ASIC Design Laboratory
Cooperative Design Lab (CDL)
USB Full-Speed Bulk-Transfer Endpoint AHB-Lite SoC Module
Design Manual (3-Person Teams)

Spring 2023

The purpose of this multi-week cooperative lab is to help you gain experience working with a team of designers to implement and validate a larger scale design. The previous labs have all be focused on smaller self-contained designs to allow you to develop your system and hardware design skills, along with developing your ability to use an HDL to describe hardware systems. In this lab you will be directly responsible for designing and implementing a fairly sized portion of an overall design, with your team members implementing the other portions, and then working working you teammates to integrate your work into the overall design. Additionally, you will need to design the validation setup for the work one of your teammates implemented, and one of them will do so for the portion you designed. This setup is more similar to what you would experience in industry, where someone other than you will be responsible for validating your work according the standards and requirements that were set for your design, rather than based on knowing how it is implemented.

This design is going to be larger in design effort than your prior lab work and will have more opportunity for error propagation. Therefore it is paramount that you follow efficient debugging approaches based on clearly establishing cause-effect relationships through your design working backward from the chronologically first high-level problematic behavior. Other lazy or speculative debugging methods will most certainly result in vast amounts of wasted time, effort, and frustration and can easily increase debugging times by easily a factor of 10x.

In this lab, you (with your team) will perform the following tasks:

- Plan out the implementation of a larger design composed of multiple functional modules, that each are composed of multiple sub-modules, based on a provided starter architecture.
- Submit your implementation planning as a group via Brightspace submission
- Demonstrate and discuss your design planning with your TA.
- Implement the major modules of the design architecture as a hierarchy of smaller function-focused modules.
- Develop test benches to validate the major modules (the test bench you develop can not be for the major module you are responsible for).
- Directly validate the major modules of the design prior to integrating them to form the overall design.
- Integrate the major modules to form the overall required design.
- Synthesize the overall design (or at least its major modules) using Design Compiler®
- Validate the Synthesized/Mapped version of the overall design (or at least its major modules)
- Prove via demonstration to course staff that your design (or at least the major modules) works as required.
- Perform area and timing analysis of the major modules and overall design.
- Document any needed revisions to your design's architectural approach, your validation methodology, and the area and timing analysis of your design via a final report.
- Submit your final report as a group via Brightspace submission.

1 Lab Work Overview

1.1 Design Overview

For the Cooperative Design Lab you will be designing, implementing, and verifying a System-on-Chip peripheral module that would add USB Full-Speed Bulk-Transfer End-point support to an AHB-Lite based SoC. USB in general is a standard for governing a shared bus for connecting multiple peripheral devices (USB Endpoints) such as input devices (keyboards, mice, joysticks, etc.), external storage devices, and printers to a common primary device (USB host) such as computer. Each of these peripheral devices will generally be an SoC of it's own, with it's own (usually light-weight) CPU/microcontroller, on-board volatile memory, potentially some non-volatile/permanent memory, and other peripheral modules for managing various other IO connections used within the device. Some devices might also have power-optimized compute accelerators if high-precision or sophisticated controls are needed, such as in printers or high resolution scanning equipment. An example of one such SoC system is shown in Figure 1 for a contextual reference. The AHB-Lite and USB Full-Speed Bulk-Transfer related specifications are discussed in dedicated sections, followed by sections discussing the required Cooperative Design Lab architecture, required core components, and required team workload allocations.

