CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Design synthesizable USB 3.0 using Verilog HDL and simulate design using Cadence

A graduate project submitted in partial fulfillment of the requirements
for the degree of Masters of Science
in Electrical Engineering.

By

Shashank Mehta

May 2012

The graduate project of Shashank Mehta is approved:

Dr. Ramin Roosta	Date
Dr. Ali Amini	Date
Dr. Ronald Mehler, Chair	Date

CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

ACKNOWLEDGEMENT

I would like to thank my professor Dr. Ronald Mehler for being my advisor and guide. I am grateful to him for his continuous support and invaluable inputs he has been providing me through the development of the project. This work would not have been possible without his support and encouragement. I would also like to thank him for showing me some examples that related to the topic of my project.

Besides, I would like to thank the Department of Electrical and Computer Engineering for providing me with a good environment and facilities to complete this project. It gave me an opportunity to participate and learn about Hardware designing. In addition, I would like to thank my professor that he provided me huge and valuable information as the guidance of my project.

Table of Contents

Chapter 1 1

1.1 Architectural Overview 1

1.2 USB 3.0 System Description 1

1.3 Superspeed Architecture 2

1.4 Physical Layer 3

1.5 Link Layer 4

1.6 Protocol Layer 5

1.7 Robustness and Error handling 5

1.8 Power Management 6

1.9 Devices 7

1.10 Peripheral Devices 7

1.11 Hubs 7

1.12 Hosts 8

1.13 Data Flow Models 8

Chapter 2 10

2.1 Superspeed Data Flow Model 10

2.2 Superspeed Communication Flow 10

2.3 Superspeed Protocol 11

2.4 Superspeed Packets 12

2.5 Superspeed Transfer 13

2.6 In Transfer 14

2.7 OUT Transfer 15

2.8 Control Transfers 16

2.9 Bulk Transfers 17

2.10 Interrupt Transfers 17

2.11 Isochronous Transfers 18

2.12 Device Notification 18

2.13 Reliability 18

2.13.1 Physical Layer 18

2.13.2 Link Layer 18

2.13.3 Protocol Layer 18

2.14 Efficiency 18

Chapter 3 19

3.1 Physical Layer 19

3.2 PCI Express PHY Layer 20

3.3 USB Superspeed PHY Layer 21

3.4 PHY/MAC Interface 21

3.5 Transmitter Block Diagram 23

3.6 Receiver Block Diagram 24

3.7 PHY/MAC Interface Signals 25

3.8 Pipe Operation Behavior 27

3.8.1 Clocking 27

3.8.2 Reset 27

3.8.2 Power Management 27

3.8.4 Changing the Signal Rate 28

3.8.5 Clock Tolerance Compensation 29

3.8.6 Error Detection 29

3.8.7 Polarity Inversion 30

3.8.8 Setting Negative disparity 30

3.9 Link initialization and training 30

3.10 Normative Clock Recovery Function 32

Chapter 4 33

4.1 Link Layer 33

4.2 Packets and Packet Framing 33

4.2.1 Header Packet Structure 33

4.2.2 Packet Header 34

4.2.3 Link Control Word 34

4.2.4 Data Packet Payload 35

4.3 Link Command 35

4.4 Link Error 35

4.5 Link Training and Status State Machine (LTSSM) 36

4.5.1 SS_Disable 37

4.5.2 SS_Inactive 38

4.5.3 Rx_Detect 38

4.5.4 Polling 39

4.5.5 Compliance Mode 39

4.5.6 U0 39

4.5.7 U1 40

4.5.8 U2 40

4.5.9 U3 40

4.5.10 Recovery 40

4.5.11 LoopBack 40

4.5.12 Hot_Reset 40

Chapter 5 42

5.1 Design and Simulation 42

5.2 Physical Layer 42

5.2.1 PHY.v 42

5.2.2 CLOCK_GEN.v 42

5.2.3 DATA_RATE.v 42
5.2.4 ENCODER1.v 42
5.2.5 ClockDiv.v 42
5.2.6 PartoSer.v 43
5.2.7 DPLL2.v 43
5.2.8 SertoPar.v 43
5.2.9 ClockDiv.v 43
5.2.10 DFF.v 43
5.2.11 FIFO2.v 44
5.2.12 RX_STATUS.v 44
5.2.13 DECODE.v 44
5.3 Link Layer 44
5.3.1 LTSSM.v 44
5.4 Simulation 44
5.4.1 PHY Layer 44
5.4.2 Link Layer 46
Chapter 6 47
Conclusion 47
Reference 48
Appendix 49

List of Figures

Figure 1.1 USB 3.0 Architecture Overview 1
Figure 1.2 Superspeed communication layer and Power management 3
Figure 2.1 Two back-to-back transactions USB 2.0 vs. SS 15
Figure 2.2 Two back-to-back transactions USB2.0 vs. SS 16
Figure 3.1 Partitioning PHY Layer for USB Superspeed 20
Figure 3.2 PHY/MAC Interface 22
Figure 3.3 Transmitter Block Diagram 23
Figure 3.4 Receiver Block Diagram 24
Figure 3.5 Clock recovery and Data recovery circuit 32
Figure 4.1 Header packet framing 34
Figure 4.2 Packet Header 34
Figure 4.3 Link control word 34
Figure 4.4 Data packet payload with CRC-32. 35
Figure 4.5 State diagram of the LTSSM 37

List of Tables

Table 3.1 Transmit Data Interface Signal 25
Table 3.2 Command Interface Signals 25
Table 3.3 Status Interface Signal 26
Table 3.4 External Signals 26
Table 3.5 Training Sequence Values 31

ABSTRACT

Design synthesizable USB 3.0 using Verilog HDL and simulate design using Cadence

By

Shashank Mehta

Masters of Science in Electrical Engineering

In this project I design USB 3.0 using Verilog HDL and simulate the design in Cadence. My design mainly includes two layers of USB 3.0, Physical Layer and Link Layer. Along with USB 2.0 functionality it includes Superspeed functionality .Physical Layer contains PCI Express and PIPE interface. The design transferred data from transmitter to receiver serially. In the project I manage to transfer data either on 2.5GT/s or on 5.0GT/s depends upon the mode and rate. The design generates clock that runs on two different frequencies i.e. 125MHz and 250MHz that used to transfer data on parallel interface. In Design I manage to capture the data that are coming asynchronously and lock the receiver clock with incoming asynchronous serial data. The Link Layer contains Link Transition and Status State Machine (LTSSM). This is used to manage the link between two ports. It manages the Superspeed and Power of the link by putting the link into appropriate stage according to its usage.

Chapter 1

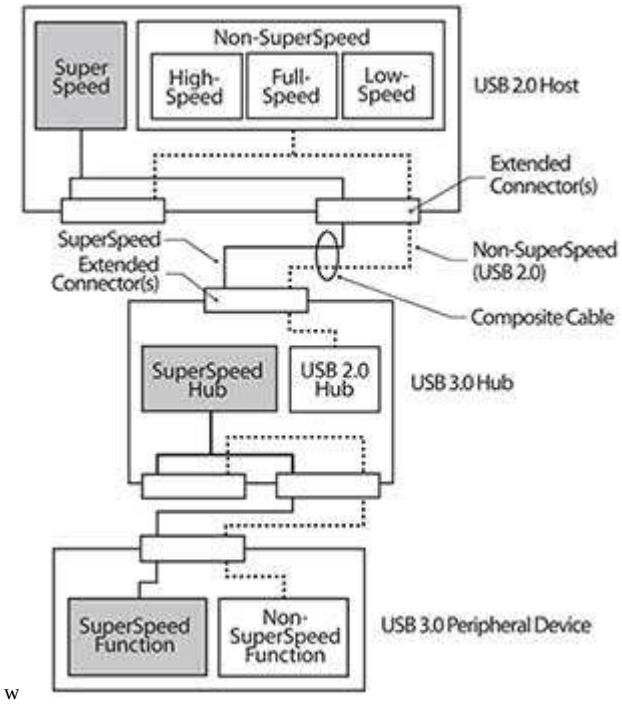
1.1 Architectural Overview

[1] This chapter represents an overview of USB 3.0 architecture and key concepts. It is similar to USB2.0 that it is cable bus supporting data exchange between a host computer and wide range of accessible devices. All the attached devises used host scheduled protocol i.e. that is bus allowed device to be attached, configured, used and detached while others are in operations.

USB 3.0 used dual bus architecture that allowed backward compatibility with USB 2.0. It provides simultaneous operations of Superspeed and non Superspeed.

1.2 USB 3.0 System Description

[1] It has same component like USB2.0 like host, device and interconnect.



w

ww.usb.org

Figure 1.1 USB 3.0 Architecture Overview

USB 3.0 topology is the same as USB 2.0 i.e. tiered [5] star topology with single host at the tier 1 and hubs at lower tier to provide bus connectivity. USB 3.0 provides backward and forward compatibility for connecting USB 3.0 or USB2.0 to USB 3.0 bus.

USB 3.0 connection model allows discovery and configuration of the devices at the highest signaling speed supported by the devices, the highest signaling support between all the hubs and devices.

USB 3.0 hubs are specific class of the devices who provide more connections points to the bus then provided by the devices.

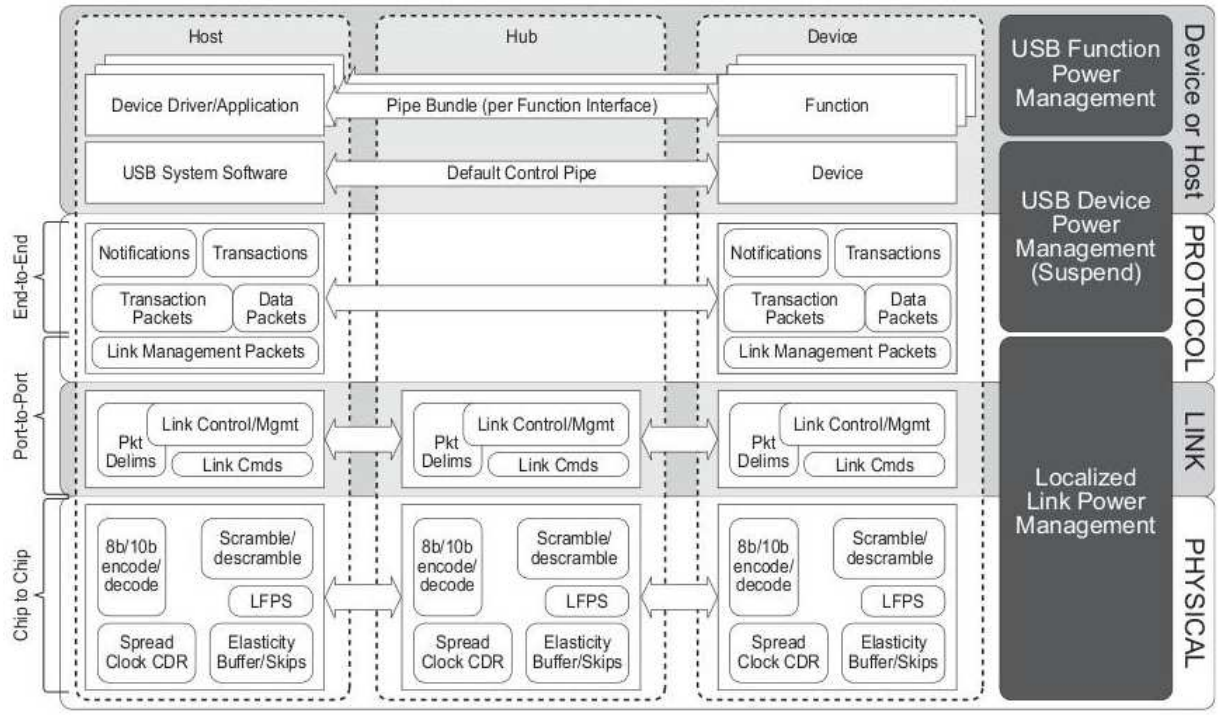
1.3 Superspeed Architecture

[1] Superspeed architecture consists of the following elements.

Superspeed Interconnect: is the manner on which devices are connected and communicate with the host over the Superspeed bus. This includes topology, communication layer and how they interacted to accomplish the data exchange.

Devices: are sources or sinc of information exchanges. They implemented the required device end, Superspeed communication layers to accomplish data exchange between drive on the host and logical functions of the devices.

Host: it holds the Superspeed data activity schedule and management of the Superspeed bus and all the devices connected to it. Figure 1.2 illustrate reference diagram interconnect.



www.usb.org

Figure 1.2 Superspeed communication layer and Power management

Following sections provide architectural overview of each communication layers.

1.4 Physical Layer

[1] The physical layer defines the PHY portion of a port and physical connection between a downstream facing port and upstream facing port of the device. Superspeed physical connection is the comprised of two differential data pairs transmit and receive path. The nominal data rate is 5Gbps.

Electrical aspects of each path are characterized as a transmitter, channel and receiver. These are unidirectional differential link. Each differential link is AC- coupled with capacitors located on the transmitter side. The channels include electrical characteristics of the cable and connectors.

The each differential link is initialized with receiver terminations. The Transmitter is responsible for detecting Receiver at other end of the link. When receiver termination is present but no signaling is occurring then it is consider to be an electrical idle state and in this state, low frequency periodic signal is generated which is simple to generate and consume very little power.

Each PHY has its own clock domain. The USB 3.0 cable does not include a reference clock so bit level timing synchronization relies on the local receiver to align bit recovery clock to the remote transmitter's clock by phase-locking to the signal transitions in the received bit stream.

To ensure the proper transitions occur in the bit stream independent of the data content being transmitted, the transmitter encode the data and special characters using 8b/10b code. Control symbols are used to achieve byte alignment and are used for framing data and managing the link.

1.5 Link Layer

[1] It is logical and physical connection of the two ports. The connected ports are called link partners. A Port has logical portion and physical portion, link layer defines the logical portion. Logical portion include,

- Initialization of physical layer and event management i.e. connect, removal and power management
- State machines and buffering for managing information exchanges. It implements protocol for flow control, reliable delivery of packet headers, and link power management.
- Buffering for data and protocol layer information elements.
- Detect receive packets and error checks for received header packets.
- Provide an appropriate interface to the protocol layer for information exchanges.

Physical portion include,

- Managing state of its PHY i.e. power management and events.

- Transmit and receive byte streams.

1.6 Protocol Layer

[1] It defines end to end communication rules between a host and device. It provides data information exchange between host and device endpoint. This communication relationship is called PIPE. Host determines when application data is transferred between host and device. Device is able to asynchronously request service from the host.

Protocol communications are accomplished via the exchange of packets. These packets are sequence of data with specific control sequence.

Packet headers are the building block of protocol layer. They are fixed in size packets with a type and subtype. A small record within a packet header is utilized by the link layer to manage the flow of the packet from port to port. Packet headers are delivered through the link layer reliably. The remaining fields are utilized by the end –to –end protocol. Application data is transmitted within the data packet payloads. They are encoded with the data packet headers.

1.7 Robustness and Error handling

[1] For the robustness there are several attributes,

- CRC protection for header and data packets.
- Link level header packets to ensure reliable delivery.
- Detection of attach and detach and system level configuration of resource.
- Data and control pipe constructs for ensuring independent interactions between functions.

To provide protection against the bit error each packet includes CRC. The protocol includes separate CRC for header and data packet payloads. Additionally link control word has its own CRC. A failed CRC in the header and link control word consider being a serious error. In such situation link level retry to recover from error. Link and physical layer work together to provide reliable packet header transmission. Physical layer provide error rate and link layer provide error checking. The only way to recover error in hardware is to retry the header packet.

1.8 Power Management

[1] Superspeed provides power management in bus architecture, link, device and function. These areas are in coupled with each other based on allowable power state transitions. Link power management occur asynchrounsly on every link in the connected hierarchy. It depends upon host, device or combination of both. Link power state may be driven by host or downstream port. The link power states are propagated upward by hubs. The decisions to change link power state are made locally. Links that are not being used for communication can be placed in low power state.

The host does not directly control visibility of individual link power state that means one or more link in the path between host and device can be in reduced power state. There are in-band protocol mechanisms that force these link to transition to the operation power state and notify the host that a transition has occurred similarly, a device initiating a communication on the bus with its upstream link in a reduced power state, will first transition its link into an operational state which will cause all links between it and host to transition to the operation state.

The key points of link power management include

- Devices send asynchronous ready notification to the host.
- Packets are routed, allowing links that are not involved in communication to transition into or to remain in low power state.
- Packets that encounter ports in low power state cause those ports to transition out of low power state

Superspeed provides function power management in addition to device power management. For multi functions devices each function can be placed in low power state. The device will transition into the suspended state by the host via a port command. The devices will not automatically transition into the suspended state when all the individual functions within it are suspended.

1.9 Devices

[1] All Superspeed devices share their base architecture with USB 2.0. They are required to carry information for identification and configuration. All devices support one or more pipe through which the host may communicate with the device. All devices must support designated pipe at endpoint zero to which device's default control pipe is attached.

1.10 Peripheral Devices

[1] Must support both Superspeed and at least one other non Superspeed. USB 3.0 devices within a single physical package can consist of a number of functional topologies including single function, multiple functions on a single peripheral device and permanently attached peripheral devices behind an integrated hub.

1.11 Hubs

[1] Hubs provide implementation specific number of downstream ports to which device can attach. Hubs provide additional downstream ports so they provide user with a simple connectivity expansion mechanism for the attachment of additional devices to the USB.

