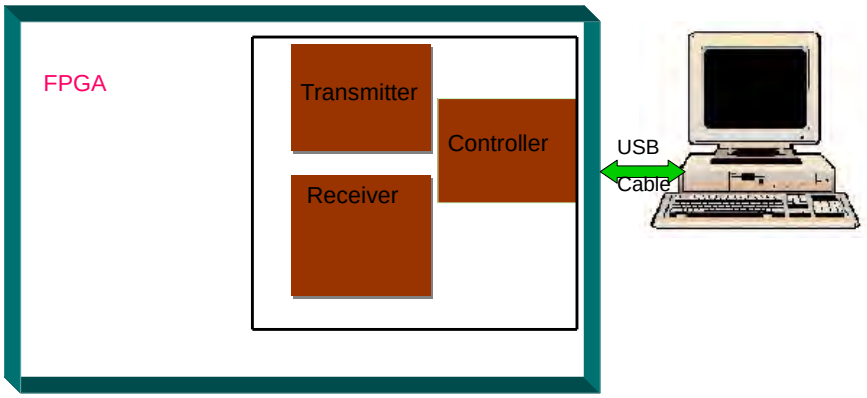

Figure 4-13: USB Interface Hardware implemented on target FPGA and tested with PC

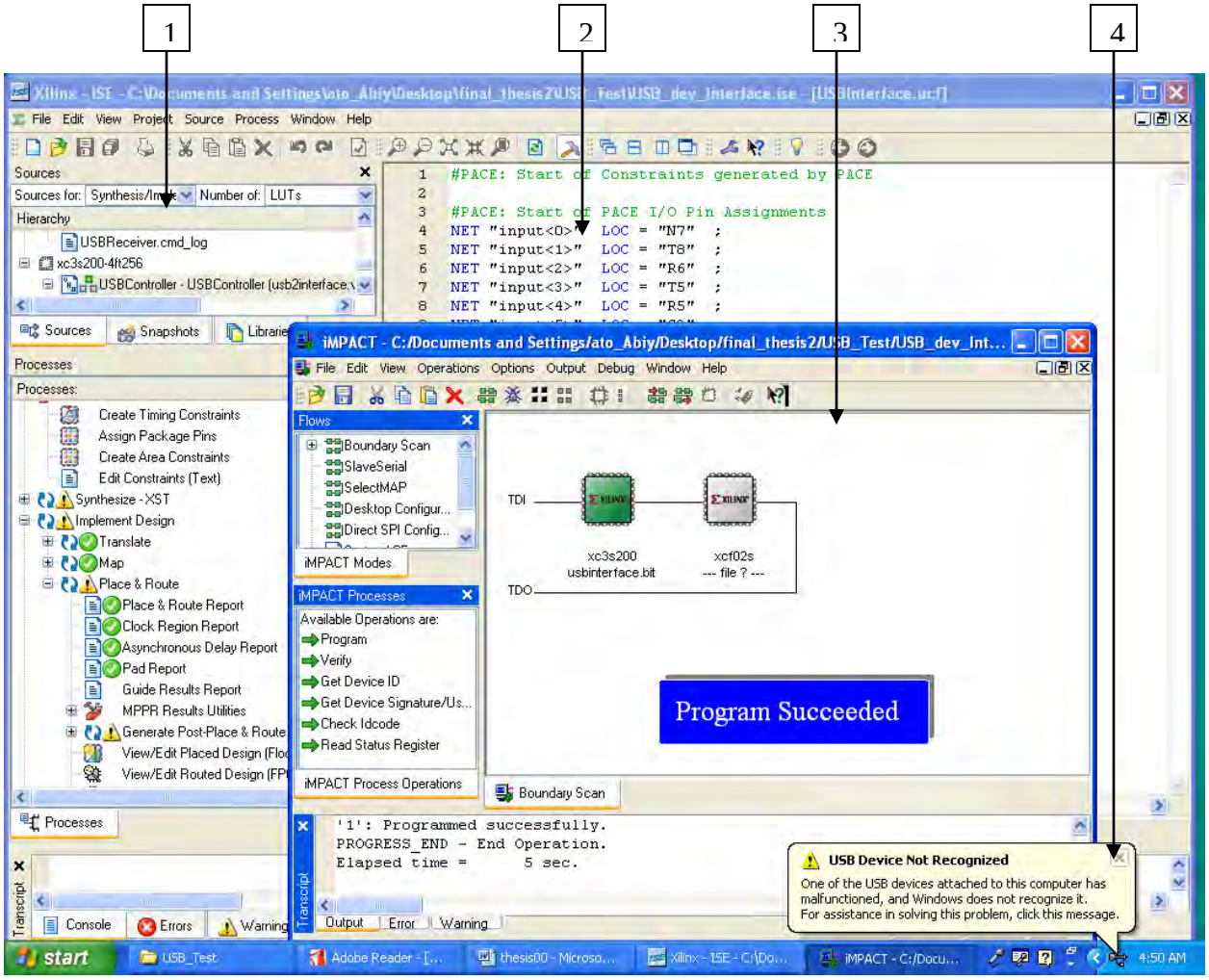

Figure 4-14: Result of testing the FPGA with PC using USB cable

Labels used in Figure 4-14:

1. Xilinx ISE
2. PACE Editor
3. IMPACT software,
4. Message displayed by the OS

Chapter 5

Conclusion, Application of the thesis and Future work

5.1. Conclusion

The objective of describing USB Interface hardware in VHDL, functional simulation of the design, synthesis and implementation of it on the target FPGA is achieved. The simulation results show that functional test of the design is working such that the transmitter is able to transmit and the receiver could also receive as intended. The controller has performed its controlling activities by synchronizing the transmitter and the receiver. It enabled and disabled the transceiver interface based on the type of the transaction. The synthesis results show that how the VHDL descriptions of the hardware are converted to logical components as can be seen in RTL and technology level schematics; and verifying or simulating the VHDL description at the system or chip level. Also, implementation results show how the design has been mapped, placed and routed taking optimal resources of the target FPGA. The test of the programmed FPGA showed that the operating system of the PC detected the device as USB device. This can be further improved to provide descriptors for the device so that the PC enumerates it and searches the appropriate USB device driver to recognize it as one of the USB device classes. Despite the lack of adequate information on the USB protocol and hardware standard specifications that consumed much time for study, the research also resulted in giving much experience to perform designing such complex systems for the future. Moreover, USB standard, VHDL, Xilinx ISE, FPGA, and Windows class drivers and more have been well experienced during the research periods.

5.1 Application of the thesis

Because the USB Interface is provided in the form of VHDL, programmable device designers can incorporate this entire work in their own design to give their design a USB capability, or use the USB Interface on FPGA and then interface hardware such as microcontroller or PC. It could be adjusted according to their design needs and it can save much design time and test processes; so, they need not use a separate chip for this purpose.

5.3 Future Study

Some modifications can be made to include more features to the work. The USB Interface Hardware for this thesis is made to comprise most of the USB controller features. In order to maintain the complexity of the design, it is made not to include descriptors by itself leaving this to the programmable device. The PC can enumerate the device and searches the appropriate device driver for it after knowing its different attributes such as the device's capability, its class, vendor ID and product ID and other required attributes. This will be valuable feature to include.

Bibliography

[1] Kevin Skahill, VHDL for Programmable Logic, Addison Wesley publishing,1996.

[2] Don Anderson, USB System Architecture, Addison-Wesley Developer's Press,2001.

[3] http://www.usb.org/, [Accessed March 12, 2008].

[4] Zainalabedin Navabi, VHDL Analysis and Modeling of Digital systems, McGraw-Hill, 1993.

[5] http://www.opencores.org, [Accessed December 7, 2007].

[6] Ronald L.Krutz, Intrfacing Techniques In Digital Design With Emphasis on Microprocessors, Jhon Wisley and Sons, Inc., 1988.

[7] M. Morris Mano, Computer System Architecture, 3rd Ed, Print - Hall, 2003.

[8] Spartan-3 FPGA Family Complete Data Sheet, DS099 November 30, 2007, http://www.xilinx.com/spport/docmentation/data-sheets/ds099.pdf, [Accessed March 1, 2008].

[9] MYKE PREDKO, Programming and Customizing Microcontroller, 6 Ed, TATA McGraw-Hill, 2003.

[10] http://www.intel.com, [Accessed March 9, 2008].

[11] Staurt M.Asser, Vincent J.Stigliano, and Richard F.Bahrenburg, Microcomputer Servicing Practical System and Troubleshooting, MERRILL PUBLISHING COMPANY , 2003.

[12] www.beyondlogic.org, [Accessed March 10, 2008].

[13] http://en.wikipedia.org/wiki/Universal_Serial_Bus, [Accessed March 27, 2008].

[14] Wayne Wolf, Modern VLSI Design A system Approach, PTR Prentice Hall, 1994.

[15] http://www.digilentinc.com, [Accessed March 9, 2008].

[16] Stallings William, Computer Organization and Architecture, 3rd Edition, Macmillan Publishing Company, 1993.

[17] Peter Hipson, Mastering Windows XP Registry, SYBEX Inc., 2002.