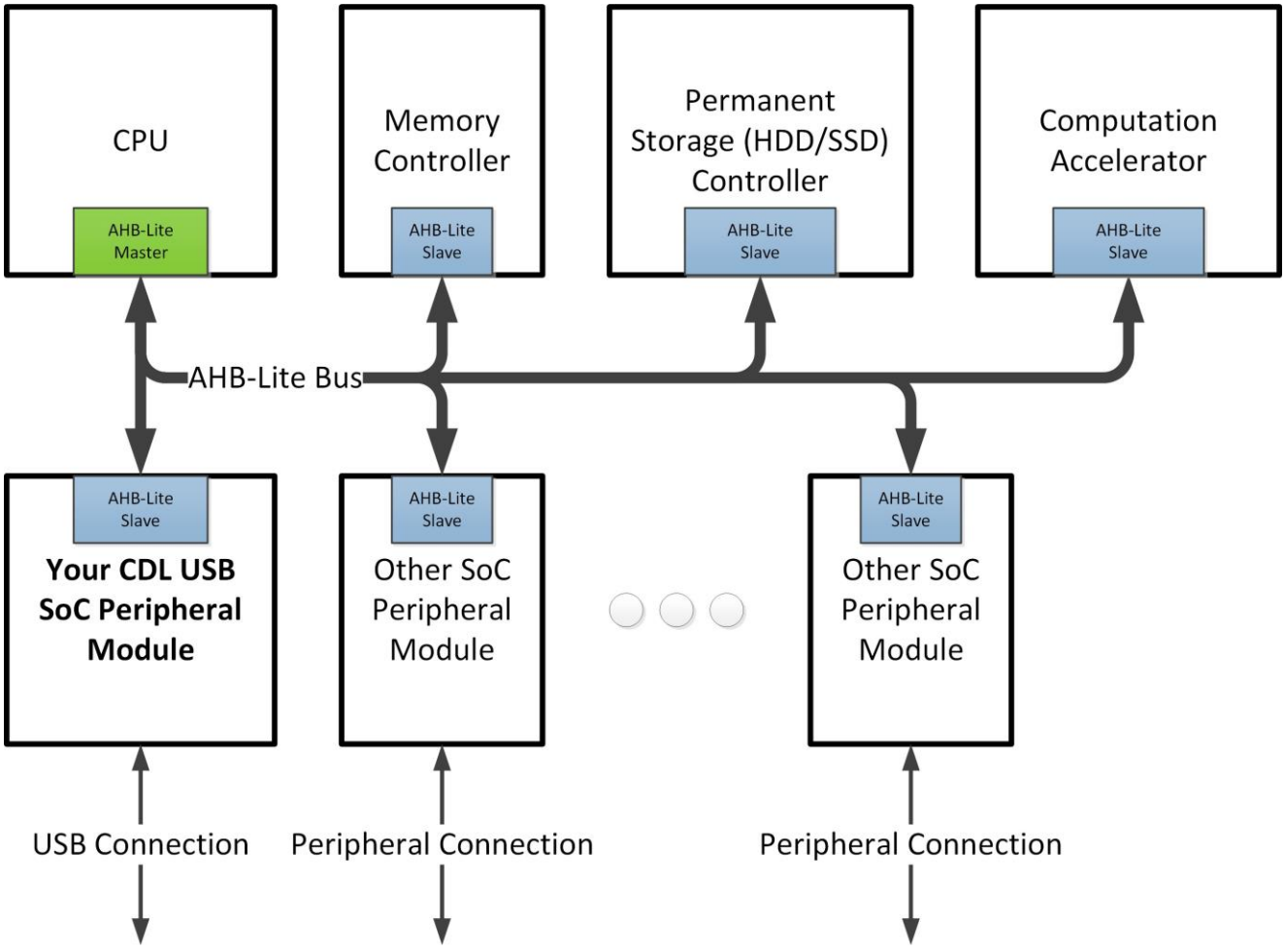

Figure 1: Example of CDL USB Full-Speed Bulk-Transfer End-point peripheral usage in an end-point device SoC

1.2 Expectations Regarding the Design Planning Stage

During the design planning stage you are required to:

- Create separate functional block or hierarchical RTL diagrams to describe the internal structure of each of the major modules shown in the design's overall architecture
- Create separate RTL diagrams for each block indicated in a major module's diagram.
- For each block provide a description of the function of that block in one of the following forms: pseudocode, state transition diagram, or flow-chart.
- Demonstrate and discuss your diagrams with your TA.
- Submit your prepared diagrams as a group via a Brightspace assignment.

You must have these diagrams submitted as either PDF file(s) or image files using common standards (JPEG, PNG, or BMP) in order to earn points for them.

2 Advanced High-performance Bus Lite Protocol

You should already be somewhat familiar with AHB-Lite from Lab 9. However, this lab will require supporting more of the full protocol and thus this section will both recover the portions of the protocol covered in the Lab 9 manual and also discuss the additional portions of the protocol that your design will be supporting during this Cooperative Design Lab.

The Advanced High-performance Bus Lite (AHB-Lite) protocol[1] is one of many that form the Advanced Micro-processor Bus Architecture (AMBA) Bus System design by ARM.

Like most SoC buses, AHB-Lite has three main aspects or parts:

1. The 'master' interface device which is in charge of initiating any/all requests on the bus
2. At least one 'slave' interface device which responds to requests that are directed to it through the bus
3. The bus 'fabric' which handles how all of these devices are physically connected and how control and data signals for requests are routed between the two devices involved in an bus transaction.

2.1 AHB-Lite Signals

The following are the various signals (by name) that are used within the AHB-Lite protocol and their respective roles:

HCLK This is the bus (and commonly system) clock signal.

HRESETn This is the bus (and usually system) active-low reset signal.

HADDR This is used for providing the address (within the slave only) that the transaction involves.

HPROT This signal is used for selecting between protection levels for the transfers. [Not required for this project]

HMASTERLOCK This signal indicates that the transfer is 'locked' by the master and thus must be treated as an atomic operation. [Not required for this project]

HSELx This is 'slave' selection signal where the 'x' is replaced by a number for the slave it is connected to and each 'slave' device is given its own dedicated one.

HTRANS This indicates the type of the current transfer (value encoding provided in Table 1).

HSIZE This indicates the size of the transfer. Typically is Byte, half-word, or full-word size and scales with the data bus size, with the number of bytes equal to 2^{HSIZE}.

HWRITE This indicates whether a transaction is a read (logic low value) or write (logic high value).

HWDATA This is used for transferring the data written to the 'slave' device.

HRDATA This is used for transferring the data read from the 'slave' device and can be up to 32-bits wide.

HREADY This is an active-high ready feedback signal from the 'slave' device which is pulled low by the 'slave' if it needs to stall/pause the transaction.

HRESP This is an active-high error feedback signal from the 'slave' device, and must be asserted when there is a transaction error (such as trying to write to read-only addresses).

HBURST This indicates the nature of the current burst transfer (value encoding provided in Table 2). [Not required for this project, but can be done for bonus]

Table 1: The value encoding scheme for the 'HTRANS' signal

Value	Label	Description
0	IDLE	The bus in an idle phase.
1	BUSY	There is currently a master triggered delay cycle within a variable length burst.
2	NONSEQ	This is either an individual transfer or a the first beat of a burst.
3	SEQ	This is one of the beats in a burst.

Table 2: The value encoding scheme for the 'HBURST' signal [Not required. Bonus only.]

Value	Label	Description
0	SINGLE	This is a single transfer.
1	INCR	This is an incrementing burst of variable length.
2	WRAP4	This is a 4-beat wrap-around burst.
3	INCR4	This is a 4-beat incrementing burst.
4	WRAP8	This is a 8-beat wrap-around burst.
5	INCR8	This is a 8-beat incrementing burst.
6	WRAP16	This is a 16-beat wrap-around burst.
7	INCR16	This is a 16-beat incrementing burst.