The Superspeed hubs manage the Superspeed portions of the downstream ports. Each physical port has bus-specific control/status registers. A Superspeed hub consists of two logical components: Superspeed hub controller and a Superspeed repeater/forwarder. The hub repeater/forwarder is protocol- control router. It also has hardware support for reset and suspend/resume signaling

Superspeed hubs actively participate in the protocol in several ways including:

- Router out-bound packets to explicit downstream ports
- Aggregates in-bound packets to the upstream port
- Propagates the timestamp packet to all downstream ports not in a low-power state.
- Detects when packets encounter a port that is in a low power state. The hub transitions the targeted port out of the low power state and notifies the host and device that the packet encountered a port in low power state.

1.12 Hosts

[1] Host interacts with devices through a host controller. To support the dual-bus architecture of USB3.0, a host controller must include both Superspeed and USB2.0 elements, which can simultaneously manage control, status and information exchange between the host and devices over each bus.

The host has implementation specific downstream port that includes

- Manage control flow between the host and USB devices
- Manage data flow between the host and USB devices
- Collect status and activity statistics
- Provided power to attached USB devices

USB system software inherits its architectural requirement from USB 2.0 including

- Device enumeration and configuration
- Scheduling of periodic and asynchronous data transfer
- Device and function power management
- Device and bus management information

1.13 Data Flow Models

[1] Data and control exchanges between the host and devices are via sets of either unidirectional or bi-directional pipes.

- Data transfers occur between host software and particular endpoint on a device. The endpoint is associated with particular function on the device. These associations are called pipes.
- Most pipes come into existence when device is configured by system software. However, one message pipes the default control pipe always exist once a device has been powered and is in the default state tom provide access to device configurations, status and control information.
- The pipes support one on or more transfer types

- Bulk transfer has extension for Superspeed called stream which is in-band protocol-level support for multiplexing multiple independent logical data stream through a standard bulk pipe.

Chapter 2

2.1 Superspeed Data Flow Model

[1] Superspeed is very similar to USB2.0 in that it provides communication service between a USB host and attached USB devices. This chapter describes the differences of how data and control information is communicated between a Superspeed host and its attached Superspeed devices. Following concepts explain Superspeed data flow.

- Communication flow models: flows between the host and devices through the Superspeed bus
- Superspeed protocol overview: gives a high level overview of the Superspeed protocol.
- Generalized Transfer Description: provides an overview of how data transfers work in Superspeed and subsequent sections defines the operating constraints for each transfer type.
- Device Notifications: a feature which allows a device to asynchrounsly notifies its host of events or status on the devices.
- Reliability and Efficiency: summarized the information and mechanisms available in Superspeed to ensure reliability and increase efficiency.

2.2 Superspeed Communication Flow

[1] It support endpoints, pipes and transfer types. The endpoints characteristic are reported in the endpoint descriptor and the Superspeed endpoint descriptor.

[1] All Superspeed devices must implement at least the default control pipe. The Superspeed pipe is an association between an endpoint on a device and software on the host. The pipe represents the ability to move the data between the software on the host via the memory buffer and endpoint on the device. The main difference is when non-isochronous endpoint in Superspeed is busy it returns a not ready (NRDY) response and send an end point ready (ERDY) notification when it ready to accept data.

[5] USB 2.0 broadcasts packets to all enabled downstream ports. Every device is required to decode address triple to each packet to determine if it needs to respond. Superspeed unicast the packets, downstream packets are sent over the directed path between the host and targeted devices while upstream packets are sent over the directed path between the device and host. Superspeed packets contain routing information that the hubs use to determine which downstream port the packet needs to traverse to reach the device. Only Isochronous timestamp packet is multicast to all active ports.

USB 2.0 style polling has been replaced with asynchronous notification. The Superspeed transaction is initiated by the host making the request followed by a response from the device. If the device can grant the response it either accepts or sends data. If the end point is halted, the device shall respond with a STALL handshake. If it cannot honor the request it will respond with Not Ready (NRDY) to tell the host that it is not ready to accept or send data. When the device can honor the request, it will send an Endpoint Ready (ERDY) to the host.
To move to unicasting and the limited multicasting of packets together with asynchronous notifications allows link that are not actively passing the packets to be put into reduced power states.

2.3 Superspeed Protocol

[1] It has dual simplex physical layer that support Superspeed along with USB2.0 protocol. However there are some differences with USB 2.0 protocol.
USB 3.0 uses same three part transection like [5] USB 2.0 but for OUT token is incorporated in the data packet and for Ins token is replaced by handshake.

- USB 2.0 does not support bursting
- USB 2.0 is half-duplex broadcast while Superspeed is dual-simplex unicast bus which allow concurrent In and OUT transection.
- USB2.0 uses polling model while Superspeed uses asynchronous notification.
- USB 2.0 does not have streaming capability while Super Speed has streaming capabilities for bulk transfer.

- USB 2.0 has no mechanism for isochronous capable devices to enter in low power USB bus state between service intervals. Superspeed allows these devices to go in low power state between the service intervals.
- USB 2.0 has no mechanism to inform host how much latency the device can handle if the system enters the low power state. USB 3.0 provides this mechanism using Latency Tolerance Messaging.
- USB 2.0 power management, including Link Power Management is always initiated by the host. USB 3.0 supports Link level management that may be initiated from either end of the link.
- USB 2.0 handles transection error detection and recovery and flow control at the end to end level for each transaction. Superspeed splits these functions between the end-to end and link levels.

2.4 Superspeed Packets

[1] Superspeed packets start with a 16-byte header. Some packets consist of header only. All the packets start with the information used to decide how to handle the packet. The header is protected by 16 bit CRC-16 and ends with 2-byte link control word. Depending upon the type all the headers contain the routing information and address triple. The route string is used to direct the packets sent by the host on the directed path through the topology. Hub always forwards packets from downstream ports to upstream ports. There are four types of packets: Link Management Packets, Transaction Packets, Data Packets and Isochronous Timestamp Packets.

- A Link Management Packet (LMP) travels between directly connected ports and is primarily used to manage that link.
- A Transaction Packet (TP) traverses all the links in the path directly connecting the host and a device. It is used to control the flow of the data packets, configure devices and hubs. It does not have a data payload.
- A Data Packet (DP) traverses all the links in the path directly connecting the host and device. It consist two parts: Data Packet Header (DPH) which is similar to TP and a Data

Packet Payload (DPP) which consist of data block plus a 32-bit CRC used to ensure data integrity.

- An Isochronous Timestamp Packet (ITP) is multicast packet sent by the host to all the active links.

2.5 Superspeed Transfer

Non isochronous data packet sent to receiver is acknowledged by the handshake. However, due to separate path for transmit and receive, the transmitter does not have to wait for an explicit handshake for each data packet transferred before sending the next packet.

[1] USB 2.0 uses serial communication which means host starts and complete one bus transection and then start other transection. Superspeed improves by using independent transmit and receive paths. Superspeed USB transection protocol is essentially a split transection protocol that allows more than one OUT bus transection as well as at most one IN bus transection to be active on the bus at the same time. The order device respond is based on endpoint. The order device respond to ACK or DPs is implementation specific.

[5] The USB 2.0 protocol completes an entire IN or OUT transection before continuing to the next bus transection. All the transmission from the host broadcast on the USB 2.0 bus. In Superspeed protocol does not broadcast any packets. The host starts all transactions by sending handshakes or data and devices respond with either data or handshakes. If the device does not have data it will respond with the packets that say it does not have any data to respond. When the device is ready to accept or send data it will send a signal to host that it is ready to resume or start transection. In addition to this Superspeed is also provides mechanism to put the link into low power stage when it is not in used to save power. Devices report the maximum packet size for each endpoint in its endpoint descriptor. The size indicates data payload length only and does not include any of the overhead for link and protocol level. Bandwidth allocation is similar to USB 2.0. Device report the maximum packet size for each endpoint in its endpoint descriptor.

[1] Data Bursting enhances the efficiency by eliminating the wait time for acknowledgment on a per data packet basis. Endpoint on the Superspeed device indicates the number of pockets that it

can send or receive before it has to wait for a handshake. Maximum data burst size is an individual endpoint capability.

[1] The host may change the burst size on per transition basis up to configured burst size. When the endpoint is an OUT, the host can easily control the burst size. When the endpoint is an IN host can limit the burst size on a per-transition basis via a filed in the acknowledgement packet sent to the device.

2.6 In Transfer

[1] The host initiates a transfer by sending acknowledgment packet to the device. This packet contains the addressing information required to route the packet to the intended endpoint. The host tells the device the number of packets it can send and the sequence number of the first data packet expected from the device. In response the endpoint will transmits data packet with the appropriate sequence number back to the host. Host can send up to the number of data to the device without waiting for the acknowledgment packet.

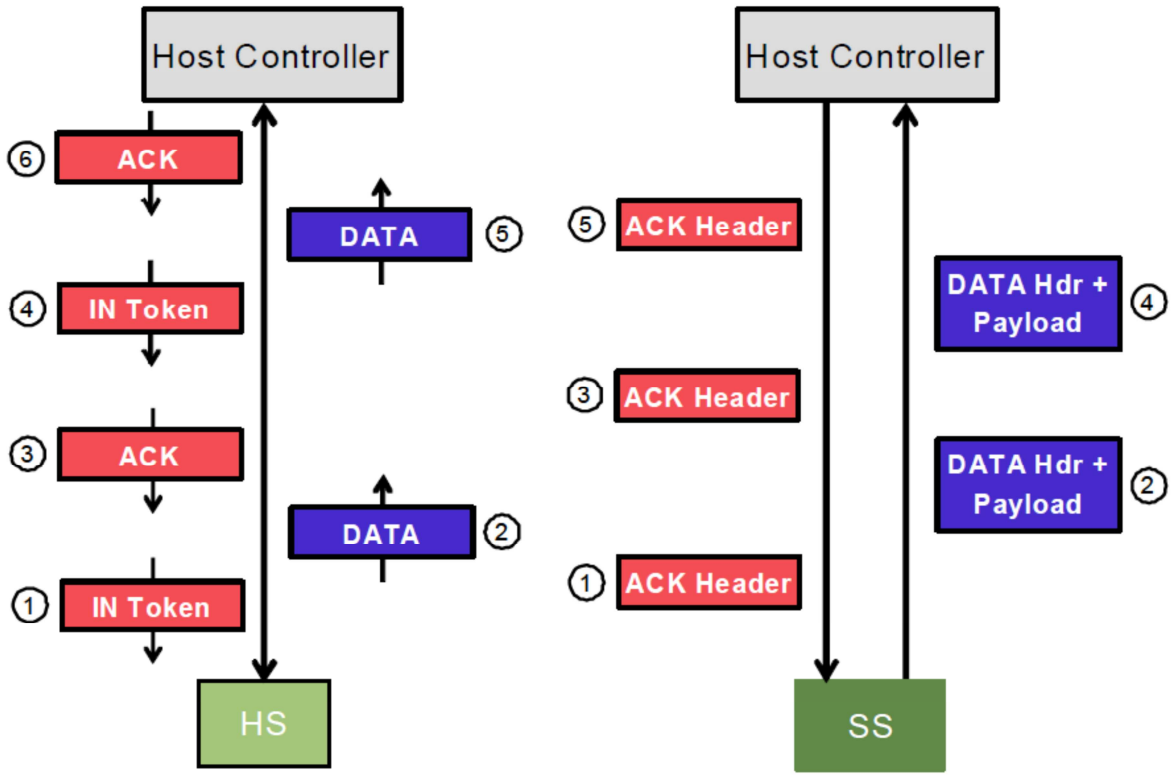
[10] Consider the example IN transection in figure 2.1. The left side indicates the sequence of packets required to perform two back-to-back token/data/handshake transection, requiring 6 packets be exchange as follows.

1. Host broadcasts an IN token Packet (1) to initiate the transection
2. Device returns the requested DATA packet (2)
3. Host acknowledge receipt of data with ACK handshake packet (3)
4. Step 1-3 are repeated

The example on the right indicates the packet sequence needed to perform two back-to-back SS IN transections, which requires only 5 packets be exchanged.

1. SS USB uses an ACK header (packet 1) to initiate an IN transaction.
2. The SS device returns Data (packet2)
3. The second ACK header (3) both acknowledge receipt of the data and request a second transaction
4. The second data packet (4) is delivered by the device.

5. The final ACK header (5) acknowledges receipt of the data, but does not request additional data.



http://www.mindshare.com/files/resources/MindShare_Intro_to_USB_3.0.pdf

Figure 2.1 Two back-to-back transactions USB 2.0 vs. SS

2.7 OUT Transfer

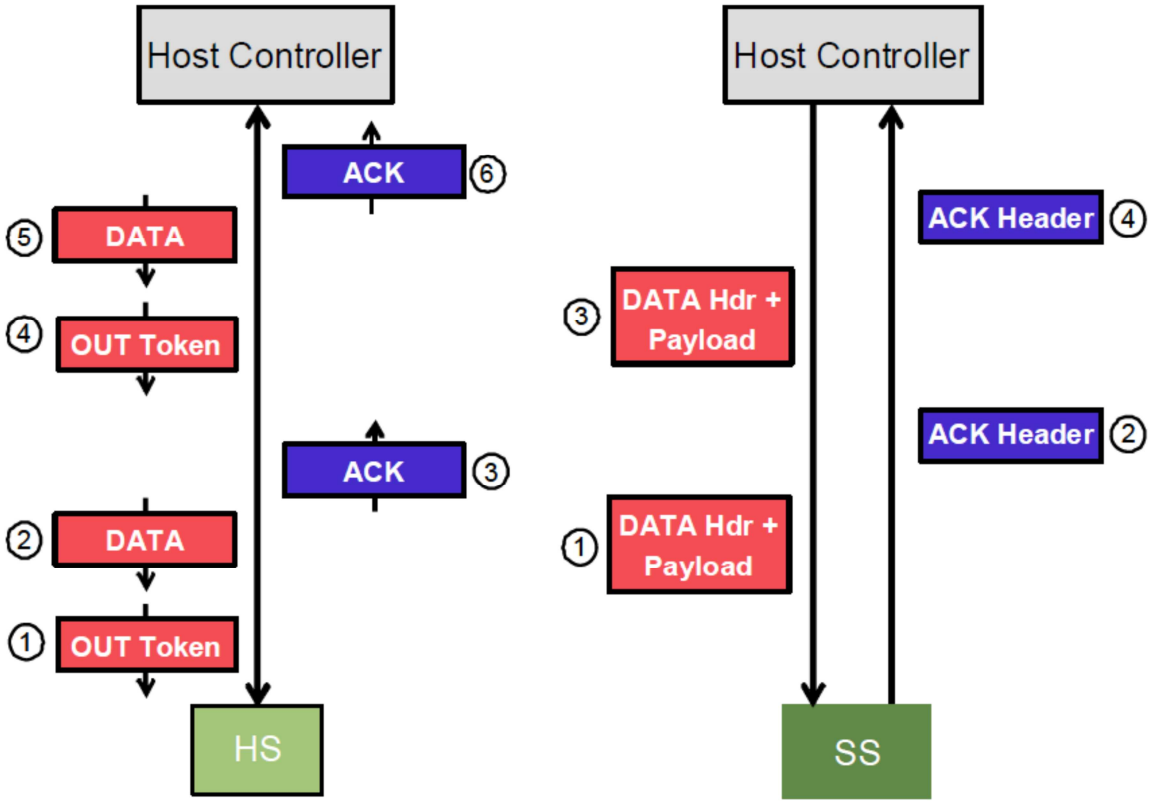
[10] Difference between USB 2.0 and SS OUT transaction are shown in figure 2.2. The example on the left depicts two back-to-back OUT transactions that require 6 packets:

1. Host broadcast an OUT Token packet (1) to initiate the transaction
2. Host sends DATA packet (2) to the device.
3. Device acknowledges receipt of data with an ACK handshake packet (3).
4. Step 1-3 are repeated

The right side of figure 2.2 indicates the packet sequence required to perform two back-to back SS OUT transactions, but requires only 4 packet be exchanged.

1. SS USB uses a data header (packet 1) to initiate an OUT transaction and to deliver data to the device.

2. Device acknowledges receipt of data via an ACK packet (2).
3. The Second data packet (3) initiates the second transaction and delivers data to the device.
4. Device acknowledges receipt of data via ACK packet (4), completing the sequence.



http://www.mindshare.com/files/resources/MindShare_Intro_to_USB_3.0.pdf

Figure 2.2 Two back-to-back transactions USB2.0 vs. SS

2.8 Control Transfers

[1] Each device is required to implement the default control pipe as a message pipe. The pipe is intended for device initialization and management. The pipe is used to access device descriptors and to make requests of the device to manipulate its behavior.

Control endpoint have a fixed maximum control transfer data payload size of 512 bytes and have maximum burst size of 1. There is no way to indicate the desired bandwidth for control pipe. A

host balances the bus access requirements of all control pipes and pending transaction on those pipes to provide a best effort delivery between the client software and functions on the device. Superspeed requires that bus bandwidth be reserved to be available for use by the control transfers as follows:

- The transactions of a control transfer may be scheduled according to endpoints.
- Retries of control transfers are not give priority over other best effort transactions.
- If there are control and bulk transfers pending for multiple endpoints, control transfers for different endpoints are selected for service is based on host controller implementation.
- When control endpoint delivers a flow control event, the host will remove the endpoint from scheduled endpoints. The host will resume the transfer to the endpoint upon receipt of a ready notification from the devices.

2.9 Bulk Transfers

[1] The bulk transfer type is used to support devices that want to communicate relatively large amounts of data at highly variable times where the transfer can use any available Superspeed bandwidth. It provides access to the Superspeed bus on a bandwidth available basis. It guarantees delivery of data but no guarantee of bandwidth or latency. An endpoint for bulk transfer shall set the maximum data packet payload size in its endpoint descriptor to 1024 bytes. Bulk transactions occur on the Superspeed bus only on a bandwidth available basis.