Appendix

---VHDL Code For USBController

```
library IEEE;
use IEEE.std_logic_1164.all;

entity USBController is
        port (
              Input: in std_logic_vector (7 downto 0);
                    ------- Output: out std_logic_vector (7 downto 0);
              RCV: in std_logic;
              Vmo: out std_logic;
              Vpo: out std_logic;
              ClockSpeed: in std_logic;
              Reset: in std_logic;
              End_of_Pkt: inout std_logic;
              ByteSent: inout std_logic;
              Pid:inout std_logic_vector (1 downto  0);
              Mux: inout std_logic_vector (2 downto 0);
              TransactionSetup:  inout std_logic;
              Enable: in std_logic

                    -----EndpointNo: out std_logic_vector (3 downto 0);
                    -----DeviceAddress: out std_logic_vector (6 downto 0)
);
end USBController;

architecture USBController of USBController is

component USBTransmitter
      port (
            Reset: in std_logic;
            Clock: in std_logic;
            Input: in std_logic_vector (7 downto 0);
            InputSelect: in std_logic_vector(2 downto 0);
            PacketIDSelect: in std_logic_vector(1 downto 0);
            TransmitterEnable: in std_logic;
            End_of_pkt_Send: in std_logic;
            Vpo: out std_logic;
            Vmo: out std_logic;
            ByteSentOut: out std_logic
);
end component;

component USBReceiver
     port (
          Reset: in std_logic;
          Clock: in std_logic;
          Input: in std_logic;
         OutputSelect: in std_logic_vector(2 downto 0);
          ReceiverEnable: in std_logic;
          SpeedEnable: in std_logic;
          Output: out std_logic_vector(7 downto 0);
          ReceivedByte: out std_logic;
          PacketIDOut: out std_logic_vector(3 downto 0);
          Endpoint: out std_logic_vector(3 downto 0);
          Start: out std_logic;
          HandshakePkt: out std_logic;
          TransactionSetup: out std_logic;
          DeviceAddress: out std_logic_vector(6 downto 0);
```

```
        ClockOut: out std_logic
);
end component;

---=========================================================
---Declaration of signals for SpeedEnable
---=========================================================

signal SpeedEnable: std_logic;
signal MuxSelect: std_logic_vector (2 downto 0);
signal EnableTransmitter: std_logic;
signal EnableReceiver: std_logic;

---=========================================================
---Declaration of signals which are to be applied on controller.
---=========================================================

signal PacketIDToSend: std_logic_vector(1 downto 0);
signal End_of_pkt: std_logic;
signal ByteSentOut: std_logic;
signal ReceivedByte: std_logic;

---=========================================================
---Declaration of signals for the controller
---=========================================================

constant TransactionIn: std_logic_vector (3 downto 0) := "0011";
constant TransactionOut: std_logic_vector (3 downto 0) := "0010";
constant TransactionSOF: std_logic_vector (3 downto 0) := "0000";
signal TransmitterMuxSelect: std_logic_vector (2 downto 0);

---=========================================================
---Declaration of signals that control the type of transaction going on.
---=========================================================

signal TransactionType: std_logic_vector (3 downto 0);

---=========================================================
---The state machine of the controller represented with 10 named states: ShiftRegister1 type.
---=========================================================

type ShiftRegister1Type is (Start, PacketIDToken, FirstByte, SecondByte, SyncronData, PacketIDData, CRCData,
PacketIDHsk, Data, SyncronHsk);
signal ShiftRegister1: ShiftRegister1Type;

---=========================================================
---Declaration of signals for the controller which were ports initially.
---=========================================================

signal Resume: std_logic;
signal HandshakePkt: std_logic;
signal PacketIDType : std_logic_vector(3 downto 0);
signal Isochron: std_logic := '0';
signal Clock: std_logic;

begin

USBTransmitter1: USBTransmitter port map(Reset, Clock, Input, TransmitterMuxSelect, PacketIDToSend,
EnableTransmitter, End_of_pkt, Vpo, Vmo, ByteOut);

USBReceiver1: USBReceiver port map (Reset, ClockSpeed, RCV, MuxSelect,  EnableReceiver,  SpeedEnable, Output,