In this lab you are going to be focusing on implementing a 'slave' device for the AHB-Lite protocol and so the following sections are going to focus on bus transfers as seen by the 'slave' devices after the bus 'fabric' has already handled the routing of signals and data to or from the 'master' and 'slave' devices.

2.2 Operation of a Basic Write transfer

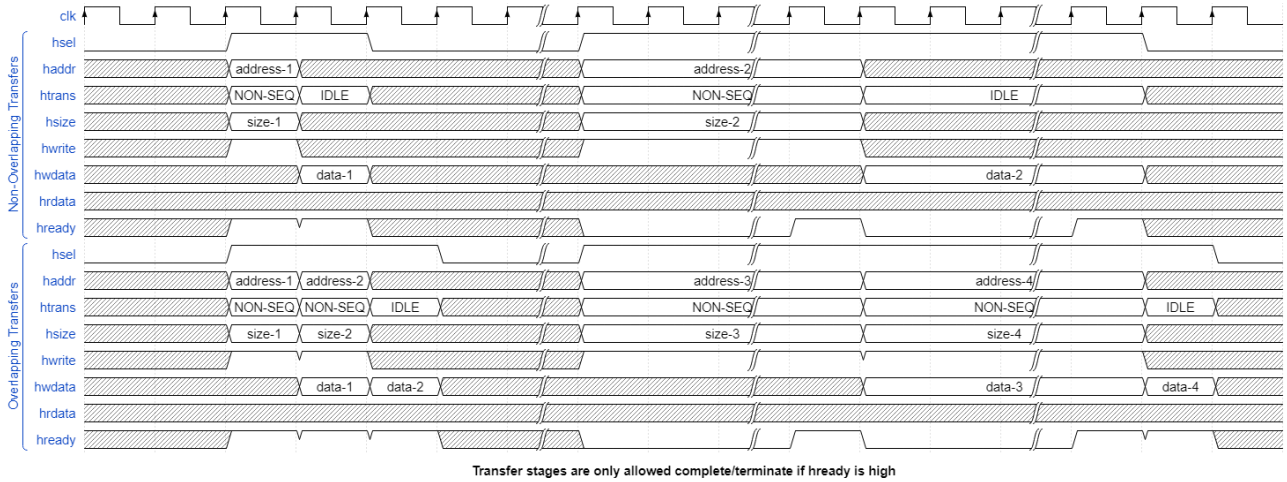

Figure 2: Example operation and timing of AHB-Lite Writes from the perspective of a 'slave'

Write transfers are only allowed to finish and release the bus once the 'slave' maintains the 'hready' signal (via its 'hreadyout') at a logic-high value during the second stage, to indicate that the write was either completed or at least internally buffered depending on the design. If the 'hready' signal is at a logic low value then the bus (and involved master) must stall for as long as 'hready' stays at a logic low value. This requirement is true for both the address phase and the data phase of transfers and can result in indefinite stall or 'freezing' of the bus if 'hready' is misused. Additionally, subsequent transfers can be overlapped or spaced out by the master.

2.3 Operation of a Basic Read transfer

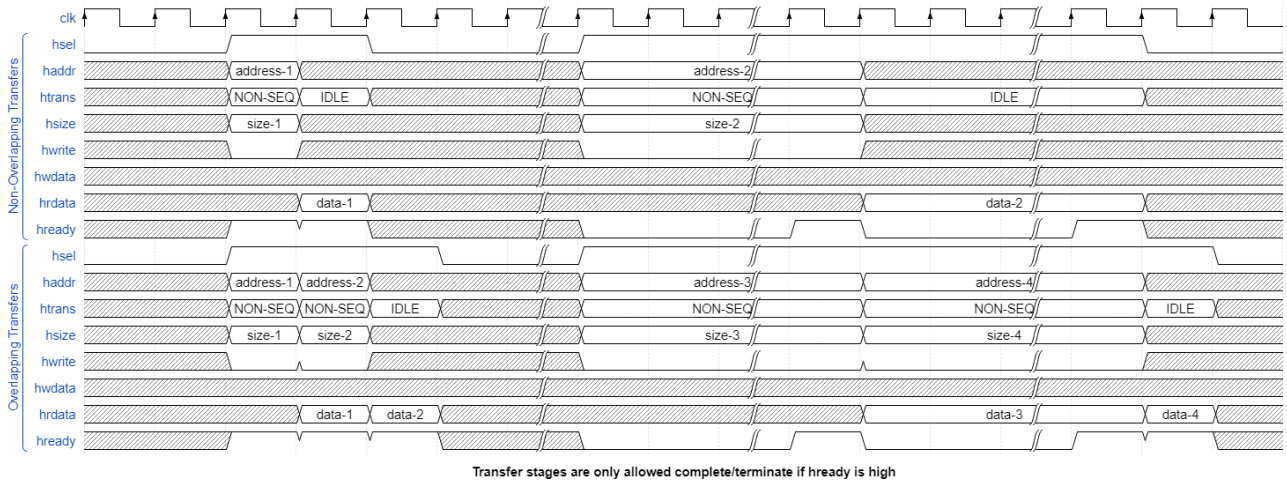

Figure 3: Example operation and timing of AHB-Lite Reads from the perspective of a 'slave'

Similar to write transfers, read transfers are only allowed to finish and release the bus once the 'slave' maintains the 'hready' signal (via its 'hreadyout' at a logic-high value during the second stage, to indicate that the data requested has been available on the 'hrdata' bus. If the 'hready' signal is at a logic low value then the bus (and involved master) must stall for as long as 'hready' stays at a logic low value. This requirement is true for both the address phase and the data phase of transfers and can result in indefinite stall or 'freezing' of the bus if 'hready' is misused. Additionally, subsequent transfers can be overlapped or spaced out by the master.