2.10 Interrupt Transfers

[1] The Superspeed interrupt transfer types are intend to support devices that require a high reliability method to communicate small amount of data with a bounded service interval. It guaranteed maximum service interval and retry of transfer attempt in the next service interval in case of failure. The maximum data packet payload size that it can accept from or transmit on the Superspeed bus is 1024 byte. An endpoint for an interrupt pipe specifies its desired service interval bound via its endpoint descriptor.

2.11 Isochronous Transfers

[1] Superspeed isochronous transfer type is intended to support stream that want to perform error tolerant, periodic transfers within a bounded service interval. It guaranteed bandwidth for transaction attempts on the Superspeed bus with bounded latency and rate through the pipe as long as data is provided to the pipe. Isochronous transaction are attempted each service interval for an isochronous endpoint. Superspeed isochronous pipe is a stream pipe and is always unidirectional. The endpoint description identifies whether a given isochronous pipe's communication flows, two isochronous pipes must be used, one in each direction. Maximum size for isochronous endpoints is 1024 bytes.

2.12 Device Notification

[1] Device notifications are standard method for a device to communicate asynchronous device and bus level event information to the host. This does not map to the pipe model defines for the standard transfer types.

2.13 Reliability

There are several layers of protection used to provide reliable operation.

2.13.1 Physical Layer

Physical layer provides bit error rates less than 1bit in 10^{12} bits.

2.13.2 Link Layer

The Superspeed link layer has mechanism that provide bit error rate less than 1bit in 10^20 bits for header packets. It uses numbers of techniques including packet framing ordered sets. Link level flow control and retries to ensure reliable end-to-end delivery for header packets.

2.13.3 Protocol Layer

The Superspeed protocol layer depends on 32bit CRC to the data payload and timeout coupled with retries to ensure reliability of data.

2.14 Efficiency

[1] Superspeed efficiency depends upon 8b/10b encoding scheme [4], packet structure and framing, link level flow control, and protocol overhead.

Chapter 3

3.1 Physical Layer

Physical layer defines the signaling technology for Superspeed bus. This chapter will define Superspeed physical layer.

[1] The PHY interface for the PCI Express and USB Superspeed architecture is intend to enable the development of functionally equivalent PCI Express and USB Superspeed PHY's the specification defines the set of PHY function that must be incorporated in a PIPE compliant PHY and it defines a standard interface between PHY and MAC & Link Layer. [6] This spec provides some information about how MAC could use PIPE interface for various LTSSM states and Link states. One of the intents of pipe specification is to accelerate PCI Express endpoint and USB Superspeed device development.

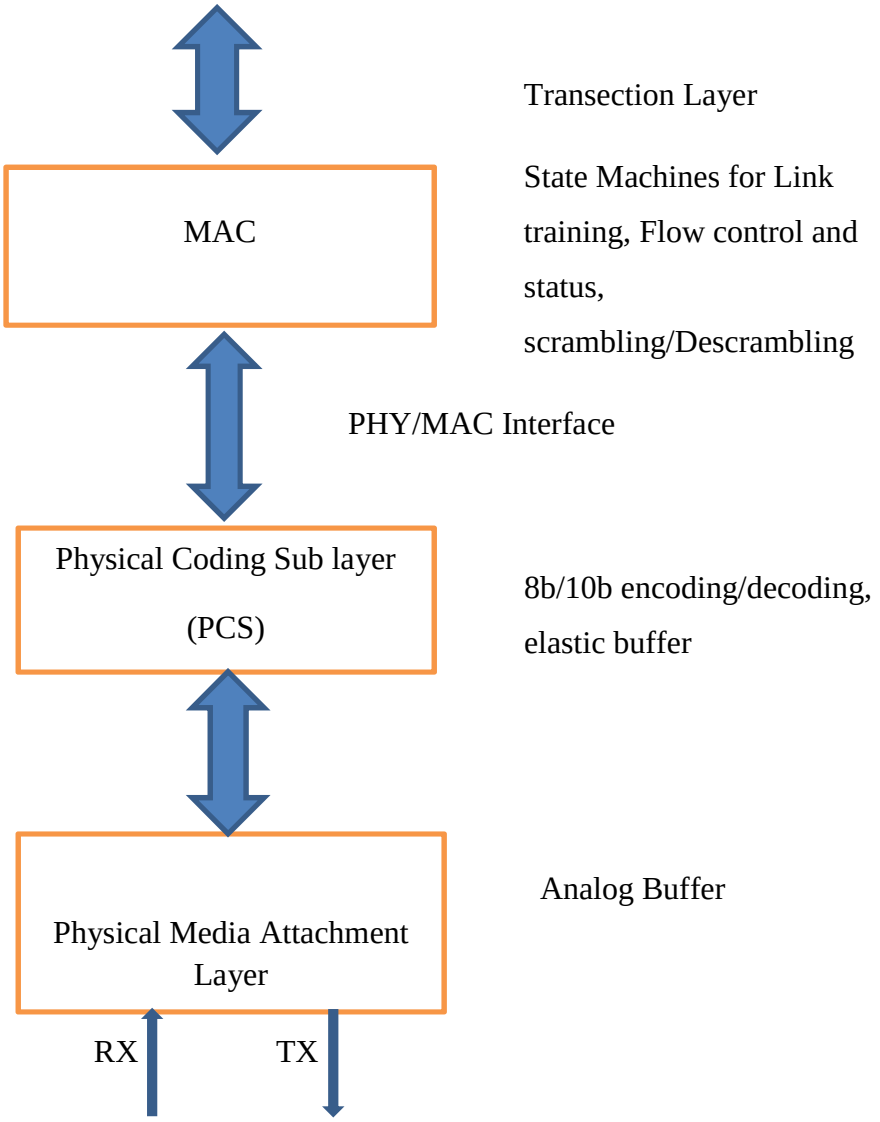


Figure 3.1 Partitioning PHY Layer for USB Superspeed

3.2 PCI Express PHY Layer

[6] It handles low level PCI Express protocol and signaling. The primary focus is to shift the clock domain of the data from the PCI Express rate to one that is compatible with general logic in the ASIC.

It includes following features

- Standard PHY interface enables multiple IP Sources.

- Supports 2.5GT/s only or 2.5GT/s and 5.0GT/s serial data transmission rate.
- Utilize 8bits, 16bits or 32bits parallel interface to transmit and receive
- Allow integration of high speed components into single functional block as seen by the endpoint device manager.
- Data and clock recovery from serial stream on the PCI Express bus.
- Holding registers to transmit and receive data.
- Support direct disparity control for transmitting compliance patterns.
- 8b/10b encode/decode and error indication.

3.3 USB Superspeed PHY Layer

[6] Superspeed PHY Layer handles the low level USB Superspeed protocol and signaling. It includes following features,

- Standard PHY interface enables multiple IP Sources.
- Supports 5.0GT/s serial data transmission rate.
- Utilize 8bits, 16bits or 32bits parallel interface to transmit and receive
- Allow integration of high speed components into single functional block as seen by the endpoint device manager.
- Data and clock recovery from serial stream on the PCI Express bus.
- Holding registers to transmit and receive data.
- Support direct disparity control for transmitting compliance patterns.
- 8b/10b encode/decode and error indication.

3.4 PHY/MAC Interface

[6] Figure 3.2 shows the data and logical command/status signals between the PHY and the MAC layer. Full support of PCI Express mode requires 12 control and 6 status signals. Full Support of Superspeed requires 16control signals and 7 status signals.

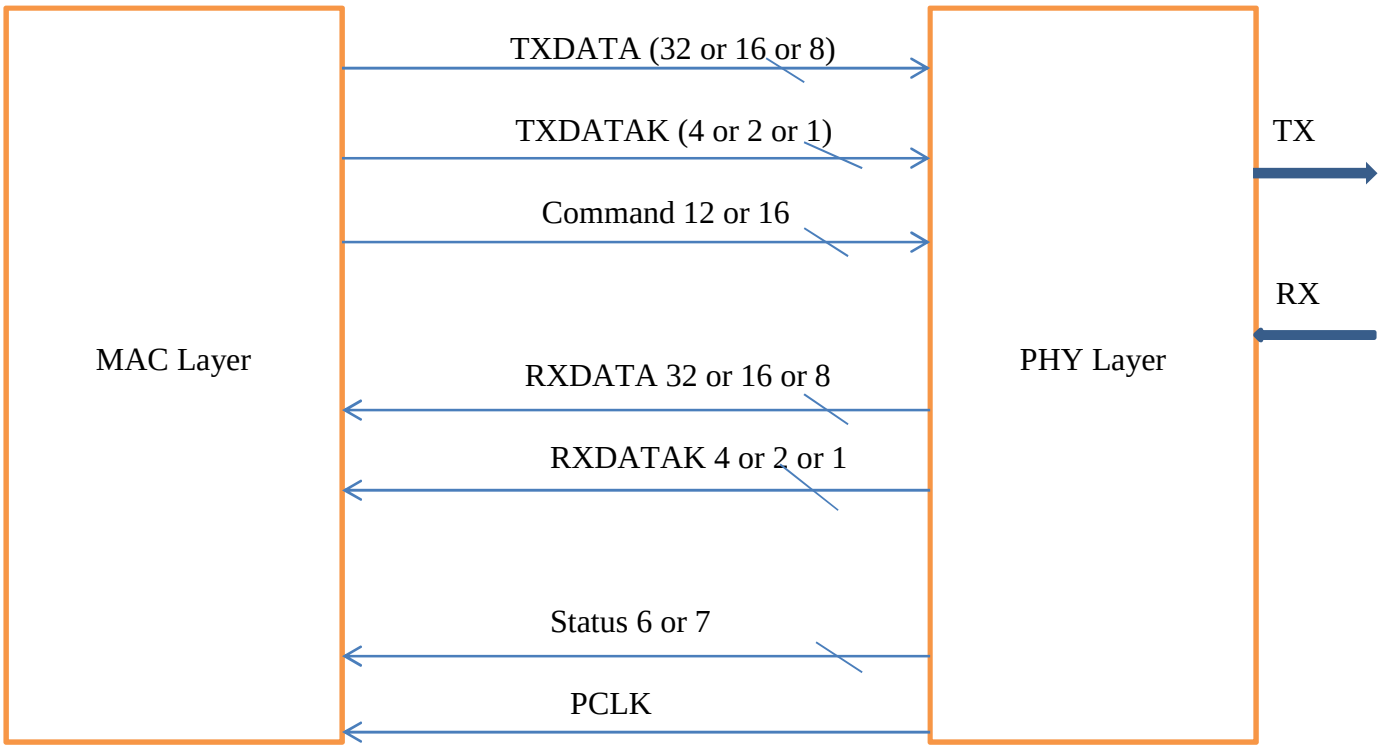


Figure 3.2 PHY/MAC Interface

[6] This specification allows different PHY/MAC interface configuration to support various signaling rates. For pipe PCI Express mode can choose 16bit data paths with PCLK running at 125MHz, or 8bit data path with PCLK running at 250MHz. In the design PCLK is running at 250MHz for 8bit data. Pipe implementation that support 5.0GT/s signaling and 2.5GT/s signaling in PCI Express mode, and therefore are able to switch between 2.GT/s and 5.0GT/s signaling rates PCLK should be 250MHz for 8bit and 16bit data. Other way is to fixed the data rate i.e. 8bit data and changing the PCLK frequency. For 2.5GT/s PCLK frequency set for 250MHz and for 5.0GT/s PCLK frequency should be set to 500MHz.

3.5 Transmitter Block Diagram

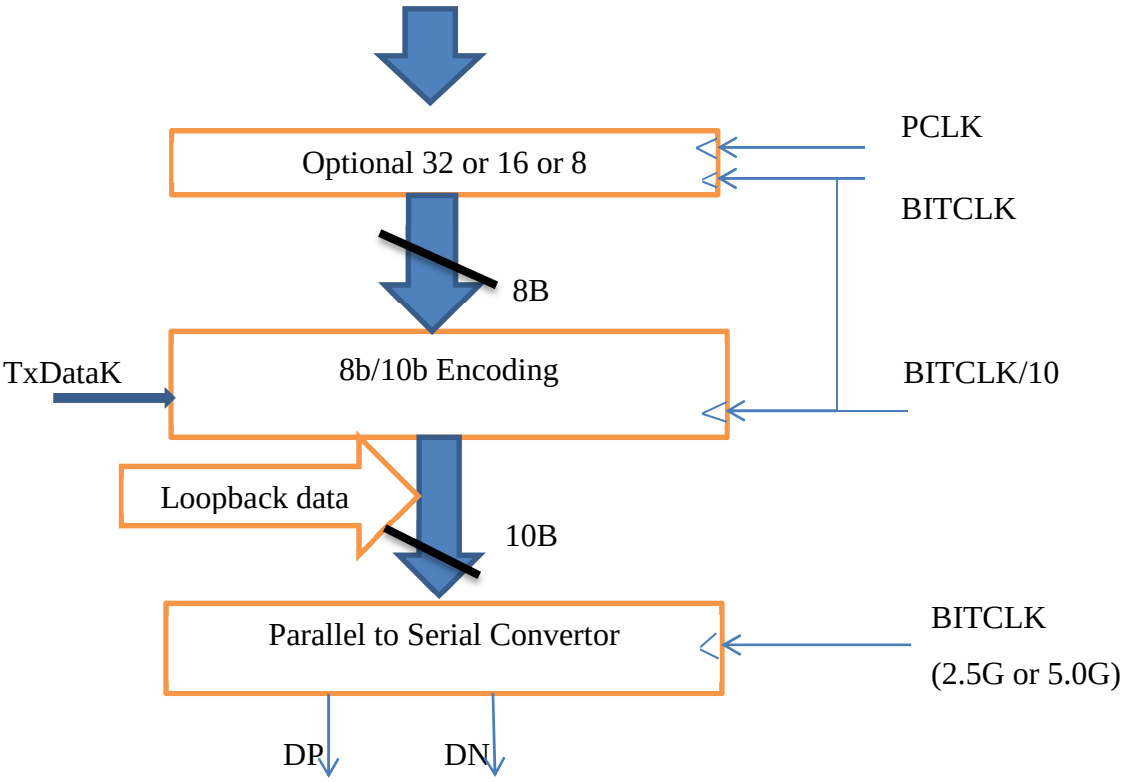


Figure 3.3 Transmitter Block Diagram

3.6 Receiver Block Diagram

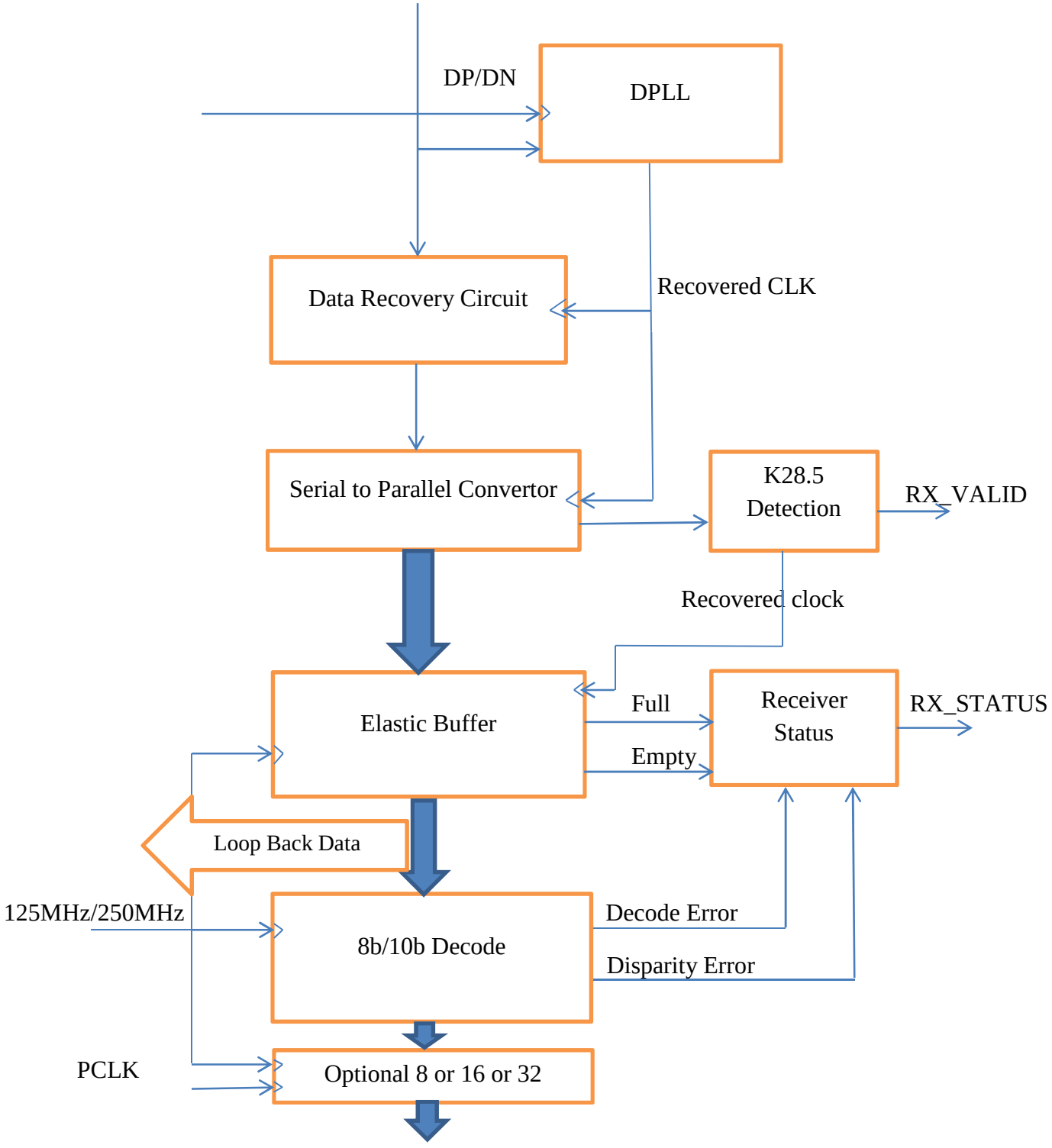


Figure 3.4 Receiver Block Diagram

3.7 PHY/MAC Interface Signals

Following are the Interface signals for PHY/MAC.