ReceivedByte, PacketIDType,  EndpointNo, Resume,
 HandshakePkt, TransactionSetup, DeviceAddress, Clock);
```

```
---=====================================================
---The controller's internal process continues
---=====================================================

StateControl: process (Clock, Reset)

---=====================================================
---Declaration of variable for the state machine.
---=====================================================

variable EndCount: std_logic;
begin

if Reset='1' then
        ShiftRegister1 <= Start;

---=====================================================
---Setting the states of the controller by providing default values and conditions.
---=====================================================

        EndCount := '1';
        EnableReceiver <= '0';
        EnableTransmitter <= '0';
        Enable <= '0';
        PacketIDToSend <= "00";
        TransmitterMuxSelect <= "111";
elsif Clock'event and Clock = '1' then
if ReceivedByte = '1' OR Resume = '1' OR ByteOut = '1' then

---=====================================================
--- Values and conditions to move the controller from one state to onother..
---=====================================================

        EndCount := '1';
case ShiftRegister1 is
when Start =>
        EnableReceiver <= '0';
        EnableTransmitter <= '0';
if Resume = '1' then
        ShiftRegister1 <= PacketIDToken;
        EnableReceiver <= '1';
end if;
when PacketIDToken =>
if HandshakePkt = '1' then
       ShiftRegister1 <= Start;
elsif HandshakePkt = '0' then
       ShiftRegister1 <= FirstByte;
end if;
when FirstByte =>
       TransactionType <= PacketIDType;
       ShiftRegister1 <= SecondByte;
when SecondByte =>
if PacketIDType = TransactionSOF then
        ShiftRegister1 <= Start;
elsif TransactionType = TransactionOut and Resume = '1' then
        ShiftRegister1 <= PacketIDData;
elsif TransactionType = TransactionIn then

        ShiftRegister1 <= SyncronData;
        TransmitterMuxSelect <= "111";
        EnableReceiver <= '0';
        EnableTransmitter <= '1';
end if;
when SyncronData =>
```

```
if TransactionType <= TransactionIn then
        EnableReceiver <= '0';
        EnableTransmitter <= '1';
end if;
if ByteSentOut = '1' then
        ShiftRegister1 <= PacketIDData;
end if;
        TransmitterMuxSelect <= "000";
        PacketIDToSend <= "11";
when PacketIDData =>
if TransactionType = TransactionOut then
        ShiftRegister1 <= Data;
elsif ByteOut = '1' then
        ShiftRegister1 <= Data;
end if;
        TransmitterMuxSelect <= "011";
when CRCData =>
if Isochron = '1' and EndCount = '1' then
        ShiftRegister1 <= Start;
elsif TransactionType = TransactionIn and Resume = '1'
and EndCount = '1' then
        ShiftRegister1 <= PacketIDHsk;
elsif Isochron = '0' and TransactionType = TransactionOut
and EndCount = '1' then
        ShiftRegister1 <= SyncronHsk;
        EnableTransmitter <= '1';
        EnableReceiver <= '0';
        TransmitterMuxSelect <= "111";
end if;
when PacketIDHsk =>
if ByteSentOut = '1' then
        ShiftRegister1 <= Start;
end if;
when Data =>
if EndCount = '1' then
if TransactionType = TransactionOut then
        ShiftRegister1<= CRCData;
elsif (TransactionType = TransactionIn) and (ByteSentOut = '1') then
        ShiftRegister1 <= CRCData;
end if;
end if;
when SyncronHsk =>
if TransactionType = TransactionOut then
        EnableTransmitter<= '1';
        EnableReceiver <= '0';
        TransmitterMuxSelect <= "000";
end if;
if ByteOut = '1' then
        ShiftRegister1 <= PacketIDHsk;
end if;
when others =>
null;
end case;
end if;
end if;
end process;

---=======================================================
---Multiplexer outputs based on different selected values provided here.
---=======================================================

MuxSelectAssignment:
MuxSelect  <= "000" when (ShiftRegister1 = PacketIDToken) else
"001" when (ShiftRegister1 = FirstByte) else
```

```
"010" when (ShiftRegister1 = SecondByte) else
"111" when (ShiftRegister1 = SyncronData) else
"000" when (ShiftRegister1 = PacketIDData) else
"100" when (ShiftRegister1 = CRCData) else
"000" when (ShiftRegister1 = PacketIDHsk) else
"011" when (ShiftRegister1 = Data) else
"111" when (ShiftRegister1 = SyncronHsk) else
"000";

SpeedEnable: process(Resume)

begin

if Resume = '1' and ShiftRegister1 = Start then
        SpeedEnable <= '1';
else
        SpeedEnable <= '0';
end if;
end process;
end USBController;
```