2.4 Transfer Error during Basic Read or Write

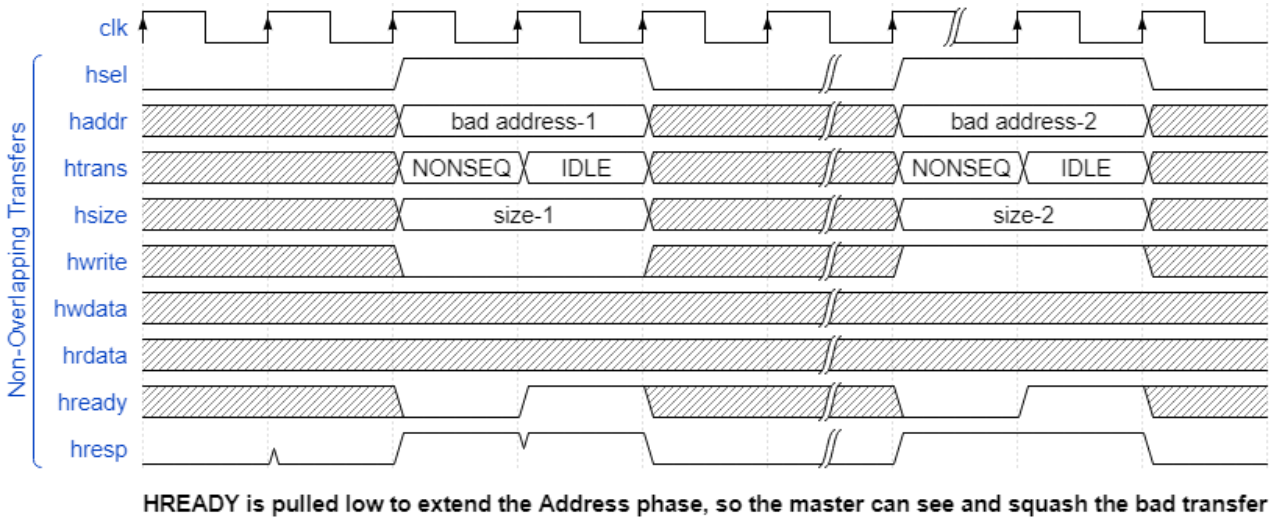

Figure 4: Example of erroneous read and write transfers.

In order for the master to be able to see the 'hresp' value (during an error) in time, the slave should pull 'hready' low to extend the address phase to be at least two cycles long. Once the master sees the error response it will generally choose to nullify the transfer by overriding it with an IDLE transfer value before the bus pipeline advances.

2.5 Burst Transfer Styles

Not applicable to teams three or smaller except for bonus credit.

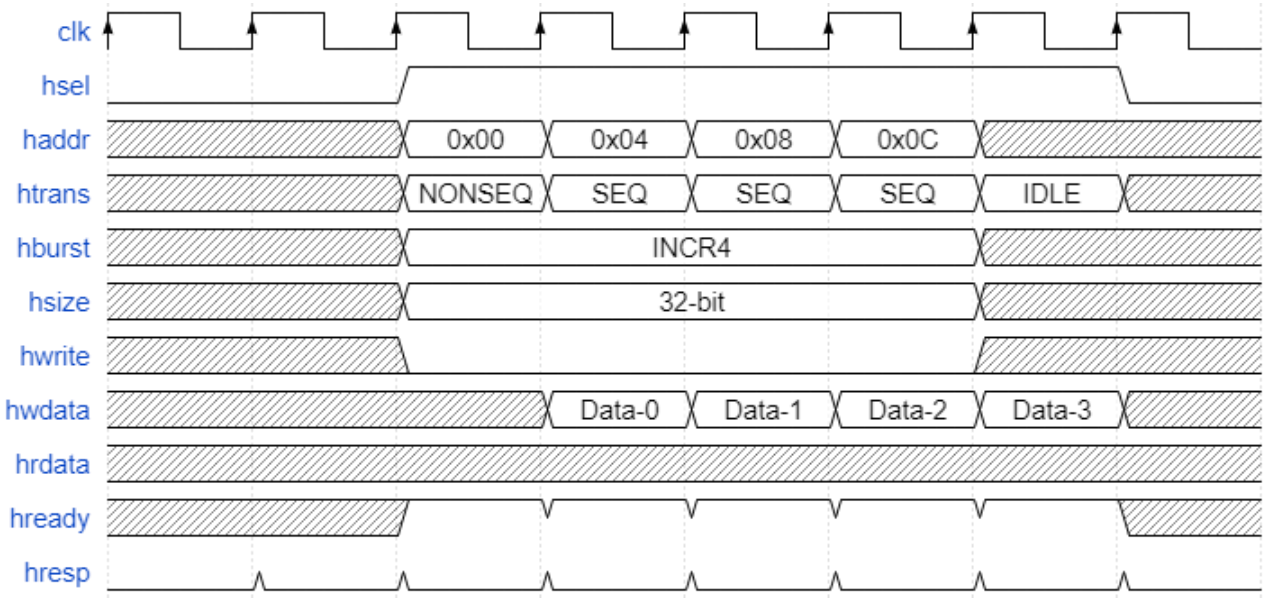

Figure 5: Example of a 4-beat incrementing write burst transfer

NOTE: There is no control or timing difference between read and write bursts, only which data bus is active.