Table 3.1 Transmit Data Interface Signal

Name	Width	Direction	Description
DP/DN	1	Output	PCI and Superspeed Differential outputs
TX_DATA	8	Input	Parallel PCIExpress or Superspeed data input bus.
TX_DATAK	1	Input	Data control for symbol or transmit data
RX_DATA	8	Output	Parallel PCI Express output bus
RX_DATAK	1	Output	Data control bit for received symbol

Table 3.2 Command Interface Signals

Name	Width	Direction	Description
PHY_MODE	2	Input	0 For PCI Express 1 For USB SS 2 Reserved 3 Reserved
TX_ELECIDLE	1	Input	
RX_POLARITY	1	Input	0 PHY does no polarity inversion 1 PHY does polarity inversion
PHY_RST	1	Input	Reset the Transmitter and Receiver
POWER_DOWN	2	Input	For PCI Express 00 P0s,NormalOperation 01 P0, Low recovery time latency, Power saving

			state 10 P1, Long recovery time latency, low power state 11 P2, Lowest Power state For USB Superspeed 00 P0,NormalOperation 01 P1, Low recovery time latency, Power saving state 10 P2, Long recovery time latency, low power state 11 P3, Lowest Power state
PHY_RATE	1	Input	Control the link signaling rate 0 use 2.5GT/s signaling rate 1 use 5.0GT/s signaling rate

Table 3.3 Status Interface Signal

Name	Width	Direction	Description
RX_VALID	1	Output	Indicates symbol lock and valid data on RX_DATA and RX_DATAK
PHY_STATUS	1	Output	Use to communicate completion of several PHY function.
RX_STATUS	3	Output	000 Received data OK 100 Both 8B/10B decode error and Receiver Disparity error 110 Elastic Buffer underflow 101 Elastic Buffer overflow

Table 3.4 External Signals

Name	Width	Direction	Description
PHY_CLK	1	Input	This is used to generate BIT_CLK and PCLK

PCLK	1	Output	All data movement across parallel interface are control by PCLK

3.8 Pipe Operation Behavior

3.8.1 Clocking

[6] There are two clock signals, the first PHY_CLK is the reference signal that PHY uses to generate internal bit rate clock for transmitting and receiving data. The specification for this signal is implementation dependent for the design the frequency for the bit rate clock is 2.5GHz. The specification may vary for different operating modes of the PHY. The second clock PCLK is an output from the PHY and is the parallel interface clock used to synchronize data transfer across the parallel interface. This clock runs at 125MHz or 250MHz depends upon the PHY_MODE and PHY_RATE.

3.8.2 Reset

[6] When reset is asserted the MAC must hold the PHY reset until PHY_CLK to the PHY are stable. For this reason AASD is used in the circuit to hold the reset. The PHY signals that PCLK is in the specific power state by the desertion of PHY_STATUS.

3.8.2 Power Management

[6] Power management for PCI Express and USB Superspeed mode is almost identical. There are 4 power states in each mode for the PCIExpress mode these stages are P0, P0s, P1 and P2. When directed from P0 to lower power state, PHY can immediately take whatever powers saving measures are appropriate.

In states P0, P0s, and P1 the PHY is required to keep the PCLK operational. For all state transitions between these states, the PHY indicates successful transition into the designated power state by assertion of PHY_STATUS signal. [6]

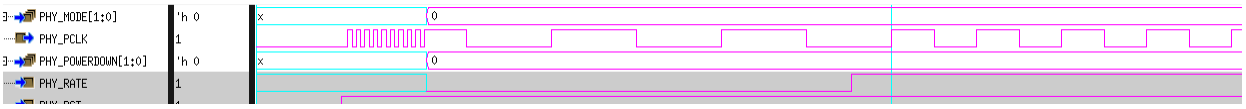
Mapping of PHY power states to states in the Link Training and Status State Machine (LTSSM) found in the base specification are included below

P0 state: all internal clocks in the PHY are operational. P0 is the only state where the PHY transmit and receives PCI Express signaling.

P0s state: PCLK output must stay operational. The MAC may move PHY to this state only when the transmit channel is idle. While the PHY is in eitherP0 and P0s power states, if the receiver is detecting electrical idle, the receiver portion of the PHY can take appropriate power saving measures.

P1 state: Selected internal clocks in the PHY can be turned off. PCLK output must stay operational. The MAC will move the PHY to this state only when both transmit and receive channel idle.

P2 state: Selected internal clock in the PHY can be turned off. The parallel interface is in an asynchronous mode and PCLK output is turned off. In the design PCLK is given a BIT_CLK frequency when it is in P2 state.



Power management for the USB Superspeed mode is same as the PCIExpress mode accept P0, P0s, P1 and P2 are replaced by P0, P1, P2, and P3. In the design there are declare as parameter. There values are 00, 01, 10, and 11 respectively.

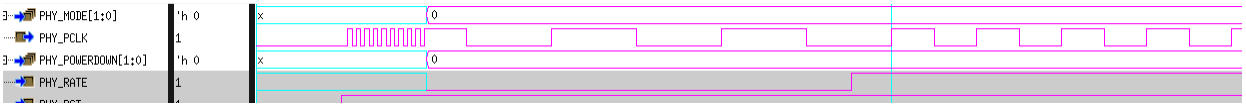
3.8.4 Changing the Signal Rate

[6] Signaling rate of the link can be changed only when the PHY is in the P0 or P1 power state and TX_ELECIDLE is asserted. When MAC changes the PHY_RATE signal, the PHY performs the rate change and signals its completion with signal cycle assertion of PHY_STATUS. The MAC must not perform any operational sequences, power state transitions, deassert TX_ELECIDLE or further signaling rate changes until PHY has indicated that the signaling rate change has completed.

There are two ways,

1. Fixed data path implementations
2. Fixed PCLK implementation

In the design 1st approach is been used when MAC assert the TX_ELECIDLE and changing the PHY_RATE signal while POWER_DOWN is in P0 or P1 or P2 stage, PCLK will change its frequency.

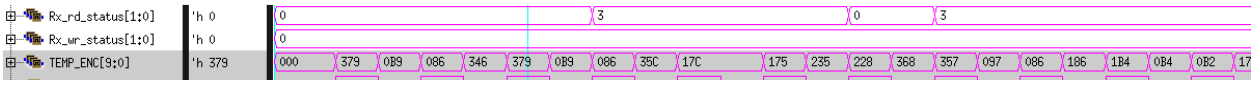


MAC indicates change in PCLK frequency with assertion of PHY_STATUS signal.

3.8.5 Clock Tolerance Compensation

[6] The PHY receiver has an elastic buffer used to compensate for differences in frequencies between bit rates at the two ends of the link. The elastic buffer must be capable of holding enough symbols to handle worst case differences in frequency. Two models are defined for the elastic buffer operation in the PHY. The PHY may support one or both of these models. The Nominal empty buffer model is only supported in the USB Superspeed mode.

Whenever the elastic buffer is in nominal half full buffer or in nominal empty buffer which are indicated by the Rx_rd_status and Rx_wr_status in the design MAC will assert SKP symbol in the PCI Express mode or SKP order SET in USB Superspeed mode or remove SKP symbol or SKP order set. In the design nominal half full buffer will represented by values 2 and nominal empty buffer is represented by value 3.



3.8.6 Error Detection

[6] The PHY is responsible for detecting receive errors. These errors are signal by the MAC layer using the receiver status signal i.e. RX_STATUS signal. Because of the higher level error detection mechanism like CRC built into the data link layer there is no need to specifically identify symbol with errors. When receiver error occurs, the appropriate error code is asserted. There are 4 error conditions that can be encoded on the RX_STATUS signals. The error should be signaled with the priority shown below

1. 8b/10b decode error
2. Elastic buffer overflow
3. Elastic buffer underflow
4. Disparity Errors

For a detected 8b/10b decode error, the PHY should place an EDB symbol for PCIe or SUB symbol for USB Superspeed in the data stream in place of bad byte. For the disparity error negation XOR is used in the design to find out encoded data which are not DC balanced. For the elastic buffer errors an underflow should be signaled during the clock cycle or clock cycles when elastic buffer is empty and overflow should be signaled during clock cycle or clock cycles when elastic buffer is full.

3.8.7 Polarity Inversion

PHY must invert received data when RX_INVERSION is asserted. Inversion can happen in many places in the received data or somewhere in the serial path.

3.8.8 Setting Negative disparity

[4] The 8b/10b encoder is designed in such a way that it will generate dc balanced code with positive disparity in one clock cycle and dc balanced code with negative disparity in the next clock cycle. This features in only used in PCI Express mode.

[10] As show in the figure for 8’h02 encoder will generate 10’b0010101101 in one clock cycle and 10’b1101010010 in the third clock cycle.

3.9 Link initialization and training

[1] Training sequences are used for initializing bit alignment, symbol alignment and optimizing the equalization. Training sequence order sets are never scrambled but always 8b/10b encoded. Bit lock refers to the ability of the clock/data recovery circuit to extract the phase and frequency

information from the incoming data stream. Once CDR is properly recovering data bits the next step is to locate and end of 10bit symbol for this purpose the special K-code COMMA is selected from the 8b/10b codes. The bit pattern of the COMMA code is unique, so that it is never found in other data patterns.

Training sequences are composed of ordered sets used for initializing bit alignment, symbol alignment and receiver equalization. The following rules are applied,

- It shall be 8b/10b encoded.
- It shall not be interrupted by the SKP order sets. SKP order set shall be inserted either before or after the training sequences.
- No SKP ordered sets are to be transmitted during the entire TSEQ time.

Table 3.5 Training Sequence Values

Symbol Number	Name	Value
0	K28.5	COM (BC)
1	D31.7	FF
2	D23.0	17
3	D0.6	C0
4	D20.0	14
5	D18.5	B2
6	D7.7	E7
7	D2.0	02
8	D2.4	82
9	D18.3	72
10	D14.3	6E
11	D8.1	28
12	D6.5	A6
13	D30.5	BE
14	D13.3	6D
15	D31.5	BF
16-31	D10.2	4A

3.10 Normative Clock Recovery Function

[1] In order to recover the clock or to generate the clock which locks its phase with incoming serial data DPLL is used design. [16] DPLL is the functional circuit that generates the signals that are phase locked with the external input signals in design which is incoming serial data. Then the serial data is used to synchronize with the output signal as shown in the figure.

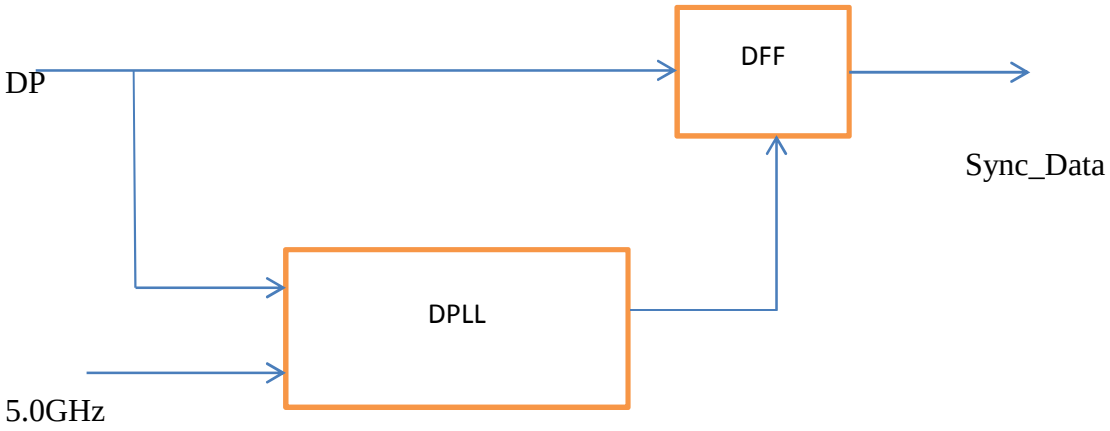


Figure 3.5 Clock recovery and Data recovery circuit

Chapter 4

4.1 Link Layer

[1] This layer is responsible of maintaining the link connectivity so that successful data transfer between two ends of the link is ensured. The link flow is defined based on packets and link commands. Packets are prepared in the link layer, they carry data and different information between host and a device. Link commands are used to communicate between two link partners. Packet frame order sets and link command ordered set are also constructed to tolerant one symbol error. The link layer also facilitates link training, testing/debugging and link power management. This is accomplished by Link training Status State Machine (LTSSM).

Multiple byte fields in packet or a link command are moved over the bus in order where least significant byte (LSB) first and most significant byte (MSB) last. Each byte is encoded in 8b/10b encoding.

4.2 Packets and Packet Framing

[1] Superspeed uses packets to transfer information. These packets are formatted in Link Management Packets (LMP), Transaction Packets (TP), Isochronous Timestamp Packets (ITP), and Data Packets (DP).

4.2.1 Header Packet Structure

[1] All header packets are 20 symbols long, there are formatted in figure 4.1. it consist of three parts a header packet framing, a packet header and link control word. Header packet framing start with HPSTART order set. It is 4 symbols starting with K-symbol. It is defined as three consecutive symbol of SHP followed by a K-symbol of EPF. Header packet is always starts with HPSTART order set.

1	1	CRC	CRC	1	2	3	4	5	6	7	8	9	10	11	12	EPF	SHP	SHP	SHP

Figure 4.1 Header packet framing

4.2.2 Packet Header

[1]Packet header consists of 14 bytes as shown in figure 4.2. It includes 12 bytes of header information and 2 bytes of CRC-16 which is used to protect the data integrity of the 12 byte header information.

CRC	CRC	1	2	3	4	5	6	7	8	9	10	11	12

Figure 4.2 Packet Header

- The polynomial for CRC-16 shall be 100Bh.
- The initial value shall be FFFFh.

4.2.3 Link Control Word

[1]Link control word consists of 2 bytes as shown in figure 4.3. It is used for both link level and end-to-end flow control. It contains 3bit header sequence number, 3 bit reserved, 3bit hub depth index, a delayed bit (DL), a deferred bit (DF) and 5 bit CRC-5.

byte 1	Byte 2

CRC-5	CRC-5	CRC-5	CRC-5	CRC-5	Delay	Delay	DEPTH	DEPTH	DEPTH	RES	RES	RES	HSN	HSN	HSN

Figure 4.3 Link control word

- CRC-5 polynomial shall be 00101b.
- Initial value shall be 11111b.
- It is calculated for the remaining 11 bits of the link control word.

4.2.4 Data Packet Payload

[1]It consists of Data packet header (DPH) and Data packet payload (DPP). DPP framing consist of 8 K-Symbol a 4 symbol DPP starting frame order set and 4 symbol DPP ending frame order set. As shown in the figure 4.4 it start with DPPSATRT order set which is DPP starting frame order set, consist of three consecutive K-symbol of SDP followed by a single K-symbol of EPF.

EPF	END	END	END	CRC	CRC	CRC	CRC	0 to 1024 Data bytes	EPF	SDP	SDP	SDP

Figure 4.4 Data packet payload with CRC-32.

- CRC-32 polynomial shall be 04C11DB7h.
- Initial values shall be FFFF FFFFh.
- CRC-32 shall be calculated for each bytes of the DPP.

4.3 Link Command

[1] Link commands are used for link level data integrity, flow control and link power management. They are fixed length of eight symbols and contain repeated symbols to increase the error tolerance. They are defined for four usage cases. First, link commands are used to ensure the successful transfer of a packet. Second, link command is used for link flow control. Third, link commands are used for link power management. Fourth, link command is special link command defined an upstream port to signal it presence in U0. Successful header packet transaction between the two link partners requires proper header packet acknowledgment.

4.4 Link Error

[1] Data transfer between two ends of the link is in the form of packet. A set of link command is defined to ensure successful packet flow across link. When symbol error occurs on the link, the integrity of a packet or a link command can be compromised. There are various types of errors at link layer. Header packet errors occur due to missing of header packet, invalid header packet due to CRC error, and mismatch of RX Header sequence number. Missing of the header packet shall result in a port transitioning to recovery. Training sequence error happens due to symbol

corruption during the TS1 and TS2 order sets in Polling. Configuration, Recovery. Active and Recovery. Configuration subsets are expected until the requirement are met to transition to the next state. Other 8b/10b error in data encoding error also created when unexpected link commands or header packet is received.

4.5 Link Training and Status State Machine (LTSSM)

[1] LTSSM is the state machine defined for link connectivity and link power management. It consists of 12 different link states that can be characterized based on their functionalities.