---VHDL Code for USBTransmitter

```
library IEEE;
use IEEE.std_logic_1164.all;

entity USBTransmitter is
port (
        Reset: in std_logic;
        Clock: in std_logic;
        Input: in std_logic_vector (7 downto 0);
        InputSelect: in std_logic_vector(2 downto 0);
        PacketIDSelect: in std_logic_vector(1 downto 0);
        TransmitterEnable: in std_logic;
        End_of_pkt_Send: in std_logic;
        Vpo: out std_logic;
        Vmo: out std_logic;
        ByteSentOut: out std_logic
);
end USBTransmitter;

architecture USBTransmitter of USBTransmitter is

signal PreviousOutput: std_logic;
signal DataStuffed: std_logic;
signal CountBit: integer range 0 to 6;
signal CountShift: integer range  0 to 8;
signal ShiftRegister: std_logic_vector(7 downto 0);
signal MuxOutput: std_logic_vector(7 downto 0);
signal SentByte: std_logic;
signal Output: std_logic;

---====================================================================
--- Declaration of signals for CRC 16 calculator.
---====================================================================

signal CRCState, NextCRCState, CRC16Out: std_logic_vector(15 downto 0);

---====================================================================
-- Declaration of signal for PacketID Generator.  (Packet Identifier)
---====================================================================
```

```
signal PacketID: std_logic_vector(7 downto 0);

begin

TranInterface: process (Output, End_of_pkt_Send)

begin

    if End_of_pkt_Send = '1' then
       Vpo <= '0';
       Vmo <= '0';
       else
    if Output = '1' then
       Vpo <= '1';
       Vmo <= '0';
       else
       Vpo <= '0';
       Vmo <= '1';
       end if;
       end if;
       end process;

Par_to_Ser_n_Bit_Add : process (Clock, Reset)

variable ShiftRegisterTemp: std_logic_vector(7 downto 0);
variable DataStuffedTemp: std_logic;
variable CountShiftTemp: integer range 0 to 8;
variable CountBitTemp: integer range  0 to 6;

begin

     if Reset = '1' then
          Output <= '1';
          PreviousOutput <= '0';
          DataStuffedTemp := '1';
          CountBitTemp := 0;
          CountShiftTemp := 0;
          ShiftRegisterTemp := "00000000";
          SentByte <= '0';
          DataStuffed <= DataStuffedTemp;
          CountShift <= CountShiftTemp;
          CountBit <= CountBitTemp;

elsif Clock'EVENT and Clock = '1' then
          DataStuffedTemp := DataStuffed;
          CountShiftTemp := CountShift;
          CountBitTemp := CountBit;
          ShiftRegisterTemp := ShiftRegister;
     if CountShiftTemp = 0 then
          ShiftRegisterTemp := MuxOutput;
     end if;

     if TransmitterEnable = '1' then
     if CountBitTemp = 6 then
          DataStuffedTemp := '0';
          CountBitTemp := 0;
else
          DataStuffedTemp := ShiftRegisterTemp(7);
          ShiftRegisterTemp := ShiftRegisterTemp(6 downto 0) & '1';

---============================================================
-- Counter  is incremented only if  normal bit is sent.
---============================================================
          CountShiftTemp := CountShiftTemp + 1;
```

```
     end if;

---============================================================
-- Number of 1's is counted for the purpose of adding stuffed bit.
---============================================================

    if DataStuffedTemp = '1' then
        CountBitTemp := CountBitTemp + 1;
else
        CountBitTemp := 0;
    end if;

    if CountShiftTemp = 8 then
        ByteSentOut <= '1';
        CountShiftTemp := 0;
        ShiftRegisterTemp := MuxOutput;
else
        ByteSentOut <= '0';
    end if;
    end if;

if DataStuffedTemp = '0' then
        PreviousOutput <= not(PreviousOutput);
        Output <= PreviousOutput;
end if;

---Update signal values.

ShiftRegister <= ShiftRegisterTemp;
DataStuffed <= DataStuffedTemp;
CountShift <= CountShiftTemp;
CountBit <= CountBitTemp;
end if;
end process;

---=========================================================
--- Calculation of 16 bit CRC
---=========================================================

process(Clock, Reset, DataStuffed)
begin

if Clock'EVENT and Clock = '1' then
         CRC16Out <= NextCRCState;
         CRCState <= NextCRCState;
end if;

if Reset = '1' then
         CRCState <= "0000000000000000";
         CRC16Out <= "0000000000000000";
         NextCRCState <= "0000000000000000";
else
         NextCRCState (15) <= CRCState (14) XOR DataStuffed XOR CRCState (15);
         NextCRCState (14) <= CRCState (13);
         NextCRCState (13) <= CRCState (12);
         NextCRCState (12) <= CRCState (11);
         NextCRCState (11) <= CRCState (10);
         NextCRCState (10) <= CRCState (9);
         NextCRCState (9) <= CRCState (8);
         NextCRCState (8) <= CRCState (7);
         NextCRCState (7) <= CRCState (6);
         NextCRCState (6) <= CRCState (5);
         NextCRCState (5) <= CRCState (4);
         NextCRCState (4) <= CRCState (3);
```

```
        NextCRCState (3) <= CRCState (2);
        NextCRCState (2) <= CRCState (1) XOR DataStuffed XOR CRCState (15);
        NextCRCState (1) <= CRCState (0);
        NextCRCState (0) <= DataStuffed XOR CRCState (15);
end if;
end process;

---================================================================
---Generating Packet Identifier (PID)
---================================================================

process (PacketIDSelect)

begin

case PacketIDSelect is


---ACKNOWLEDGMENT
when "00" => PacketID <= "01001011";

---NEGATIVE ACKNOWLEDGMENT
when "01" => PacketID <= "01011010";

--- DATA0 Packet.
when others =>PacketID <= "11000011";
end case;
end process;

---================================================================
---The working of the Multiplexer
---================================================================

process (InputSelect, Input, CRC16Out)

begin

case InputSelect is

when "000" => MuxOutput <= PacketID;
when "100" => MuxOutput <= CRC16Out(7 downto 0);
when "101" => MuxOutput <= CRC16Out(15 downto 8);
when "011" => MuxOutput <= Input;
when "111" => MuxOutput <= "00000001";
when others => MuxOutput <= Input;
end case;
end process;
end USBTransmitter;
```