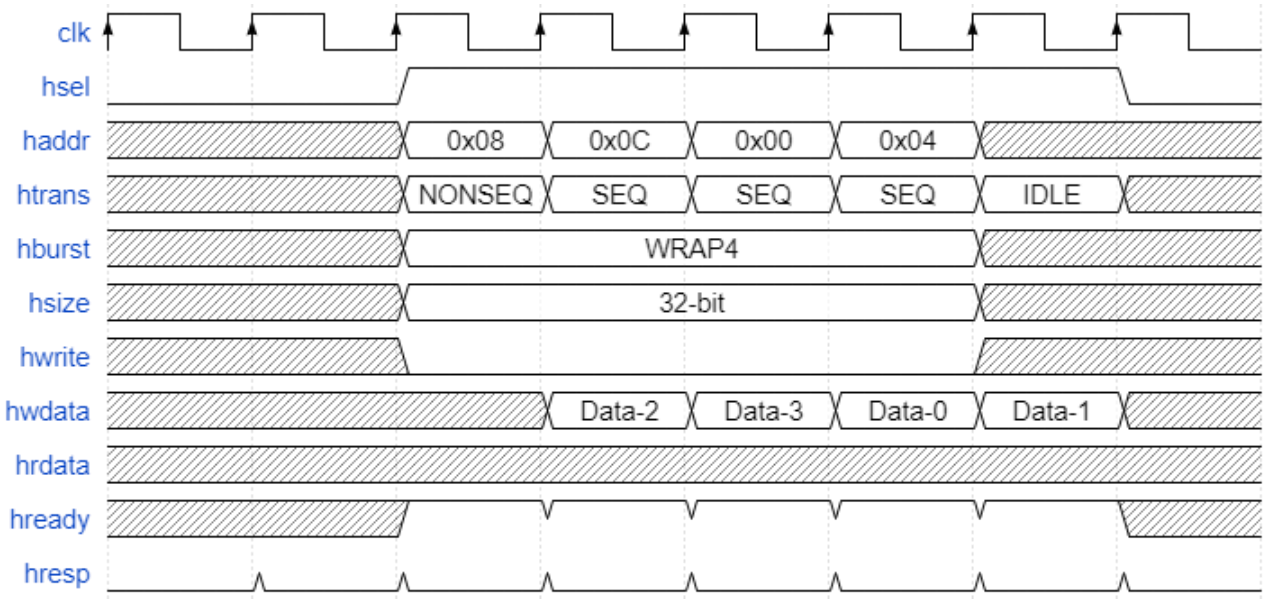

Figure 6: Example of a 4-beat wrap-around write burst transfer

3 Universal Serial Bus Protocol 1.1 (Full-Speed)

You are likely already familiar with the USB protocol[2] by name and general purpose (to connect nearly every possible peripheral to a computer). USB is an asynchronous serial communication protocol with a fairly sized but flexible base standard of available transfer modes. There is also a whole suite of protocols that have been built on top the the base protocol to standardize everything from USB connected mass-storage devices to various human interfacing devices, like keyboards and mice. For this design we are going focus only on the core basics of USB packets and one of the core styles of transfers called Bulk Transfers in order to keep things tractable.

There are also a wide range of free online resources that describe and explain the various aspects of this ubiquitous communication standard. One that is well structured and recommended for you checkout if you want to learn more about USB but not directly dive into the full official standard yet is https://www.beyondlogic.org/usbnutshell/usb4.shtml.

Before going forward it is important to establish a little design context and a couple key terms that will be heavily used later. USB as a protocol is designed entirely around the usage model of a single host (similar to a master in an SoC context) that is nearly always a computer/processor and a plethora of peripheral devices referred to as 'endpoints'. Your design is going to be one such endpoint. However, one thing you should notice later on is that a given peripheral (designated by it's address) can have multiple 'endpoints', this is allow some flexibility for how peripherals are communicated with and to enable the existence of USB hub designs. Your design will only have one internal 'endpoint' which will be the default endpoint number of '0'. Also, since we are not requiring you implement the start-up sequence where addresses are assigned by the 'host', your design should either simply use the 'default address' of '0' or a value that you preload into your design as if it had been assigned during a USB start-up sequence.

3.1 Expectations for project

Your design is only required to implement the ability to send and receive the types of USB packets described in this manual. It is not up to your design to keep track of the protocol sequences required for USB. That could be handled by software on a CPU that would be sending commands to this module to check on packets that were received and specify packets to be transmitted.

3.2 USB Line Encoding and Data Synchronization

USB encodes data bits within packets using a Non-Return-to-Zero Inverted (NRZI) encoding scheme to encode data as changes or non-changes in the wire voltage. More specifically the NRZI convention used by USB encodes '0' as toggling the current wire value and '1' as maintaining it as show in Figure 7. This allows for easier data reception/sampling synchronization by being able to have a clear point when a data bit starts or ends. In order to support clean sending and reception of large packets like those in USB, any data sampling timing needs to be re-synchronized to each edge in the D+/D- lines of a USB receiver design.

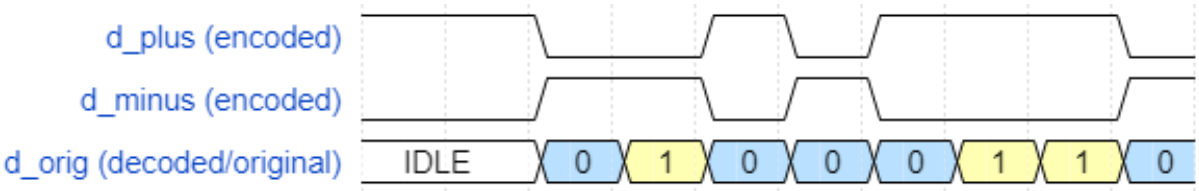

Figure 7: An example of USB-style NRZI encoding/decoding of data

In order to bound the maximum number of bits between edges in the data lines, USB also employs 'bit-stuffing' where a '0' is inserted between valid data bits after six continuous '1' bits have been sent the the data line encoder, in order to force a data line edge at that point. Conversely, on the receiving end these stuffed bits must be filtered so that they don't disturb the data values sent in the packets. Therefore, after every six bit periods of no data transitions

on the data lines, the decoders must assume the next bit period is going to be a stuffed ’0’ and thus prevent the receiver from considering that ’0’ as a valid data bit.