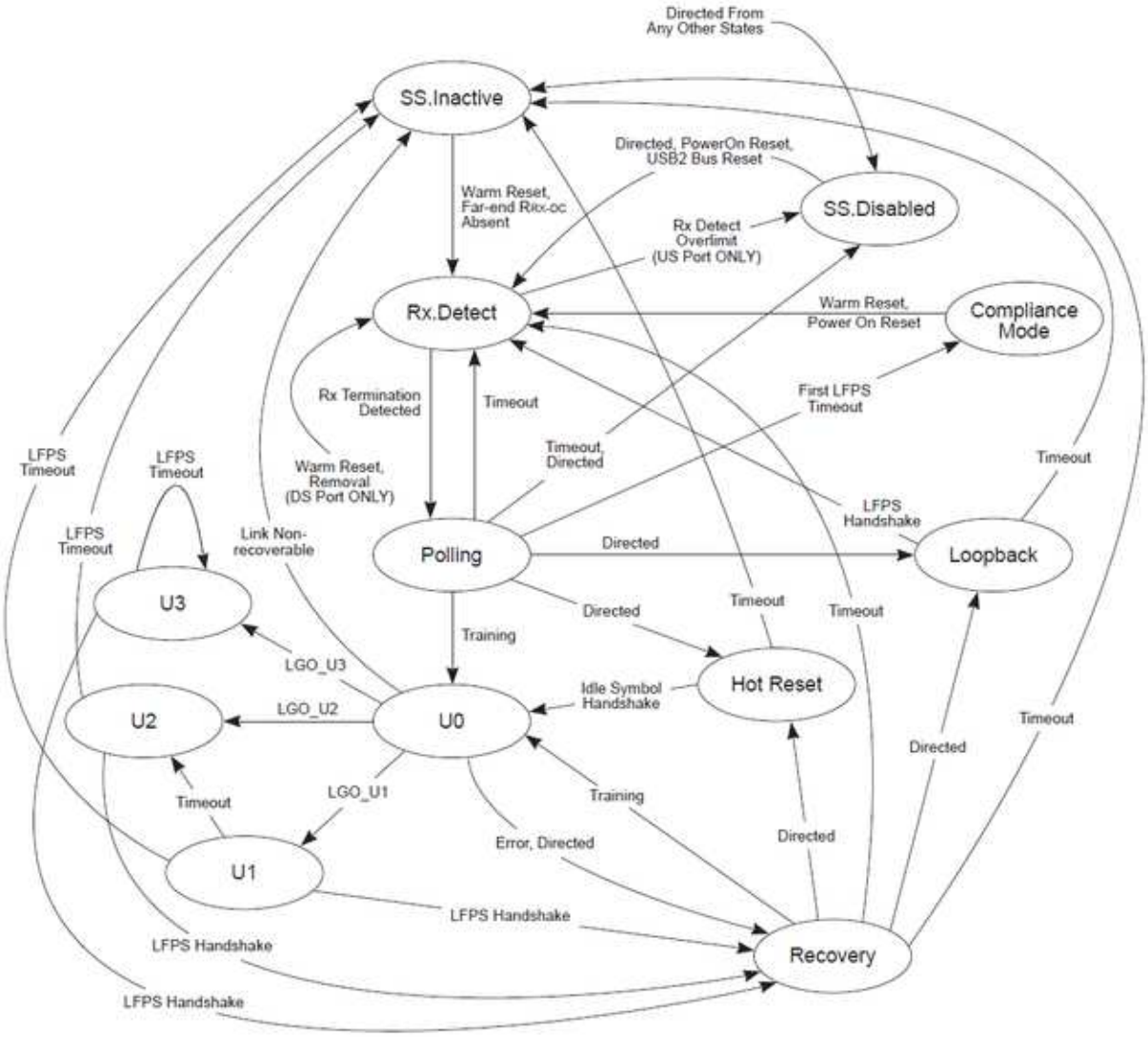
First four operational link states are U0, U1, U2 and U3. In the design there are declare as parameter. U0 (4'h4) is the state where Superspeed link is enable. Packet transfers are in progress or in idle state. U1(4'h5) is low power link state where no packet transfer is carried out and Superspeed link connectivity can be disable to save power. U2 (4'h6) is also low power state but allow more power saving opportunities. U3 (4'h7) is the link suspended state where aggressive power saving is carried out.

Second there are four link states, Rx_Detect, Polling, Recovery and Hot_Reset that are introduced for link initialization and training. Rx_Detect (4'h2) represent initial power on link state where port is attempting to determine the present of link partner, link training process is started. Polling (4'h3) is state defined that two link partners have their Superspeed transmitter and receiver trained, synchronized and ready to transfer data packet. Recovery (4'h9) is the state for retraining the link when two link partner exit from low power state, or when link partners have detected that the link is not operating in U0 state. Hot_Reset (4'h8) is state defined to allow downstream port to reset upstream port.

Third consist of two state LoopBack (4'hA) and Compliance_mode (4'hB) are introduced for bit error test and transmitter compliance test.

SS_Inactive (4'h1) is the link error state where a link is in non-operable state and software intervention is needed. SS_Disable (4'h0) is a link state where Superspeed connectivity is disabled and the link is operated under USB 2.0 mode.

There are also various timers defined and implemented for LTSSM in order to ensure the successful operation but for design purpose they are not taken under consideration.

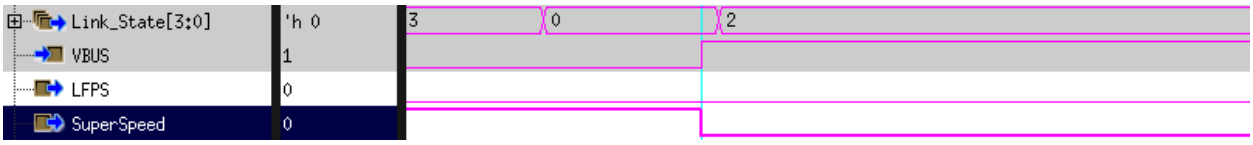


www.usb.org

Figure 4.5 State diagram of the LTSSM

4.5.1 SS_Disable

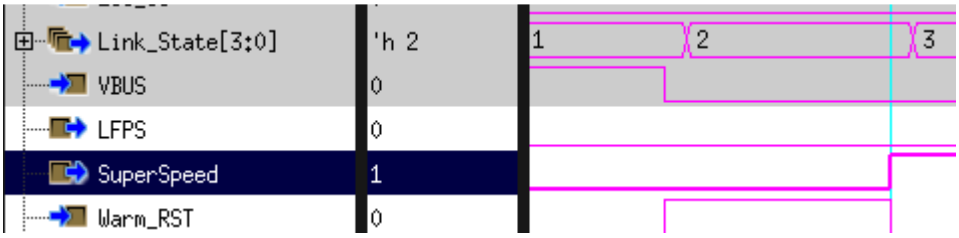
SS_Disable (4'h0) is the state with port's low impedance receiver termination removed and Superspeed connectivity is disabled. In this state VBUS may be present and port shall be disabled from transmitting and receiving LFPS and Superspeed signals.



Downstream port shall be Rx_Detect by enabling VBUS.

4.5.2 SS_Inactive

SS_Inactive (4'h1) is the state where link is failed Superspeed operation. Downstream port and upstream port can only exit from this state when directed i.e. upon Warm_RST. VBUS shall be present



There are 2 sub state of SS_Inactive.

1. SS_Inactive_Disconnect
2. SS_Inactive_Quiet

But for simplicity they are not consider in the design.

4.5.3 Rx_Detect

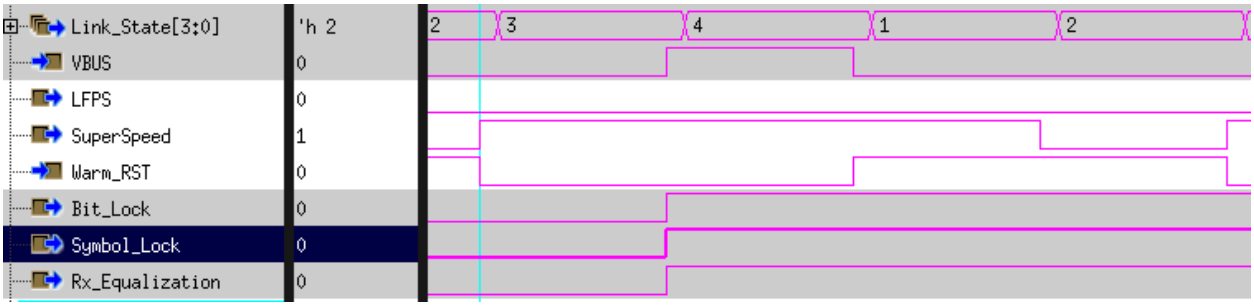
Rx_Detect (4'h2) is the power on state for both downstream and upstream port. It is enabled for downstream port upon receiving of Warm_RST. The purpose of Rx_Detect is to synchronized operation between two ports. This state consist of three subset,

1. Rx_Detect_Reset
2. Rx_Detect_Active
3. Rx_Detect_Quiet

They are not consider in the design. A port will perform the far- end receiver termination periodically during Rx_Detect.

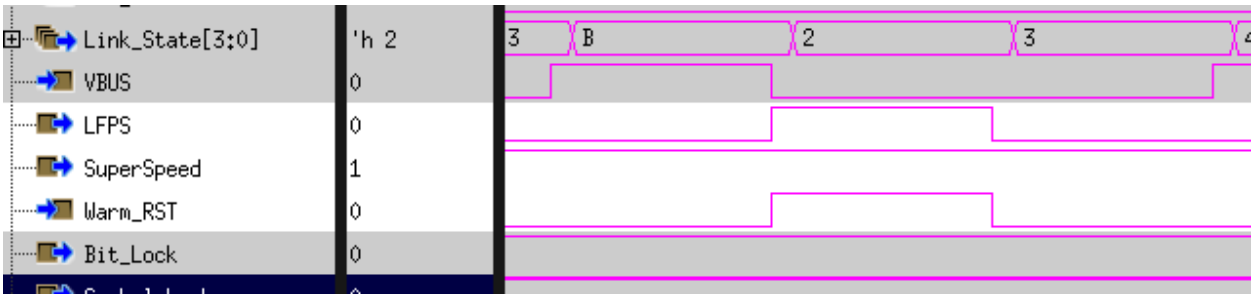
4.5.4 Polling

Polling (4'h3) is the state for link training. In Polling state LFPS handshake shall take place between two ports. Bit_Lock, Symbol_lock and Rx_Equilization are achieved. There are 5 subsets but are not consider for the design purpose.



4.5.5 Compliance Mode

This mode (4'hB) is used to test the transmitter for compliance to voltage and timing specifications. Port will move out from this state upon receiving of Warm_RST.



4.5.6 U0

This (4'h4) is the normal operation state where packets can be transmitted and receive. Port can move from U0 to lower power state when it receives command LG0_U1, LG0_U2 and LG0_U3.

4.5.7 U1

When port is in the U0 state and receives LG0_U1 command from software it will move to state U1 (4'h5).in this state more power is saved. Port will move from U1 to appropriate state upon directed.

4.5.8 U2

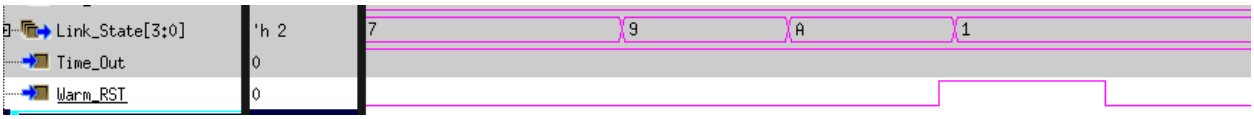
Port will move from U1 to U2(4'h6) state upon timeout. In this state more power is saved compare to U1.

4.5.9 U3

Port will move from U2 to U3 (4'h7) state upon receiving timeout. In this state extreme power saving measurements are carried out.

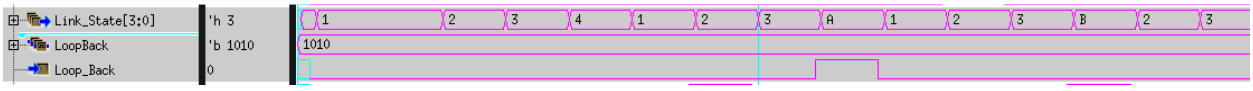
4.5.10 Recovery

This state (4'h9) is used to retain the link or to perform Hot_Reset or to switch to LoopBack mode. In order to retain link and to minimize the recovery latency, the two link partners do not train the receiver equalization. Port will move from Recovery state to other state upon directed.



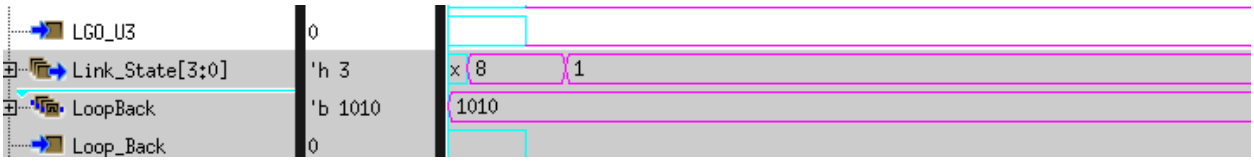
4.5.11 LoopBack

This state (4'hA) is used for test and fault isolation. LoopBack includes a bit error rate test state machine. Port will move from LoopBack to other state upon directed.



4.5.12 Hot_Reset

Only downstream port can initiate Hot_Reset (4'h8). It will reset the whole operation. Port can move to other state when it receive appropriate signal.


LGO_U3
0
Link_State[3:0]
'h 3
x
8
1
LoopBack
'b 1010
1010
Loop_Back
0

Chapter 5

5.1 Design and Simulation

There are two design build in this project

1. Physical Layer
2. Link Layer (LTSSM)

5.2 Physical Layer

5.2.1 PHY.v

This is top module for USB 3.0 Physical layer. It mainly consist two parts Transmitter and Receiver.

There are different sub modules within the top module.

5.2.2 CLOCK_GEN.v

This module is used to generate clock with the different clock frequency i.e. 125MHz, 250MHz and 2.5GHz.

5.2.3 DATA_RATE.v

The main purpose for this module is to generate PCLK depends upon Power state, Rate and PHY mode. A combinational logic is used with nonblocking assignment used so that PCLK will change its value if any of the port from the sensitivity list will change its value.

5.2.4 ENCODER1.v

This module is used to convert the 8b input data into 10b of dc balanced code. This module is designed based on reference []. Instead of deciding +ve or –ve disparity by assigning an additional port. This design will generate +ve and –ve disparity code on the consecutive clock cycle. The proper encoded output is available after one clock cycle.

5.2.5 ClockDiv.v

This module is used to divide the BIT_RATE clock by 10. This clock is used to hold the 10bit encoded data until each bit is serially transmitted using parallel to Serial conversion.

5.2.6 PartoSer.v

This module is used to convert the 10bit encoded parallel data into 1bit serial data. There are different approaches has been taken into consideration one is to use extra 1bit input to load the data. But as encoder will generate proper DC balanced code after delay of 1 clock cycle car has to be taken while giving this input. More the encoder build in design is generating proper outputs after delay of few clock cycles, hence it is difficult to determine when to give load. The other approach is to assign integer and use for loop but as the data is not changing for the whole clock cycle it only sent the last bit. The approach used in the design has two always block one running at BIT_CLK and other at BIT_CLK/10. So the internal temp_reg will get the data at BIT_CLK/10 clock and do the shifting in BIT_CLK. So in the simulation after one clock cycle delay of BIT_CLK/10 the shifting of the encoded data starts.

5.2.7 DPLL2.v

The main purpose of this module is to synchronize the incoming serial data with receiver's local frequency by generating a clock with help of incoming asynchronous data. This module is designed from the reference [11]. This module will compare the output signal with input signal and based on whether the output signal is lagging or leading with input signal based on comparing the edge, it will add or remove clock cycle.

5.2.8 SertoPar.v

This module is used to generate 10bit data from the incoming serial data.

5.2.9 ClockDiv.v

Again this module is used to capture 10bit data. As the receiver accepting single bit data continuously and serial to parallel convertor will continuously generate the 10bit data there are chances that Receiver might receive different data other than transmitted data but in order to save memory and to capture the proper data especially K28.5 this module is used so that output will get input on every 10 clock cycle of BIT_CLK and receiver get proper data.

5.2.10 DFF.v

This module stored the data for 1 clock cycle.

5.2.11 FIFO2.v

FIFO built in ECE527 is used for data bursting. It served as elastic buffer. The additional feature is added to determine the real and write level so that SKP symbol can be added or removed from the incoming package.

5.2.12 RX_STATUS.v

This module will generate proper Rx_status based on the incoming 10bit data or elastic buffer status.

Priority is been given to 8b/10b error by using if statement. Disparity error is find out by negation XOR of 10bit data. If it is DC balanced it will result 1 else it will result 0.

5.2.13 DECODE.v

This is used to convert the 10bit dc balanced data into 8bit data.

5.3 Link Layer

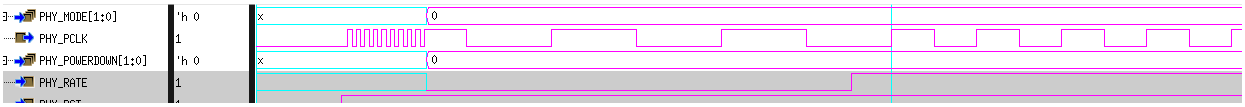
5.3.1 LTSSM.v

This module is basically consists of state machine designed from figure 4.5.