---VHDL Code For USBReceiver

```
library IEEE;
use IEEE.std_logic_1164.all;

entity USBReceiver is
        port (
                Reset: in std_logic;
                Clock: in std_logic;
                Input: in std_logic;
                OutputSelect: in std_logic_vector(2 downto 0);
                ReceiverEnable: in std_logic;
                SpeedEnable: in std_logic;
```

```
            Output: out std_logic_vector(7 downto 0);
            ReceivedByte: out std_logic;
            PacketIDOut: out std_logic_vector(3 downto 0);
            Endpoint: out std_logic_vector(3 downto 0);
            Start: out std_logic;
            HandshakePkt: out std_logic;
            TransactionSetup: out std_logic;
            DeviceAddress: out std_logic_vector(6 downto 0);
            ClockOut: out std_logic
);
end USBReceiver;

architecture USBReceiver of USBReceiver is

signal PreviousInput: std_logic;
signal DataStuffed: std_logic;

---=========================================================
--Signal for counting consecutive 1's
---=========================================================
signal CountBit: integer; --Keeps track of consecutive 1's.
signal Data: std_logic_vector (7 downto 0);

---=========================================================
 ---Signal for counting number of bits shifted by serial to parallel converter.
---=========================================================

signal CountShift: integer;


---=========================================================
--- clock signals
---=========================================================

signal LeadToBuffer: std_logic;

---=============================================================
---Declaration of signals for Phase Locked Loop (PLL)
---=============================================================

signal PreviousPllInput: std_logic;
signal UserClock: std_logic;
signal PreviousOutputSignal: std_logic;
signal CountPll: integer;

---=========================================================
---Declaration of signals for Multiplexor.
---=========================================================

signal ByteOut: std_logic_vector(7 downto 0);

---=========================================================
---Declaration of signals for Packet ID checker.
---=========================================================

signal PacketID: std_logic_vector(7 downto 0);

---=========================================================
--Declaration of signals for SOP (start of packet detector).
---=========================================================

type SOPStateType  is (S0,S1,S2,S3,S4,S5,S6,S7,S8);
signal SOPState,NextSOPState:SOPStateType;
```

```
---============================================================
---Declaration of signals for CRC calculators.
---============================================================

signal NextCRCState: std_logic_vector(15 downto 0);
signal CRCState: std_logic_vector(15 downto 0);
signal CRC16Out: std_logic_vector(15 downto 0);
signal NextCRC5State: std_logic_vector(4 downto 0);
signal CRC5State: std_logic_vector(4 downto 0);
signal CRC5Out: std_logic_vector(4 downto 0);

begin

Clock_Gen_or_Driver:   process(UserClock)

begin

if UserClock = '1' then
        ClockOut <= '1';
else
        ClockOut <= '0';
end if;
end process;

StartOfPacketDetector:   process(SOPState, Input)

begin

case SOPState is

when S0=>if Input = '0' then NextSOPState <= S1;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S1=>if Input = '1' then NextSOPState <= S2;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S2=>if Input = '0' then NextSOPState <= S3;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S3=>if Input = '1' then NextSOPState <= S4;Start <= '0';

else NextSOPState <= S0;Start <= '0';end if;
when S4=>if Input = '0' then NextSOPState <= S5;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S5=>if Input = '1' then NextSOPState <= S6;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S6=>if Input = '0' then NextSOPState <= S7;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S7=>if Input = '0' then NextSOPState <= S8;Start <= '0';
else NextSOPState <= S0;Start <= '0';end if;
when S8=>NextSOPState <= S0;Start <= '1';
end case;
end process;

---============================================================
---Process for updating the state of SOP detector.
---============================================================

process(UserClock, Reset)

begin

if Reset = '1' then
        SOPState <= S0;
        elsif rising_edge(UserClock) then
        SOPState <= NextSOPState;
end if;
```

```
end process;

---================================================================
---Process for Phase Locked Loop (PLL) for clock recovery from data
---================================================================

PhaseLL: process (Clock)

---================================================================
---Declaration of variables
---================================================================

variable PreviousOutput: std_logic;

begin

if Reset = '1' then
        PreviousPllInput <= '0';
        PreviousOutputSignal <= '0';
        CountPll <= 0;
        PreviousOutput := '0';
elsif clock'EVENT and clock='1' then
        PreviousOutput := PreviousOutputSignal;

if not(PreviousPllInput = Input) then
        PreviousOutput := not (PreviousOutput);
        CountPll <= 0;
elsif CountPll = 1 then
        PreviousOutput := not (PreviousOutput);
        CountPll <= 0;
else
        CountPll <= CountPll + 1;
end if;
end if;
        PreviousPllInput <= Input;
        PreviousOutputSignal <= PreviousOutput;
        UserClock <= PreviousOutput;
end process;

ReceiverInternal: process (UserClock, Reset)

variable DataStuffedTemp: std_logic;
variable DataTemp: std_logic_vector(7 downto 0);
variable CountBitTemp: integer;
variable CountShiftTemp: integer; --Keeps track of number of bits

begin

if Reset = '1' then
        PreviousInput <= '0';
        DataStuffed <= '0';
        CountBit <= 0;
        Data <= "00000000";
        CountShift <= 0;
        ReceivedByte <= '0';
        ByteOut <= "00000000";

---========================================================
---NRZI Decoder
---========================================================

elsif UserClock'EVENT and UserClock = '1' then
if ReceiverEnable = '1' OR SpeedEnable = '1' then
        DataStuffedTemp := DataStuffed;
```

```
        DataTemp := Data;
        CountBitTemp := CountBit;
        CountShiftTemp := CountShift;
if (Input = PreviousInput) then

        DataStuffedTemp := '1';
else
        DataStuffedTemp := '0';
end if;

---=========================================================
---Counting number of 1's for the purpose of removing stuffed bit.