Furthermore, USB packets are transmitted least-significant-bit first.

3.3 USB Packet Types used in Bulk Transfers

All packets have a basic structure of:

1. A ’sync’ byte, which after line encoding is string of transitions that enable easier initial data synchronization at the receiver (illustrated in Figure 8).
2. A ’pid’ byte that encodes the type of packet it is. The first 4 bits (listed in Table 3) are sent in right-to-left order followed by their complemented value in right-to-left order.
3. An optional data field
4. An ’EOP’ or end-of-packet sequence where both D+ and D- are both driven low for 2 bit periods and the brought back to the idle bus value for at least one more bit period.

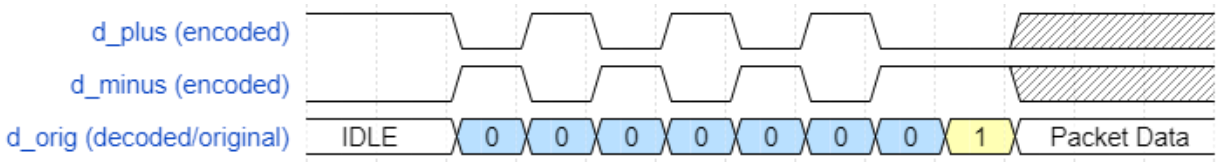

Figure 8: An illustration of the line encoded form of a USB packet’s ’sync’ field

Table 3: Listing of relevant USB packet id values and meanings

Value	Label	Description
4’b0001	OUT	’OUT’ Token packet id.
4’b1001	IN	’IN’ Token packet id.
4’b0011	DATA0	One of two available data packet ids.
4’b1011	DATA1	The other available data packet id.
4’b0010	ACK	An Acknowledgement Handshake. (Success)
4’b1010	NAK	A Negative Acknowledgement Handshake. (Usually means try again later)
4’b1110	STALL	Device is halted due to an error.

Out of the packet types you’ll be needing to handle in order support Bulk Data Transfers, only the Token and Data packets will use the optional data field. In both of these packet types, a Cyclic Redundancy Check (CRC) ’signature’ is included at the end of their respective data field layouts in order to allow the checking of the packet’s data integrity at it’s destination.

The token packets will have a data field consisting of:

1. A 7-bit address for the intended device
2. a 4-bit endpoint number to reference which sub-part or endpoint of the device it targeting
3. a 5-bit CRC (polynomial: $x^5 + x^2 + 1$, residual: ’01100’) of the first two sections of the data field. [for this project you may insert an arbitrary fixed sequence of bits]

The data packets will have a data field consisting of:

1. Up to 64 bytes of data

2. A 16-bit CRC (polynomial: $x^{16} + x^{15} + x^2 + 1$, residual: '1000000000001101') of the previous data bytes. [for this project you may insert an arbitrary fixed sequence of bits]

3.4 USB CRC [BONUS]

The wikipedia page for the calculation of CRCs (https://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks) has very useful animations that illustrate how simple modified shift registers can be used to calculate them for serial streams of data. USB also uese both the 'Preset to -1' and the 'Post-invert' options for CRC calculations. Furthermore, https://crccalc.com/ is a useful site for checking the CRC calculations for various standards/protocols and will included the 'Preset to -1' and 'Post-invert'. However, be careful to pay attention to the endianness expected for the input stream in these checkers and whether they are factoring in the two calculation options required by USB. The following site (http://www.ghsi.de/pages/subpages/Online%20CRC%20Calculation/) is good place to get started with verifying your core CRC calculation, but it does not account for the 'preset to -1' and the 'post-invert' options.

CRC checker and generators are equivalent hardware designs with the only difference being how they are used overall. A generator is initialized and fed the data stream for which to compute the checksum. If the 'Post-invert' option is used, the generator's result is finalized by inverting the value, instead of shifting zeros through. A checker is initialized, fed the same data stream followed by it's accompanying checksum, and then the result is checked against the expected residual value to detect the presences of value errors in the packet.

3.5 Bulk Transfer Sequence

This section describes how the various packet types are used when performing a bulk transfer on USB, but your design is only has to be capable of sending or receiving each of the packet types used for bulk transfer. It is not required for your design to keep track of the sequence of packet exchanges described below.