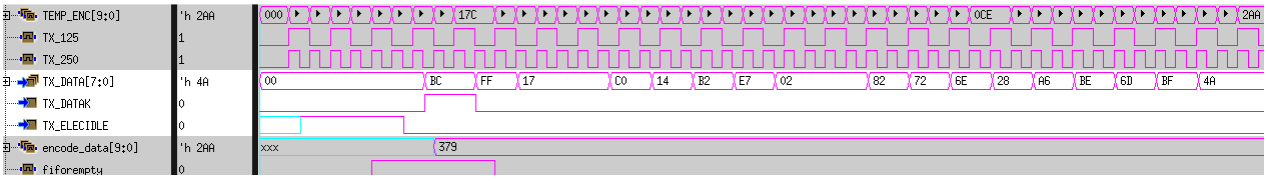
5.4 Simulation

5.4.1 PHY Layer

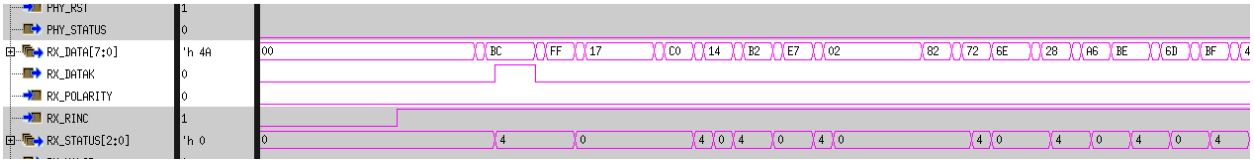
Change in PCLK frequency



Transmitting Training Sequence Data

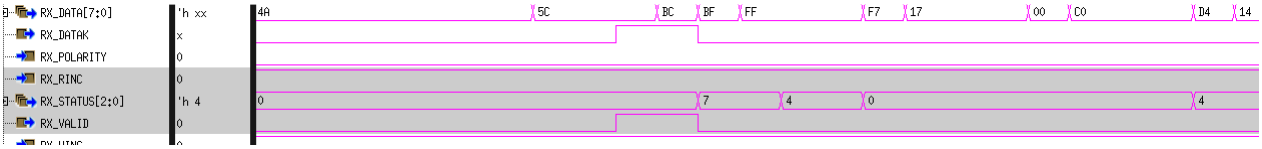


Receiving Training Sequence Data

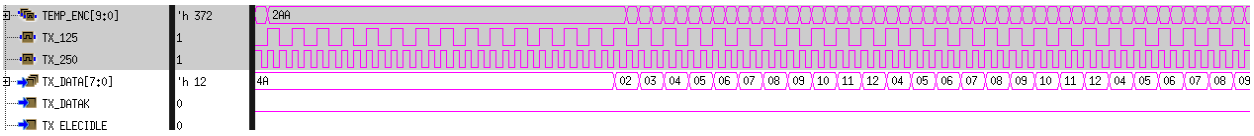


Whole Simulation

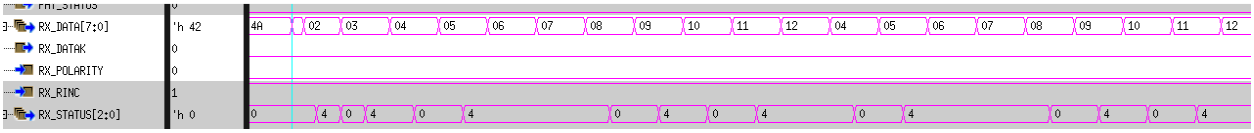
Rx_Status and Rx_Valid



TX_DATA

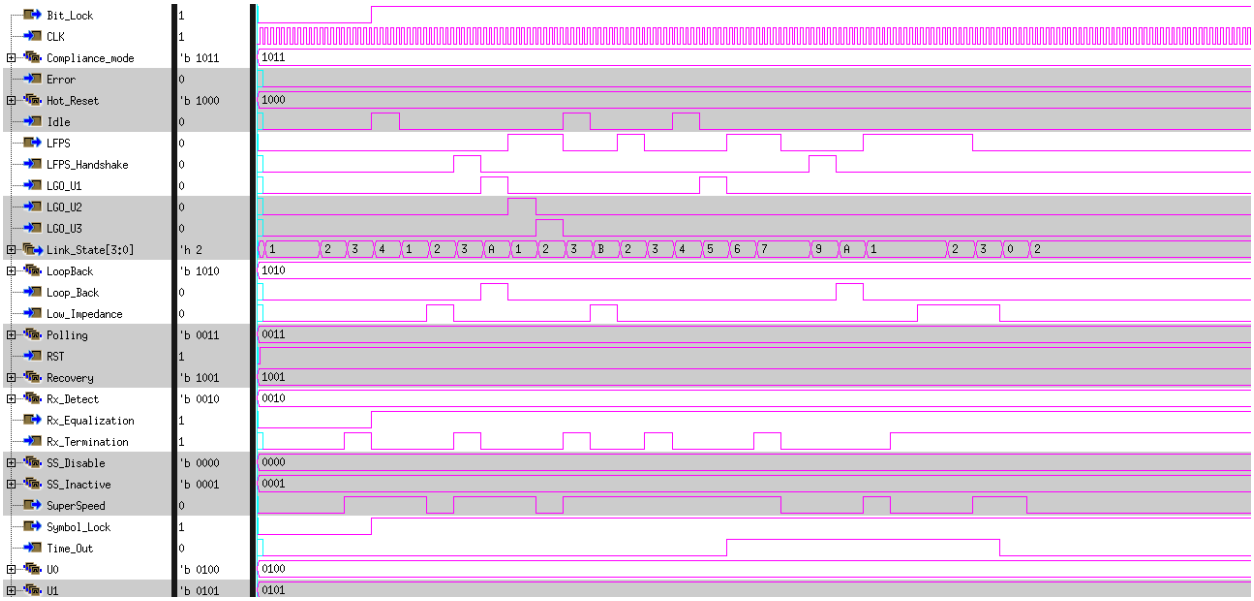


RX_DATA



5.4.2 Link Layer

LTSSM simulation



Chapter 6

Conclusion

Through this project I understand the concept of USB 3.0 and Serial data communication. In this project I am able to transfer data on 2.5GHz clock. I understand the Superspeed features of USB 3.0. Due to data bursting capability USB 3.0 provide far more speed than USB 2.0 as device does not have to wait for the hub's acknowledgment. I also understand 8b/10b encoder and decoder and how they provide a dc balanced data. Along with Superspeed USB 3.0 also provide power management and data integrity through LTSSM and CRC.

Reference

1. Universal Serial Bus 3.0 Specification Revision 1.0
2. www.usb.org
3. Data manual Texas Instruments Literature number: SLLSE16E
4. A DC balanced, partitioned –block, 8B/10B transmission code by A.X. Widmer and P.A. Franaszek
5. Universal Serial Bus 2.0 Specification Revision 1.0
6. PHY interface for the PCI Express and USB 3.0 Architecture
7. Verilog HDL by Sameer Palnitkar
8. www.asic-world.com
9. http://www.mindshare.com/files/resources/MindShare_Intro_to_USB_3.0.pdf
10. Lattice Semiconductor Corporation 8b/10b Encoder/Decoder
11. An All-Digital Phase Locked Loop for high speed clock generation by Ching-Che Chung and Chen-Yi Lee

Appendix

PHY.v

```
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
////////////////////////////////////
*** File Name:PHY.v                    Created By:Shashank Mehta
                                                              ***
***
                                                           ***
***
                                                           ***
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
////////////////////////////////////
*** USB3.0 Physical Layer consist of two main block transmitter and
receiver. It uses 8b/10b Encoder and Decoder for data integrity
***
*** it uses DPLL and data recovery on receiver side to synchronized
data to the local clock. It uses the FIFO for data bursting on
***
*** the receiver side.
                                                        ***
***
                                                        ***
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
////////////////////////////////////*/

`timescale 1ns/100ps

module
PHY(RX_DATA,RX_DATAK,RX_STATUS,RX_VALID,PHY_PCLK,PHY_STATUS,PHY_CLK,PH
Y_POWERDOWN,TX_ELECIDLE,PHY_MODE,PHY_RATE,PHY_RST,TX_DATA,TX_DATAK,RX_
RINC,RX_WINC,PHY_CLR,RX_POLARITY);
output [7:0]RX_DATA;
output RX_DATAK;
output [2:0]RX_STATUS;
output RX_VALID;
output PHY_PCLK,PHY_STATUS;
input [7:0]TX_DATA;
input TX_DATAK;
input[1:0]PHY_POWERDOWN;
input [1:0]PHY_MODE;
input PHY_CLK,PHY_RST,TX_ELECIDLE,PHY_CLR,RX_POLARITY;
input PHY_RATE,RX_RINC,RX_WINC;
wire [7:0]RX_DATA;
wire RX_DATAK;
wire [2:0]RX_STATUS;
```

```
wire RX_VALID;
wire PHY_PCLK,PHY_STATUS;
wire TX_125,TX_250,BIT_CLK,DIV_CLK,DIV_CLK2;
wire [9:0]temp_out;
wire DP;
wire rec_clk;
wire sync_data;
wire [9:0]parallel_data;
wire [1:0]Rx_wr_status,Rx_rd_status;
wire [9:0]temp_rdata,temp_data_buf,encode_data,TEMP_ENC,w_data;
wire status_full,status_empty,fifowfull,fiforempty;

AASD  PHY_AASD(.rst_out(RST),.rst(PHY_RST),.clk(PHY_CLK));
//Transmitter

//Clock generator used to generate 125MHz, 250MHz and 2.5GHz clock
CLOCK_GEN
PHY_CLOCK_GEN(.pclk_125(TX_125),.pclk_250(TX_250),.bit_rate_clk(BIT_CL
K),.clk(PHY_CLK),.rst(RST));

//This module will used to determine PCLK frequency based on PHY mode
and Rate
DATA_RATE
PHY_DATA_RATE(.PCLK(PHY_PCLK),.DATA_STATUS(PHY_STATUS),.DATA_CLK_125(T
X_125),.DATA_CLK_250(TX_250),.DATA_BIT_CLK(BIT_CLK),.DATA_RST(RST),.DA
TA_POWER(PHY_POWERDOWN),.TXELECIDLE(TX_ELECIDLE),.DATA_MODE(PHY_MODE),
.DATA_RATE(PHY_RATE));

//8b/10b encoder used to generate 10b of DC balanced encoded data from
8b data and control data
ENCODER  PHY_ENC(.ENC_OUT(temp_out),.ENC_CLK(DIV_CLK),
.ENC_RST(RST),.KI(TX_DATAK), .ENC_IN(TX_DATA));
assign TEMP_ENC=temp_out;
CLOCK_DIV PHY_CLK_DIV(.BITCLK_10(DIV_CLK),.CLK(BIT_CLK),.RST(RST));

//convert the 10b parallel data generate from 8b/10b into serial data
PartoSerial
PHY_PARSER(.SerialOut(DP),.Serialclk(DIV_CLK),.SerialBit(BIT_CLK),.Ser
ialRST(RST),.Parin(TEMP_ENC));


//Receiver

// Clock and Data Recovery
DPLL_2
PHY_dpll(.DPLL_OUT(rec_clk),.REF_IN(DP),.SYS_CLK(PHY_CLK),.DPLL_RESET(
RST));

// Serial to Parallel conversion
SertoPar
PHY_sertopar(.Parout(parallel_data),.clk(rec_clk),.rst(RST),.Serialin(
DP));
```

```
// to capture 10bit data
CLOCK_DIV PHY_CLK_DIV2(.BITCLK_10(DIV_CLK2),.CLK(rec_clk),.RST(RST));

//to perform rx polarity inversion
DFF
PHY_dff(.out(w_data),.clk(DIV_CLK2),.rst(RST),.in(parallel_data),.inv(
RX_POLARITY));

// For data bursitng
FIFO
PHY_FIFO(.RDATA(temp_rdata),.RD_LEVEL(Rx_rd_status),.WR_LEVEL(Rx_wr_st
atus),.FULL(fifowfull),.EMPTY(fiforempty),.WCLK(DIV_CLK2),.RCLK(TX_125
),.RST(RST),.WDATA(w_data),.CLEAR(PHY_CLR),.WE(RX_WINC),.RE(RX_RINC));

assign temp_data_buf=temp_rdata;
assign status_full=fifowfull;
assign status_empty=fiforempty;
// to generate appropriate RxStatus
RX_STATUS
PHY_rx_status(.RxStatus(RX_STATUS),.RxValid(RX_VALID),.clk(TX_125),.rs
t(RST),.BUFF_full(status_full),.BUFF_empty(status_empty),.DataBuffer(t
emp_data_buf));

// to decode 10b data into 8bit data
assign encode_data=temp_data_buf;
DECODE
rx_decode(.DECODE_OUT(RX_DATA),.KO(RX_DATAK),.DEC_CLK(TX_125),.DEC_RST
(RST),.DATA_IN(encode_data));

endmodule
```

AASD.v

```
/*****************************************************************
***
 ***filename: AASD.v                  Created by:Shashank Mehta
***
 ***
***

*****************************************************************
**
 ***
***
 ***
***
```

```
*************************************************************************
***/
`timescale 1ns/100ps


module AASD(rst_out,rst,clk);//module name
output rst_out;//output port
input rst,clk;//input port

reg rst_out;//internal signal
reg temp;
 //the netlist
always@(posedge clk or negedge rst)
begin
      if(!rst)
      rst_out <=1'b0;
      else
      begin
            temp <=rst;
            rst_out  <= temp;
      end
end
endmodule
```

CLOCK_GEN.v

```
/*************************************************************************
***************
*** File: CLOCK_GEN.v                                Created By:Shashank
Mehta        ***
***
***
*************************************************************************
***************
*** this modual will generate PCLK with freqency 125MHz and 250MHz and
generate    ***
*** BIT_CLK with 2.5GHz of frequency  from 5.0GHz Sys_CLK.
***
***
***
*************************************************************************
**************/

`timescale 1ns/100ps

module CLOCK_GEN(pclk_125,pclk_250,bit_rate_clk,clk,rst);//module name
output pclk_125,pclk_250,bit_rate_clk;//Output port
input clk,rst;                    // Input Port
reg pclk_125,pclk_250,bit_rate_clk;
reg [2:0]count_250;
```

```
reg bist_2,bit_4,bit_6,bit_8;


// to generate BIT_CLK with 2.5GHz
always@(posedge clk or negedge rst)
begin
      if(!rst)
      bit_rate_clk<=1'b0;
      else
      bit_rate_clk<=~bit_rate_clk;
end


always@(posedge bit_rate_clk or negedge rst)
begin
      if(!rst)
      count_250<=0;
      else if(count_250==3'b100)
      count_250<=0;
      else
      count_250<=count_250+1;
end
// to generate PCLK with 250MHz
always@(posedge bit_rate_clk or negedge rst)
begin
      if(!rst)
      pclk_250<=1'b0;
      else if (count_250==3'b100)
      pclk_250<=~pclk_250;
      else
      pclk_250<=pclk_250;
end

// to generate PCLK with 125MHz
always@(posedge pclk_250 or negedge rst)
begin
      if(!rst)
      pclk_125<=1'b0;
      else
      pclk_125<=~pclk_125;
end
endmodule
```

CLOCK_DIV.v

```
/////////////////////////////////////////////////////////////////////////
////////////////
*** File: CLOCK_DIV.v                                  Created By:Shashank
Mehta        ***
***
***
/////////////////////////////////////////////////////////////////////////
////////////////
*** This module wii generate clk for Parrallel to Serial conversion,
As 10bit      ***
*** data are generated from 8b/10b encoder
***
***
***
/////////////////////////////////////////////////////////////////////////
///////////////*/
`timescale 1ns/100ps

module CLOCK_DIV(BITCLK_10,CLK,RST);//module name
output BITCLK_10;                   //Output Port
input CLK,RST;                      //Input Port
reg BITCLK,BITCLK_10;
reg [2:0]temp_reg;

// will invert the BITCLK_10 when temp_reg==3'b100
always@(posedge CLK or negedge RST)
begin
     if(!RST)
     begin
          BITCLK_10<=1'b0;
          temp_reg<=3'b0;
     end
     else
     begin
          //temp_reg<=temp_reg+1;
          if(temp_reg==3'b100)
          begin
          BITCLK_10<=~BITCLK_10;
          temp_reg<=3'b0;
          end
          else
          temp_reg<=temp_reg+1;
     end
end
endmodule
```

DATA_RATE.v

```
/*****************************************************************
***************
*** File: DATA_RATE.v                              Created By:Shashank
Mehta          ***
***
***
*****************************************************************
***************
*** This module will generate PCLK depends upon DATA_RATE and
DATA_POWER           ***
***
***
***
***
*****************************************************************
***************/



`timescale 1ns/100ps

module
DATA_RATE(PCLK,DATA_STATUS,DATA_CLK_125,DATA_CLK_250,DATA_BIT_CLK,DATA
_RST,DATA_POWER,TXELECIDLE,DATA_MODE,DATA_RATE);//module name
output PCLK;
//Output port
output DATA_STATUS;
//Output port
input DATA_CLK_125,DATA_CLK_250,DATA_BIT_CLK;
//Input port
input DATA_RST;
//Input port
input [1:0]DATA_POWER;
//Input port
input TXELECIDLE;
//Input port
input [1:0]DATA_MODE;
//Input port
input DATA_RATE;
//Input port

reg PCLK;
reg DATA_STATUS;

always@(DATA_BIT_CLK or DATA_MODE or TXELECIDLE or DATA_POWER or
DATA_RST or DATA_CLK_125 or DATA_CLK_250)
begin
     if(!DATA_RST)
     begin
```

```
            PCLK=0;
            DATA_STATUS=0;
      end
      else if((DATA_POWER==2'b00) && (TXELECIDLE==1'b1) &&
(DATA_MODE==2'b00))//PCI Express mode
      begin
            if(!DATA_RATE)//2.5GT/s
            begin
                  PCLK=DATA_CLK_125;
                  DATA_STATUS=1'b1;
            end
            else if(DATA_RATE)//5.0GT/s
            begin
                  PCLK=DATA_CLK_250;
                  DATA_STATUS=1'b1;
            end
      end
      else if((DATA_POWER==2'b11) && (TXELECIDLE==1'b0))//P3 state
where extream power saving measures is carried out
            begin
                  PCLK=1'b0;
                  DATA_STATUS=1'b1;
            end
      else // USB SuperSpeed
      begin
            PCLK=DATA_BIT_CLK;
            DATA_STATUS=1'b0;
      end
end
endmodule
```