---=========================================================

if DataStuffedTemp = '1' then
        CountBitTemp := CountBitTemp + 1;
else
        CountBitTemp := 0;
end if;

---=========================================================
---The input is added to the data register only if it is not a stuffed bit.
---=========================================================

if not(CountBitTemp = 6) then
        DataTemp(7) := DataTemp(6);
        DataTemp(6) := DataTemp(5);
        DataTemp(5) := DataTemp(4);
        DataTemp(4) := DataTemp(3);
        DataTemp(3) := DataTemp(2);
        DataTemp(2) := DataTemp(1);
        DataTemp(1) := DataTemp(0);
        DataTemp(0) := DataStuffedTemp;
        CountShiftTemp := CountShiftTemp + 1;
else

---=========================================================
---The counter is set to '0' after the stuffed bit is removed.
---=========================================================

        CountBitTemp := 0;
end if;

---=========================================================
---When a byte is received the counter =8 and ReceivedByte=1.
---=========================================================

if CountShiftTemp = 8 then
         ReceivedByte <= '1';
         ByteOut <= DataTemp;

---=========================================================
---The counter is now being ready for the next work by reseting  it to 0.
---=========================================================

          CountShiftTemp := 0;
else

           ReceivedByte <= '0';
end if;
            DataStuffed <= DataStuffedTemp;
            Data <= DataTemp;
            CountBit <= CountBitTemp;
            CountShift <= CountShiftTemp;
```

```
            PreviousInput <= Input;
end if;
end if;
end process;
---================================================================
---Process for the working of the Demultiplexer
---================================================================-

process (OutputSelect, ByteOut)

begin

case OutputSelect is

when "000" => PacketID <= ByteOut;
when "001" =>
Endpoint(3) <= ByteOut(0);
DeviceAddress <= ByteOut(7 downto 1);
when "010" =>
Endpoint(2 downto 0) <= ByteOut(7 downto 5);
when others => Output <= ByteOut;
end case;
end process;

PacketIDChecker: process (Reset, PacketID)

begin

if Reset = '1' then
            HandshakePkt <= '0';
            PacketIDOut <= "0000";
            TransactionSetup <= '0';

---=============================================================
---Start of frame (SOF) transaction
---=============================================================

elsif PacketID = "10100101" then
             PacketIDOut <= "0000";
             HandshakePkt <= '0';

---=============================================================
---SETUP Transaction
---=============================================================

elsif PacketID = "10110100" then
             PacketIDOut <= "0001";
             HandshakePkt <= '0';
             TransactionSetup <= '1';

---=============================================================
--OUT  Transaction
---=============================================================

elsif PacketID = "10000111" then
            --EnableReceiver <= '1';
              PacketIDOut <= "0010";
              HandshakePkt <= '0';
---=============================================================
---IN  Transaction
---=============================================================
elsif PacketID = "10010110" then
             PacketIDOut <= "0011";
             HandshakePkt <= '0';
```

```
---============================================================
---DATA0 transaction
---============================================================

elsif PacketID = "11000011" then
              PacketIDOut <= "0100";
              HandshakePkt <= '0';

---============================================================
---DATA1 transaction
---============================================================
elsif PacketID = "11010010" then
             PacketIDOut <= "0101";
             HandshakePkt <= '0';

---============================================================
---ACKNOWLEDGMENT
---============================================================

elsif PacketID = "01001011" then
              PacketIDOut <= "0110";
              HandshakePkt <= '1';

---============================================================
---NEGATIVE ACKNOWLEDGMENT
---============================================================

elsif PacketID = "01011010" then
               PacketIDOut <= "0111";
               HandshakePkt <= '1';

---============================================================
---STALL
---============================================================

elsif PacketID = "01111000" then
                PacketIDOut <= "1000";
                HandshakePkt <= '1';
end if;
end process;

---============================================================
---Calculation of 16-bit CRC
---============================================================

process(Clock, Reset, DataStuffed)

begin

if Clock'EVENT and Clock = '1' then
        CRC16Out <= NextCRCState;
        CRCState <= NextCRCState;
end if;

if Reset = '1' then
        CRCState <= "0000000000000000";
        CRC16Out <= "0000000000000000";
        NextCRCState <= "0000000000000000";
else
        NextCRCState(15) <= CRCState(14) XOR DataStuffed XOR CRCState(15);
        NextCRCState (14) <= CRCState (13);
        NextCRCState (13) <= CRCState (12);
        NextCRCState (12) <= CRCState (11);
        NextCRCState (11) <= CRCState (10);
```

```
        NextCRCState (10) <= CRCState (9);
        NextCRCState (9) <= CRCState (8);
        NextCRCState (8) <= CRCState (7);
        NextCRCState (7) <= CRCState (6);
        NextCRCState (6) <= CRCState (5);
        NextCRCState (5) <= CRCState (4);
        NextCRCState (4) <= CRCState (3);
        NextCRCState (3) <= CRCState (2);
        NextCRCState (2) <= CRCState (1) xor DataStuffed xor CRCState(15);
        NextCRCState (1) <= CRCState (0);
        NextCRCState (0) <= DataStuffed XOR CRCState(15);
end if;
end process;

---=====================================================
---Calculation of 5-bit CRC
---=====================================================

process(Clock)

begin

if Clock'EVENT and Clock = '1' then
        CRC5Out <= NextCRC5State;
        CRC5State <= NextCRC5State;
end if;

if Reset = '1' then
        CRC5State <= "00000";
        CRC5Out <= "00000";
        NextCRC5State <= "00000";
else
        NextCRC5State(4) <= CRC5State(3);
        NextCRC5State(3) <= CRC5State(2);
        NextCRC5State(2) <= CRC5State(1) XOR DataStuffed XOR CRC5State(4);
        NextCRC5State(1) <= CRC5State(0);
        NextCRC5State(0) <= DataStuffed XOR CRC5State(4);
end if;
end process;
end USBReceiver;
```