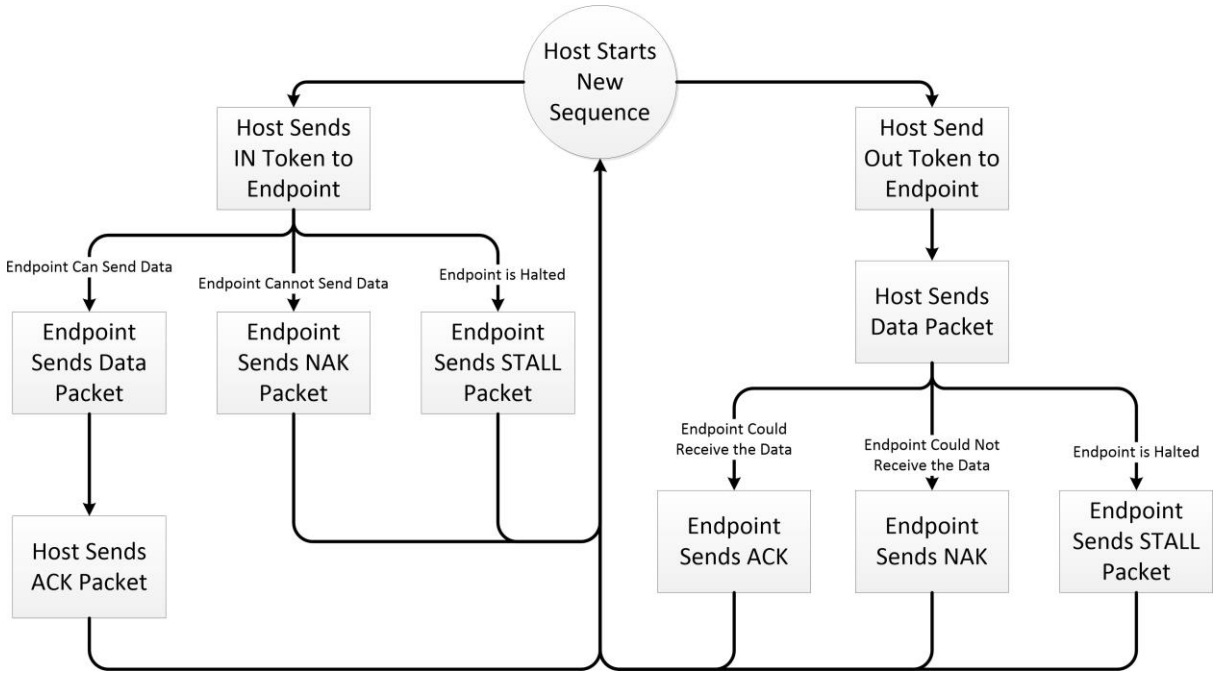

Figure 9: A overall flow-diagram illustrating the sequence of packet exchanges for both Endpoint-to-Host (IN-Token) and Host-to-Endpoint (OUT-Token) bulk-data transfers

USB transfers are always initiated by the Host (Master) device, which is typically a computer of some design. The notion of 'IN' or 'OUT' is from the perspective of the Host device, so 'IN' tokens are used to start reads/retrievals

from an endpoint and ’OUT’ tokens are used to send data to an endpoint. The ’STALL’ handshake is only used when the endpoint is halted due to some serious error, and should not be used by your design. The ’NAK’ handshake is sent as a response when either the endpoint either cannot currently receive the data (in the case of an ’OUT’ transfer) or does not have any data to send to the Host (in the case of an ’IN’ transfer). The ’ACK’ handshake is sent as success response for any transfer.

4 Design Architecture

A full SoC peripheral module contains both the SoC bus related interface module and the functional modules needed to implemented and execute the peripherals communication standard. In this over all design the SoC bus interface is a more fully capable AHB-Lite bus, and the peripheral's communication standard is a focused subset of USB 1.1 (Full-Speed). In order to keep the workload tractable for your team and the time schedule, the USB standard support is focused solely on supporting the basics of Bulk Transfer under the USB 1.1 standard. No USB startup/bus enumeration or other aspects of the standard need to be directly handled in this design.

4.1 USB Full-Speed Bulk-Transfer Endpoint AHB-Lite SoC Module Architecture

The starting architecture your team must follow (other than potential internal signal additions) is depicted in Figure 10.

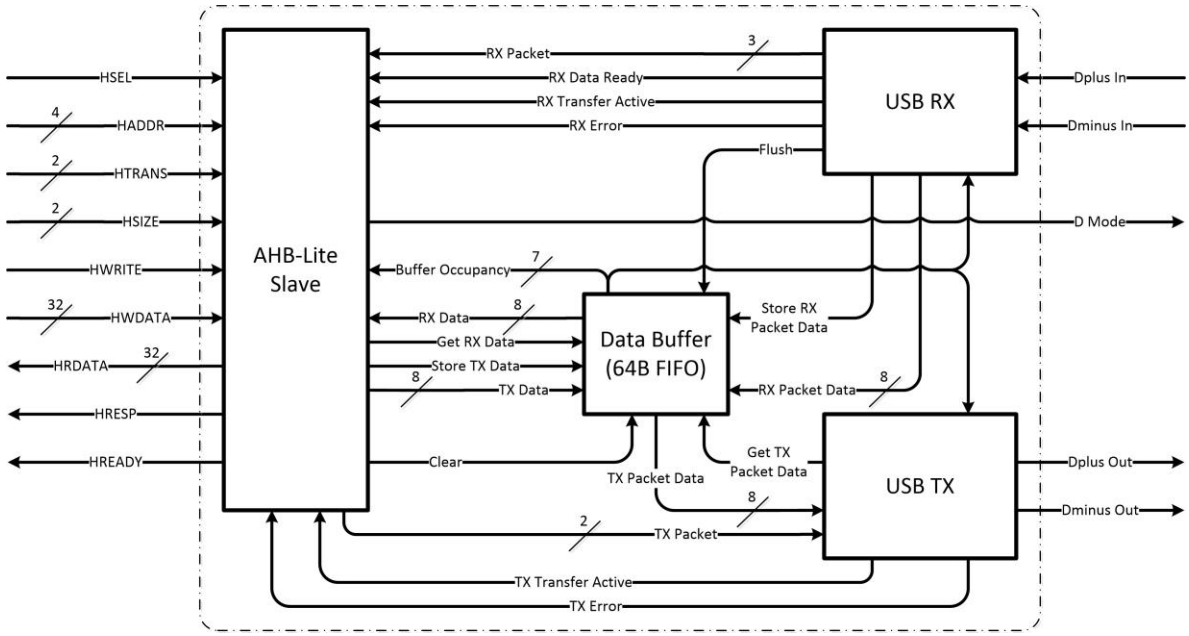

Figure 10: Architecture for USB Full-Speed Bulk-Transfer Endpoint AHB-Lite SoC Module

The top level I/O for the DUT will include all required AHB bus signals along either either Dplus In/Dminus In OR Dplus Out/Dminus Out

You may add, remove, or redefine the internal signals that go between the AHB-lite slave and other blocks in this design. Consequently it is up to you determine how exactly these signals will be used. It should be similar but not necessarily identical to what is presented here. What matters is that your design is capable of sending and receiving the required USB packets and responding appropriately to commands on the AHB-lite bus.