ENCODER_1.v

```
/*****************************************************************
***************
*** File: ENCODER_1.v                              Created By:Shashank
Mehta        ***
***
***
*****************************************************************
***************
*** This module will generate 10bit of DC balance code from 8bit of
data. This     ***
*** module from the paper presented by A.X Widmer and P.A.Franaszek.
The purpose   ***
*** is to generate DC balanced code with +ve and -ve disparity
***
*****************************************************************
**************/
```

```
`timescale 1ns/100ps

module ENCODER(ENC_OUT,ENC_CLK, ENC_RST,KI, ENC_IN);//module name

output [9:0]ENC_OUT;                                    //Output port
input ENC_CLK,ENC_RST,KI;                               //Input Port with
control signal
input [7:0]ENC_IN;                                      // 8bit input port

reg [9:0]ENC_OUT;
// Internal wires adn regs
wire ai,bi,ci,di,ei;
reg fi,gi,hi,k;
wire aeqb,ceqd,L22,L13,L31,L40,L04;
wire PDL6,PDL4,NDL6;
wire PD1S6,ND1S6,PD0S6,ND0S6;
wire PD1S4,ND1S4,PD0S4,ND0S4;
wire FNEG;
reg S;
reg LPDL6,LPDL4;
wire COMPLS4,COMPLS6;
wire SINT,NFO,NGO,NHO,NJO;
wire NAO,NBO,NCO,NDO,NEO,NIO;

//5b input function
assign ai= ENC_IN[0];
assign bi= ENC_IN[1];
assign ci= ENC_IN[2];
assign di= ENC_IN[3];
assign ei= ENC_IN[4];

//3b input function
always@(posedge ENC_CLK or negedge ENC_RST)
begin
     if(!ENC_RST)
     begin
     fi<= 0;
     gi<= 0;
     hi<= 0;
     k<=0;
end
else
     begin
     fi<= ENC_IN[5];
     gi<= ENC_IN[6];
     hi<= ENC_IN[7];
     k<=KI;
end
end
          assign aeqb= (ai & bi)|(!ai & !bi);
          assign ceqd= (ci & di)|(!ci & !di);
```

```
          assign L22 = (ai & bi & !ci & !di)|(ci & di & !ai &
!bi)|(!aeqb & !ceqd);// 2 1's and 2 0's
          assign L13 = (!aeqb & !ci & !di)|(!ceqd & !ai & !bi);//1
1's and 3 0's
          assign L31 = (!aeqb & ci & di)|(!ceqd & ai & bi); // 3 1's
and 1 0's
          assign L40 = (ai & bi & ci & di);               // all 1's
          assign L04 = (!ai & !bi & !ci & !di);           //all 0's

//Disparity Control
          assign PD1S6= (!L22 & !L31 & !ei)|(L13 & di & ei);
          assign ND1S6= (L31 & !di & !ei)|(ei & !L22 & !L13)|k;
          assign PD0S6= (!L22 & !L13 & ei)|k;
          assign ND0S6= PD1S6;


assign FNEG= fi ^ gi;
// creating S function
always@(posedge ENC_CLK or negedge ENC_RST)
begin
     if(!ENC_RST)
     S<=0;
     else
     S<=(PDL6 & L31 & di & !ei)|(NDL6 & L13 & ei & !di);
end


      assign ND1S4 = (fi & gi);
      assign ND0S4 = (!fi & !gi);
      assign PD1S4 = (!fi & !gi) | (FNEG & k);
      assign PD0S4 = (fi & gi & hi);
      assign NDL6=!PDL6;

     assign PDL6=(PD0S6 & !COMPLS6)|(COMPLS6 & ND0S6)|(!ND0S6 & !PD0S6
& LPDL4);
     assign NDL6=!PDL6;
     assign PDL4=(LPDL6 & !PD0S4 & ! ND0S4)|(ND0S4 &
COMPLS4)|(!COMPLS4 & PD0S4);

//Disparity determine complementing S4
always@(posedge ENC_CLK or negedge ENC_RST)
begin
     if(!ENC_RST)
     LPDL6<= 0;
     else
     LPDL6<=PDL6;
end
//Disparity determine complementing S6
always@(posedge ENC_CLK or negedge ENC_RST)
begin
     if(!ENC_RST)
     LPDL4<=0;
```

```
      else
      LPDL4<=~PDL4;
end


assign COMPLS4= (PD1S4 & !LPDL6) ^ (ND1S4 & LPDL6);
assign COMPLS6= (ND1S6 & LPDL4) ^ (PD1S6 & !LPDL4);

//5b/6b encoder
// Logic for non-complemented Outputs
assign NAO= ai;
assign NBO= L04|(bi & !L40);
assign NCO= ci | L04 | (L13 & di & ei);
assign NDO= di & !L40;
assign NEO= (ei & !(ei & di & L13))|(L13 & !ei);
assign NIO= (L22 & !ei)|(ei & L04)|(ei & L40)|(k & L22)|(ei & !di &
L13);


always@(posedge ENC_CLK or negedge ENC_RST)
begin
      if(!ENC_RST)
      ENC_OUT[5:0]<=6'b0;
      else
      begin
      ENC_OUT[0]<= COMPLS6 ^ NAO;
      ENC_OUT[1]<= COMPLS6 ^ NBO;
      ENC_OUT[2]<= COMPLS6 ^ NCO;
      ENC_OUT[3]<= COMPLS6 ^ NDO;
      ENC_OUT[4]<= COMPLS6 ^ NEO;
      ENC_OUT[5]<= COMPLS6 ^ NIO;
      end
end

//3B/4B encoder
// Logic for non-complimented output


assign SINT= (S & fi & gi & hi)|(k & fi & gi & hi);
assign NFO= (fi & !SINT);
assign NGO= gi | (!fi & !gi & !hi);
assign NHO= hi;
assign NJO= SINT|(FNEG & !hi);

always@(posedge ENC_CLK or negedge ENC_RST)
begin
      if(!ENC_RST)
      ENC_OUT[9:6]<=3'b0;
      else
      begin
      ENC_OUT[6]<= COMPLS4 ^ NFO;
      ENC_OUT[7]<= COMPLS4 ^ NGO;
```

```
      ENC_OUT[8]<= COMPLS4 ^ NHO;
      ENC_OUT[9]<= COMPLS4 ^ NJO;
      end
end
endmodule
```

PartoSer.v

```
/*************************************************************************
***************
*** File: PartoSerial.v                          Created By:Shashank
Mehta        ***
***
***
*************************************************************************
***************
*** This module is used to do Parallel to Serial conversion by
accepting 10bit     ***
*** parallel data on BIT_CLK/10 and do the serial Conversion on
BIT_CLK            ***
***
***
*************************************************************************
***************/
`timescale 1ns/100ps

module
PartoSerial(SerialOut,Serialclk,SerialBit,SerialRST,Parin);//module
name
output SerialOut;
//Output port
input Serialclk,SerialRST,SerialBit;                              //
Input port
input [9:0]Parin;
wire SerialOut;
reg [9:0]temp_reg;

// will accept encoded data on BIT_CLK/10
always@(posedge Serialclk or negedge SerialRST)
begin
      if(!SerialRST)
      temp_reg<=0;
      else
      temp_reg<=Parin;
end

// will perform Parallel to Serial conversion on BIT_CLK
always@(posedge SerialBit or negedge SerialRST)
begin
      if(!SerialRST)
      temp_reg<=0;
```

```
      else
      temp_reg<={1'b0,temp_reg[9:1]};
end

assign SerialOut=temp_reg[0];
```

DPLL2.v

```
/////////////////////////////////////////////////////////////////////
***************
*** File: DPLL2.v                              Created By:Shashank Mehta
***
***
***
/////////////////////////////////////////////////////////////////////
***************
*** This module is used to generate the local clock for the receiver
and lock the  ***
*** phase of the clock with incoming Serial data.
***
***
***
/////////////////////////////////////////////////////////////////////
***************/

`timescale 1ns/100ps

module DPLL_2(DPLL_OUT,REF_IN,SYS_CLK,DPLL_RESET);//module name
parameter filterlength=8;
parameter filterreset=4;
parameter filtermax=filterreset;
parameter filtermin=256-filterreset;
parameter dividerlength=7;
parameter dividermaxvalue=48;

output DPLL_OUT;                                    //Output Port
input REF_IN,SYS_CLK,DPLL_RESET;                    //Input port

reg DPLL_OUT;
reg lead,lag;
reg [1:0]signal_edgedetect;
wire signal_edge;
reg [filterlength-1:0]filtercount;
reg positive,negative;
reg [dividerlength-1:0]dividercount;

//Detecting the rising edge
always@(posedge SYS_CLK or negedge DPLL_RESET)
begin
      if(DPLL_RESET)
      signal_edgedetect<=0;
```

```
      else
      signal_edgedetect<={signal_edgedetect[0],REF_IN};
end

// This signal checked at the rising edge of the main clk
//it is simple detector of input signal rising edge
//when it detect level of output must be checked
assign signal_edge=(signal_edgedetect==2'b01);

// "lead signal will be generated in case of output==1 during input
rising edge
always@(posedge SYS_CLK or negedge DPLL_RESET)
begin
      if(!DPLL_RESET)
      {lead,lag}<=2'b00;
      else if((signal_edge==1'b1) && (DPLL_OUT==1'b0))
      lead<=1'b1;
      else if((signal_edge==1'b0) && (DPLL_OUT==1'b1))
      lag<=1'b1;
      else
      {lead,lag}<=2'b00;
end

always@(posedge SYS_CLK or negedge DPLL_RESET)
begin
      if(!DPLL_RESET)
      filtercount<=0;
      else if((filtercount==filtermax) || (filtercount==filtermin))
      filtercount<=0;
      else if(lead)
      filtercount<=filtercount+1;
      else if(lag)
      filtercount<=filtercount-1;
      else
      filtercount<=filtercount;
end

always@(posedge SYS_CLK or negedge DPLL_RESET)
begin
      if(DPLL_RESET)
      {positive,negative}<=2'b00;
      else if(filtercount==filtermax)
      positive<=1'b1;
      else if(filtercount==filtermin)
      negative<=1'b1;
      else
      {positive,negative}<=2'b00;
end

always@(posedge SYS_CLK or negedge DPLL_RESET)
begin
      if(!DPLL_RESET)
```

```
      dividercount<=0;
      else if(dividercount==dividermaxvalue-1)
      dividercount<=0;
      else if(positive)
      dividercount<=dividercount;
      else if(negative)
      dividercount<=dividercount+1;
      else
      dividercount<=dividercount;
end

always@(posedge SYS_CLK or negedge DPLL_RESET)
begin
      if(!DPLL_RESET)
      DPLL_OUT<=1'b0;
      else if(dividercount==0)
      DPLL_OUT<=~DPLL_OUT;//Additional divider by 2-for producing 50%
duty factor of the output signal
      else
      DPLL_OUT<=DPLL_OUT;
end

endmodule
```

FIFO1.v

```
/**********************************************************************
***************
*** File: FIFO1.v                              Created By:Shashank Mehta
***
***
***
**********************************************************************
***************
*** This module is used for data bursting. It is the same FIFO which
is build up   ***
*** in ECE527 LAB4 with an extra features which will indicates read
and write      ***
*** level so that SKP symbol can be add or removed.
***
**********************************************************************
**************/

`timescale 1ns/100ps

module
FIFO(RDATA,RD_LEVEL,WR_LEVEL,FULL,EMPTY,WCLK,RCLK,RST,WDATA,CLEAR,WE,R
E);//Module Name
parameter width=10;
parameter addr=16;
```

```
output [width-1:0]RDATA;
//Output Port
output [1:0]RD_LEVEL,WR_LEVEL;
//read andwrite level
output FULL,EMPTY;
//Output Port
input WCLK,RCLK,RST,CLEAR,WE,RE;
//Input port
input [width-1:0]WDATA;
//Input Port

wire [width-1:0]RDATA;
wire [1:0]RD_LEVEL,WR_LEVEL;
wire FULL,EMPTY;
wire [addr+1:0]raddr,waddr;
wire read_rst,write_rst,read_clr,write_clr;


POINTER
FIFO_POINTER(.rd_status(RD_LEVEL),.wr_status(WR_LEVEL),.rp_bin(raddr),
.wp_bin(waddr),.full(full),.empty(EMPTY),.wclk(WCLK),.wrst(write_rst),
.rclk(RCLK),.rrst(read_rst),.wr_clr(write_clr),.rd_clr(read_clr),.winc
(WE),.rinc(RE));

RESET
FIFO_RESET(.rd_rst(read_rst),.wr_rst(write_rst),.wr_clr(write_clr),.rd
_clr(read_clr),.wclk(WCLK),.rclk(RCLK),.reset(RST),.clr(CLEAR));

FIFOMEM
FIFO_MEM(.rdata(RDATA),.wdata(WDATA),.waddr(waddr),.raddr(raddr),.wclk
en(WE),.wfull(full),.wclk(WCLK));

assign FULL=full;

endmodule
```

SertoPar.v

```
/*****************************************************************
***************
*** File: SertoPar.v                          Created By:Shashank
Mehta          ***
***
***
*****************************************************************
***************
*** This module converts single bit serial data into 10bit parallel
data          ***
***
***
```

```
***
***
*************************************************************************
***************/

`timescale 1ns/100ps

module SertoPar(Parout,clk,rst,Serialin);//module name
output [9:0]Parout;                      //Output port
input clk,rst;                           //Input port
input Serialin;                          //Input port
reg [9:0]Parout;



always@(posedge clk or negedge rst)
begin
     if(!rst)
     Parout<=0;
     else
     Parout<= {Serialin,Parout[9:1]};
end

endmodule
```

RX_STATUS.v

```
/*************************************************************************
****************
*** File: Rx_STATUS.v                              Created By:Shashank
Mehta          ***
***
***
*************************************************************************
****************
*** This module is used to generate appropriate RxStatus according to
received     ***
*** parallel data
***
***
***
*************************************************************************
***************/


`timescale 1ns/100ps

module
RX_STATUS(RxStatus,RxValid,clk,rst,BUFF_full,BUFF_empty,DataBuffer);//
Module name
```

```
output [2:0]RxStatus;
//Output port
output RxValid;
//Output port
input clk,rst,BUFF_full,BUFF_empty;
//Input Port
input [9:0]DataBuffer;
//Input Port
reg Valid,Disp_error,SKP_add,SKP_remove;
reg [2:0]RxStatus;
wire RxValid;

//to detect K28.5
always@(posedge clk or negedge rst)
begin
      if(!rst)
      Valid<=1'b0;
      else if(DataBuffer==10'b1010111100)
      Valid<=1'b1;
      else
      Valid<=1'b0;
end

assign RxValid=Valid;

//to check the disparity
always@(posedge clk or negedge rst)
begin
      if(!rst)
      Disp_error<=1'b0;
      else if(^DataBuffer==1'b1)
      Disp_error<=1'b0;
      else
      Disp_error<=1'b1;
end

// to generate valid RxStatus
always@(posedge clk or negedge rst)
begin
      if(!rst)
      RxStatus<=3'b000;
      else if(Valid)
      RxStatus<=3'b011;
      else if(Disp_error)
      RxStatus<=3'b100;
      else if(BUFF_full)
      RxStatus<=3'b101;
      else if(BUFF_empty)
      RxStatus<=3'b110;
      else
      RxStatus<=3'b000;
end
```

```
endmodule
```

DECODE.v

```
/*****************************************************************
****************
*** File: DECODE.v                              Created By:Shashank Mehta
***
***
***
*****************************************************************
****************
*** This module will conver 10bit dc balanced data into 8bit data
***
***
***
***
***
*****************************************************************
***************/

`timescale 1ns/100ps

module DECODE(DECODE_OUT,KO,DEC_CLK,DEC_RST,DATA_IN);//module name
output [7:0]DECODE_OUT;                                //Output port
output KO;                                             //Output port
input DEC_CLK,DEC_RST;                                 //Input port
input [9:0]DATA_IN;                                    //Input port

reg [7:0]DECODE_OUT;
reg KO;

wire aneb,cned,eei,p13,p22,p31;
wire ika,ikb,ikc;
wire xa,xb,xc,xd,xe,xf,xg,xh;
wire or121,or122,or123,or124,or125,or126,or127;
wire or131,or132,or133,or134;
reg ior134;
wire ai,bi,ci,di,ei,fi,gi,hi,ii,ji;

assign ai=DATA_IN[0];
assign bi=DATA_IN[1];
assign ci=DATA_IN[2];
assign di=DATA_IN[3];
assign ei=DATA_IN[4];
assign ii=DATA_IN[5];
assign fi=DATA_IN[6];
assign gi=DATA_IN[7];
```

```
assign hi=DATA_IN[8];
assign ji=DATA_IN[9];


//6b Input function
assign p13=(aneb & (!ci & !di)) | (cned & (!ai & !bi));
assign p31=(aneb & ci & di) | (cned & ai & bi);
assign p22=(ai & bi & (!ci & !di)) | (ci & di & (!ai & !bi)) | (aneb &
cned);
assign aneb= ai ^ bi;
assign cned= ci ^ di;
assign eei= ei ^! ii;

// K Decoder
assign ika=(ci & di & ei & ii) | (!ci & !di & !ei & !ii);
assign ikb=p13 & (!ei & ii & gi & hi & ji);
assign ikc=p31 & (ei & !ii & !gi & !hi & !ji);



//Determine K output
always@(posedge DEC_CLK or negedge DEC_RST)
begin
     if(!DEC_RST)
     begin
     KO<=1'b0;
     ior134=1'b0;
     end
     else
     begin
     KO<= ika | ikb | ikc;
     ior134<= (!(hi & ji)) & (!(!hi & !ji)) & (!ci & !di & !ei & !ii);
     end
end


//5b Decoder
//logic to determine compliementing A,B,C,D,E,I inputs

assign or121=(p22 & (!ai & !ci & eei)) | (p13 & !ei);
assign or123=(p31 & ii) | (p22 & bi & ci & eei) | (p13 & di & ei &
ii);
assign or122=(ai & bi & ei & ii) | (!ci & !di & !ei & !ii) | (p31 &
ii);
assign or124=(p22 & ai & ci & eei) | (p13 & !ei);
assign or125=(p13 & !ei) | (!ci & !di & !ei & !ii) | (!ai & !bi & !ei
& !ii);
assign or126=(p22 & !ai & !ci & eei) | (p13 & !ii);
assign or127=(p13 & di & ei & ii) | (p22 & !bi & !ci & eei);

assign xa= or127 | or121 | or122;
assign xb= or122 | or123 | or124;
```

```
assign xc= or121 | or123 | or125;
assign xd= or122 | or124 | or127;
assign xe= or125 | or126 | or127;



//Generate and latch LS 5 decoded bits
always@(posedge DEC_CLK or negedge DEC_RST)
begin
      if(!DEC_RST)
      begin
            DECODE_OUT[0]<=1'b0;
            DECODE_OUT[1]<=1'b0;
            DECODE_OUT[2]<=1'b0;
            DECODE_OUT[3]<=1'b0;
            DECODE_OUT[4]<=1'b0;
      end
      else
      begin
            DECODE_OUT[0]<=xa ^ ai;
            DECODE_OUT[1]<=xb ^ bi;
            DECODE_OUT[2]<=xc ^ ci;
            DECODE_OUT[3]<=xd ^ di;
            DECODE_OUT[4]<=xe ^ ei;
      end
end


//3b Decoder
assign or131= (gi & hi & ji) | (fi & hi & ji) | ior134;
assign or132= (fi & gi & ji) | (!fi & !gi & !hi) | (!fi & !gi & hi &
ji);
assign or133=(!fi & !hi & !ji) | (ior134) | (!gi & !hi & !ji);
assign or134= (!gi & !hi & !ji) | (fi & hi &ji) | (ior134);
//assign ior134= (!(hi & ji)) & (!(!hi & !ji)) & (!ci & !di & !ei &
!ii);

assign xf= or131 | or132;
assign xg= or132 | or133;
assign xh= or132 | or134;

//Generate and latch MS 3 decoded bits
always@(negedge DEC_CLK or negedge DEC_RST)
begin
      if(!DEC_RST)
      begin
      DECODE_OUT[5]<=1'b0;
      DECODE_OUT[6]<=1'b0;
      DECODE_OUT[7]<=1'b0;
      end
      else
      begin
```

```
      DECODE_OUT[5]<=xf ^ fi;
      DECODE_OUT[6]<=xg ^ gi;
      DECODE_OUT[7]<=xh ^ hi;
      end
end
endmodule
```