---VHDL Code For Testbench

```
library ieee;
use ieee.std_logic_1164.all;

entity USBInterfaceTB is
end USBInterfaceTB;

architecture USBInterfaceTestBench of USBInterfaceTB is

---=====================================================
-- Component declaration of the tested unit
---=====================================================

component USBController
    port(
        Input: in std_logic_vector (7 downto 0);
        ------Output: out std_logic_vector (7 downto 0);
        RCV: in std_logic;
        Vmo: out std_logic;
        Vpo: out std_logic;
```

```
        ClockSpeed: in std_logic;
        Reset: in std_logic;
        End_of_pkt: inout std_logic;
        ByteSent:inout std_logic;
        Pid:inout std_logic_vector (1 downto  0);
        Mux: inout std_logic_vector (2 downto 0);
        TransactionSetup : inout std_logic ;
        Enable: in std_logic

            -----EndpointNo: out std_logic_vector (3 downto 0);
            -----DeviceAddress: out std_logic_vector (6 downto 0)
end component;

---============================================================
---Test bench stimulus signals
---============================================================

signal Input : std_logic_vector(7 downto 0);
signal RCV : std_logic;
signal ClockSpeed : std_logic;
signal Reset : std_logic;
signal End_of_pkt:  std_logic;
signal ByteSent:std_logic;
signal Mux:std_logic_vector(2 downto 0);
signal Pid:std_logic_vector(1 downto 0);
signal Output : std_logic_vector(7 downto 0);
signal Vmo : std_logic;
signal Vpo : std_logic;
signal Enable : std_logic;
signal EndpointNo : std_logic_vector(3 downto 0);
signal TransactionSetup : std_logic;

---============================================================
--Declaration of signal for stopping generated clock signal.
---============================================================

signal SimulationEnd: BOOLEAN:=FALSE;
signal RCVRegister: std_logic_vector(95 downto 0):=

"01010100010011101010101010101011010101000010100011001111101010101010101010101010101010101010
010";

begin

---============================================================
---UUT (Unit under Test) port map
---============================================================

UUT : USBController
port map(
            Input => Input,
            Output => Output,
            RCV => RCV,
            Vmo => Vmo,
            Vpo => Vpo,
             ClockSpeed => ClockSpeed,
             Reset => Reset,
              End_of_pkt=>End_of_pkt,
              mux=>mux,
              pid=>pid,
             Enable => Enable,
             EndpointNo => EndpointNo,
             TransactionSetup => TransactionSetup );
```

```
STIMULUS: process

variable Count: Integer := 0;

begin

Input <= "01000101";
RCV <= '0';
Reset <= '0';
wait for 400 ns;
RCV <= '1';
Reset <= '1';
wait for 400 ns;
RCV <= '1';
Reset <= '0';
wait for 400 ns;
while (Count < RCVRegister'length) loop
RCV <= RCVRegister(RCVRegister'left);
RCVRegister <= (RCVRegister ((RCVRegister 'left - 1) downto 0) & '0');
Count := Count + 1;
wait for 400 ns;
end loop;
SimulationEnd <= TRUE;              --- end of stimulus events

wait;
end process;                        --- end of stimulus process

ClockSpeedProcess : process

begin

if SimulationEnd = FALSE then
        ClockSpeed <= '0';
        wait for 50 ns;
else
wait;
end if;
if SimulationEnd = FALSE then
        ClockSpeed <= '1';
        wait for 50 ns;
else
wait;
end if;
end process;
end USBInterfaceTestBench;

configuration TestBenchUSBInterface of USBInterfaceTB is
      for USBInterfaceTestBench
      for UUT : USBController
use entity work. USBController (USBController);
end for;
end for;
end TestBenchUSBInterface;
```