tb_PHY.v

```
`timescale 1ns/100ps

module tb_PHY();
wire [7:0]RX_DATA;
wire RX_DATAK;
wire [2:0]RX_STATUS;
wire RX_VALID;
wire PHY_PCLK,PHY_STATUS;
reg [7:0]TX_DATA;
reg TX_DATAK;
reg [1:0]PHY_POWERDOWN;
reg [1:0]PHY_MODE;
reg PHY_CLK,PHY_RST,TX_ELECIDLE,PHY_CLR;
reg PHY_RATE,RX_POLARITY;
reg RX_RINC,RX_WINC;


PHY
uut(RX_DATA,RX_DATAK,RX_STATUS,RX_VALID,PHY_PCLK,PHY_STATUS,PHY_CLK,PH
Y_POWERDOWN,TX_ELECIDLE,PHY_MODE,PHY_RATE,PHY_RST,TX_DATA,TX_DATAK,RX_
RINC,RX_WINC,PHY_CLR,RX_POLARITY);

initial begin
$monitorb ("%d TX_DATA=%h TX_DATAK=%h RX_STATUS=%b RX_VALID=%b
RX_DATA=%h
RX_DATAK=%h",$time,TX_DATA,TX_DATAK,RX_STATUS,RX_VALID,RX_DATA,RX_DATA
K);
end
initial begin
      PHY_CLK<=1'b0;
      forever #0.1 PHY_CLK<=~PHY_CLK;
end



initial begin
TX_DATA<=8'h00; TX_DATAK<=1'b0; TX_DATAK<=1'b0; RX_WINC<=1'b0;
RX_RINC<=1'b0; PHY_CLR<=1'b0; RX_POLARITY<=1'b0;
//RESET TX
PHY_RST<=1'b0;
#4 PHY_RST<=1'b1;
```

```
//Changing PCLK frequency
#4 PHY_POWERDOWN<=2'b00; PHY_MODE<=2'b00; TX_ELECIDLE<=1'b1;
PHY_RATE<=1'b0;
#20   PHY_RATE<=1'b1;
#20   PHY_POWERDOWN<=2'b11;
#20   PHY_POWERDOWN<=2'b00; TX_ELECIDLE<=1'b0;
// Encode data
#4 TX_DATA<=8'hBC; TX_DATAK<=1'b1; RX_WINC<=1'b1; RX_RINC<=1'b0;
#10 TX_DATA<=8'hFF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h17; TX_DATAK<=1'b0;
#18 TX_DATA<=8'hC0; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h14; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hB2; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hE7; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h02; TX_DATAK<=1'b0;
#18 TX_DATA<=8'h82; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h72; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6E; TX_DATAK<=1'b0;

#8 TX_DATA<=8'h28; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hA6; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBE; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6D; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0; RX_POLARITY<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0; RX_POLARITY<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;

#4 TX_DATA<=8'hBC; TX_DATAK<=1'b1;
#10 TX_DATA<=8'hFF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h17; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hC0; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h14; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hB2; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hE7; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h02; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h82; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h72; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6E; TX_DATAK<=1'b0;
```

```
#8 TX_DATA<=8'h28; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hA6; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBE; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6D; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;


#4 TX_DATA<=8'hBC; TX_DATAK<=1'b1;
#10 TX_DATA<=8'hFF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h17; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hC0; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h14; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hB2; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hE7; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h02; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h82; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h72; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6E; TX_DATAK<=1'b0;

#8 TX_DATA<=8'h28; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hA6; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBE; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6D; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
```

```
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;

#4 TX_DATA<=8'hBC; TX_DATAK<=1'b1;
#10 TX_DATA<=8'hFF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h17; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hC0; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h14; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hB2; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hE7; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h02; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h82; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h72; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6E; TX_DATAK<=1'b0;

#8 TX_DATA<=8'h28; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hA6; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBE; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6D; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;

#4 TX_DATA<=8'hBC; TX_DATAK<=1'b1;
#10 TX_DATA<=8'hFF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h17; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hC0; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h14; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hB2; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hE7; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h02; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h82; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h72; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6E; TX_DATAK<=1'b0;

#8 TX_DATA<=8'h28; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hA6; TX_DATAK<=1'b0;
#8 TX_DATA<=8'hBE; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h6D; TX_DATAK<=1'b0;
```

```
#8 TX_DATA<=8'hBF; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h4A; TX_DATAK<=1'b0;

// giving random data
#8 TX_DATA<=8'h02; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h03; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h04; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h05; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h06; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h07; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h08; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h09; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h10; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h11; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h12; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h04; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h05; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h06; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h07; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h08; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h09; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h10; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h11; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h12; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h04; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h05; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h06; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h07; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h08; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h09; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h10; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h11; TX_DATAK<=1'b0;
#8 TX_DATA<=8'h12; TX_DATAK<=1'b0;

#1200 RX_RINC<=1'b1; RX_WINC<=1'b1;
#1200 RX_POLARITY<=1'b1;
#1200 RX_POLARITY<=1'b0;
#12000 $stop;
```

```
# 20 $finish;
end
endmodule
```

LTSSM.v

```
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
////////////////////////////////////
*** File Name:LTSSM.v                              Created By:Shashank
Mehta
***
*** Date:04/03/12
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
////////////////////////////////////
*** LTSSM is the Link Transition and Status State Machine which
Transition the link into appropriate power stage. This is used by
SuperSpeed link
*** to transker the link into low power stage when it is not in used.
***
/////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////
////////////////////////////////////*/

`timescale 1ns/100ps

module LTSSM(Link_State,SuperSpeed,LFPS,
Bit_Lock,Symbol_Lock,Rx_Equalization,CLK,RST,Loop_Back,Idle,Low_Impeda
nce,Rx_Termination,LGO_U1,LGO_U2,LGO_U3,LFPS_Handshake,Time_Out,VBUS,W
arm_RST,Error);
parameter SS_Disable=4'b0000;
parameter SS_Inactive=4'b0001;
parameter Rx_Detect=4'b0010;
parameter Polling=4'b0011;
parameter U0=4'b0100;
parameter U1=4'b0101;
parameter U2=4'b0110;
parameter U3=4'b0111;
parameter Hot_Reset=4'b1000;
parameter Recovery=4'b1001;
parameter LoopBack=4'b1010;
parameter Compliance_mode=4'b1011;
output [3:0]Link_State;
output SuperSpeed,LFPS,Bit_Lock,Symbol_Lock,Rx_Equalization;
input
CLK,RST,Loop_Back,Idle,Low_Impedance,LGO_U1,LGO_U2,LGO_U3,LFPS_Handsha
ke,Time_Out,Rx_Termination,VBUS,Warm_RST,Error;
reg SuperSpeed,LFPS,Bit_Lock,Symbol_Lock,Rx_Equalization;
reg [3:0]state,next_state;
```

```
wire [3:0]Link_State;

always@(posedge CLK or negedge RST)
begin
      if(!RST)
      begin
      state<=Hot_Reset;
      SuperSpeed<=1'b0;
      LFPS<=1'b0;
      Bit_Lock<=1'b0;
      Symbol_Lock<=1'b0;
      Rx_Equalization<=1'b0;
      end
      else
      state<=next_state;
end


always@(VBUS or RST or Rx_Termination or Idle or Loop_Back or Recovery
or Polling or Idle or Low_Impedance or LGO_U1 or LGO_U2 or LGO_U3 or
LFPS_Handshake or Time_Out or Warm_RST or Error)
begin
      next_state=state;
      case(state)
      SS_Disable: begin
            LFPS=0;
            SuperSpeed=0;
            if(VBUS)
            next_state=Rx_Detect;
            end

      SS_Inactive: begin
            SuperSpeed=0;
            if(Warm_RST)
            next_state=Rx_Detect;
            end

      Rx_Detect:begin
            SuperSpeed=1'b1;
            LFPS=1'b0;
            if(!Warm_RST || Rx_Termination)
            next_state= Polling;
            end

      Polling:begin
            SuperSpeed=1'b1;
            LFPS=1'b0;
            Bit_Lock=1'b1;
            Symbol_Lock=1'b1;
            Rx_Equalization=1'b1;
            if(Warm_RST)
            next_state=Rx_Detect;
```

```
      else if(!VBUS)
      next_state=SS_Disable;
      else if(Idle)
      next_state=U0;
      else if(Loop_Back)
      next_state=LoopBack;
      else if(Low_Impedance)
      next_state=Compliance_mode;
      end

U0: begin
      SuperSpeed=1'b1;
      LFPS=1'b0;
      if(LGO_U1)
      next_state=U1;
      else if(LGO_U2)
      next_state=U2;
      else if(LGO_U3)
      next_state=U3;
      else if(Error)
      next_state=Recovery;
      else
      next_state=SS_Inactive;
    end

U1: begin
      SuperSpeed=1'b1;
      LFPS=1'b1;
      if(LFPS_Handshake)
      next_state=Recovery;
      else if(Time_Out)
      next_state=U2;
      else if(!LFPS_Handshake)
      next_state=SS_Inactive;
      else if(Warm_RST)
      next_state=Rx_Detect;
      else if(!VBUS)
      next_state=Rx_Detect;
     end

U2: begin
      SuperSpeed=1'b1;
      LFPS=1'b1;
      if(LFPS_Handshake)
      next_state=Recovery;
      else if(Time_Out)
      next_state=U3;
      else if(!LFPS_Handshake)
      next_state=SS_Inactive;
      else if(Warm_RST || !VBUS)
      next_state=Rx_Detect;
      else
```

```
      next_state=U2;
     end

U3: begin
      SuperSpeed=1'b0;
      LFPS=1'b0;
      if(LFPS_Handshake)
      next_state=Recovery;
      else if(Time_Out)
      next_state=U3;
      else if(!LFPS_Handshake)
      next_state=U3;
      else if(Warm_RST)
      next_state=Rx_Detect;
      else if(!VBUS)
      next_state=SS_Disable;
     end

Recovery: begin
      SuperSpeed=1'b0;
      LFPS=1'b0;
      if(!VBUS)
      next_state=SS_Disable;
      else if(Loop_Back)
      next_state=LoopBack;
      else if(Idle)
      next_state=U0;
      else if(!LFPS_Handshake)
      next_state=SS_Inactive;
      else if(Warm_RST)
      next_state=Rx_Detect;
      end

LoopBack: begin
      SuperSpeed=1'b1;
      LFPS=1'b1;
      if(!LFPS_Handshake)
      next_state=SS_Inactive;
      else if(Warm_RST)
      next_state=Rx_Detect;
      end

Hot_Reset: begin
      SuperSpeed=1'b0;
      LFPS=1'b0;
      if(!LFPS_Handshake)
      next_state=SS_Inactive;
      else if(Idle)
      next_state=U0;
      end
```

```
      Compliance_mode:begin
            SuperSpeed=1'b1;
            LFPS=1'b1;
            if(Warm_RST)
            next_state=Rx_Detect;
            end
      endcase
end
assign Link_State=state;
endmodule
```

tb_LTSSM.v

```
`timescale 1ns/100ps

module tb_LTSSM();
parameter SS_Disable=4'b0000;
parameter SS_Inactive=4'b0001;
parameter Rx_Detect=4'b0010;
parameter Polling=4'b0011;
parameter U0=4'b0100;
parameter U1=4'b0101;
parameter U2=4'b0110;
parameter U3=4'b0111;
parameter Hot_Reset=4'b1000;
parameter Recovery=4'b1001;
parameter LoopBack=4'b1010;
parameter Compliance_mode=4'b1011;
wire [3:0]Link_State;
wire SuperSpeed,LFPS,Bit_Lock,Symbol_Lock,Rx_Equalization;
reg
CLK,RST,Loop_Back,Idle,Low_Impedance,LGO_U1,LGO_U2,LGO_U3,LFPS_Handsha
ke,Time_Out,Rx_Termination,VBUS,Warm_RST,Error;

LTSSM  uut(Link_State,SuperSpeed,LFPS,
Bit_Lock,Symbol_Lock,Rx_Equalization,CLK,RST,Loop_Back,Idle,Low_Impeda
nce,Rx_Termination,LGO_U1,LGO_U2,LGO_U3,LFPS_Handshake,Time_Out,VBUS,W
arm_RST,Error);

initial begin
$monitorb("%d Link_State=%d SuperSpedd=%b LFPS=%b Bit_Lock=%b
Symbol_Lock=%b Rx_Equalization=%b CLK=%b RST=%b Loop_Back=%b Idle=%b
Low_Impedance=%b Rx_Termination=%b LGO_U1=%b LGO_U2=%b LGO_U3=%b
LFPS_Handshake=%b Time_Out=%b VBUS=%b Warm_RST=%b
Error=%b",$time,Link_State,SuperSpeed,LFPS,
Bit_Lock,Symbol_Lock,Rx_Equalization,CLK,RST,Loop_Back,Idle,Low_Impeda
nce,Rx_Termination,LGO_U1,LGO_U2,LGO_U3,LFPS_Handshake,Time_Out,VBUS,W
arm_RST,Error);
end
```

```
initial begin
CLK<=1'b0;
forever #10 CLK<=~CLK;
end

initial begin
#5 RST<=1'b0;
#5 RST<=1'b1;//Hard Reset
#10 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b0; Error<=1'b0;

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1; Error<=1'b0;//
Rx_Detect

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0;
Error<=1'b0;//Polling

#100 Loop_Back<=1'b0; Idle<=1'b1; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//U0

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1;
Error<=1'b0;//SS_Inactive

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b1; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1;
Error<=1'b0;//Rx_Detect

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b1; Time_Out<=1'b0;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0;
Error<=1'b0;//Polling

#100 Loop_Back<=1'b1; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b1;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0;
Error<=1'b0;//Loop_Back
```

```
#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b1; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1;
Error<=1'b0;//SS_Inactive

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b1; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1;
Error<=1'b0;//Rx_Detect i.e 2

#100 Loop_Back<=1'b0; Idle<=1'b1; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0;
Error<=1'b0;//Polling i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b1; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0;
Error<=1'b0;//Compliance i.e B

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1;
Error<=1'b0;//Rx_Detect i.e 2

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0;
Error<=1'b0;//Polling i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b1; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//U0 i.e
4

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b1;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//U1 i.e
5

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//U2 i.e
6

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b1; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;// U3
i.e 6

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
```

```
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;// U3
i.e 7

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b1; Time_Out<=1'b1;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//
Recovery i.e 9

#100 Loop_Back<=1'b1; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b0; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//
Loop_Back i.e 10

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b0; VBUS<=1'b0; Warm_RST<=1'b1; Error<=1'b0;//
Rx_Detect i.e 2

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0; Error<=1'b0;//
Polling i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b1; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0; Error<=1'b0;//
Compliance i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b1; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b1; Error<=1'b0;//
Compliance i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b1; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b1;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0; Error<=1'b0;//
Compliance i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b1; VBUS<=1'b0; Warm_RST<=1'b0; Error<=1'b0;//
Compliance i.e 3

#100 Loop_Back<=1'b0; Idle<=1'b0; Low_Impedance<=1'b0; LGO_U1<=1'b0;
LGO_U2<=1'b0; LGO_U3<=1'b0; LFPS_Handshake<=1'b0; Time_Out<=1'b0;
Rx_Termination<=1'b1; VBUS<=1'b1; Warm_RST<=1'b0; Error<=1'b0;//
Compliance i.e 3
#2000 $stop;
#200 $finish;
end

endmodule
```