NOTE: Clock and Reset signals are assumed to be sent to the appropriate blocks and are not shown

Similar to during Lab 9, the security ('HPROT') and atomicity ('HMASTERLOCK') signals are absent from this design because they are not needed or relevant given the functionality of the design.

Unlike in Lab 9, HREADY feedback (via 'HREADYOUT') is present on this design. However, even though HREADY feedback is present the slave really only has a valid need for delaying/extending the bus transfers under error conditions as, there should not be any transfers that required longer than one address cycle or one data cycle to properly respond.

All status checking or valid data supply/retrieval is expected to be enforced by software running on the SoC core and via polling-style data status checks.

Furthermore, the burst transfer support is not required for this design and thus the 'HBURST' signal is not present.

4.1.1 List of the Functional Units/Blocks and Purpose

USB RX This module handles all work needed to receive and validity check any of the basic packet types sent from the host to an endpoint during Bulk Transfers.

USB TX This module handles all work needed to correctly send any of the basic packet types sent from an endpoint to the host during Bulk Transfers.

Data Buffer This module handles all work needed to correctly buffer the data that should be sent out during the next endpoint-to-host data transfer or was received during the most recent host-to-endpoint data transfer.

AHB-Lite-Slave This module handles all AHB-Lite-Slave Interface specific functionality.

4.1.2 List of the Top-Level Ports and Purpose

Clk This is the system clock port. It should be connected to a 100 MHz clock.

N_Rst This is the active-low asynchronous system reset signal

HADDR This is used for providing the address (within the slave only) that the transaction involves.

HSEL This is the 'slave' selection signal.

HTRANS This indicates the type of the current transfer.

HSIZE This indicates the size of the transfer.

HWRITE This indicates whether a transaction is a read (logic low value) or write (logic high value).

HWDATA This is used for transferring the data written to the 'slave' device.

HRDATA This is used for transferring the data read from the 'slave' device and can be up to 32-bits wide.

HREADY This is an active-high ready feedback signal from the 'slave' device which is pulled low by the 'slave' if it needs to stall/pause the transaction.

HRESP This is an active-high error feedback signal from the 'slave' device, and must be asserted when there is a transaction error (such as trying to write to read-only addresses).

Dplus In This is the receiving (RX) module's D+ wire connection

Dminus In This is the receiving (RX) module's D- wire connection

Dplus In This is the transmitter (TX) module's D+ wire connection

Dminus In This is the transmitter (TX) module's D−wire connection

D Mode This is an active high-signal that determines if the D+/D- wire pair should be in output/drive mode or receiving/passive mode.

4.1.3 Description of internal interface signals

You are free to add, remove, or modify the internal signals of this design as long as your design exhibits the proper behavior at the top level ports described previously. At the very least, you will have to define some of these signals in more detail than is provided here.

RX Packet When the USB RX receives a packet, this bus will indicate the type of packet received. You are to define the codes that will identify the packet type.

RX Data Ready This signal will indicate when a new packet has been received by USB RX.

RX Transfer Active Indicates when the USB receiver is busy receiving a new packet.

RX Error Indicates when an error has occurred. Unless you implement the CRC check, the only errors will be an invalid sync byte, undefined PID, end-of-packet (EOP) at an unexpected time.

Buffer Occupancy How many bytes of unread data remain in the data buffer.

RX Data Oldest unread data value in the data buffer.

Get RX Data Tell data buffer to pop the current RX Data value off the FIFO queue.

Store TX data Tell data buffer to push TX Data into the FIFO queue.

TX Data Data to be pushed into FIFO. Will eventually be transmitted by USB TX.

Clear Perform synchronous reset of FIFO in preparation to receive new data for transmission.

TX packet Indicates the type of packet to be transmitted by the USB TX. See RX Packet for more details.

TX Transfer Active Signal indicates when USB transmitted is busy sending a packet.

TX Error Indicates an error when processing a transmission request. It is up to you to identify and define possible error conditions.

Flush Perform synchronous reset of FIFO in preparation to receive new data for transmission. Same as [Clear] but initiated by USB RX.

Store FX Packet Data Tell data boffer to push the current value of RX Packet Data into the FIFO queue. Same as [Store TX Packet Data] except that it is initiated by USB RX.

RX Packet Data The most recently received byte of payload data from a data packet, to be pushed into FIFO.

Get TX Packet Data Tell data buffer to pop another value off the FIFO queue.

TX Packet Data Oldes unread data value in the data buffer to be transmitted by USB TX as part of a data packet.

4.2 Required AHB-Lite-Slave Address Mapping

Table 4: Table of the required AHB-Lite-Slave address-to-value mapping

Address	Value Size (Bytes)	Access Mode	Description
0x0	4	R/W	Data Buffer (Buffer has capacity of 64B or 16 4-B words): Writes push byte(s) into the Data Buffer. Reads pop byte(s) from the Data Buffer.
0x4	2	R	Status Register: Value of '1' for bit-0 → New Data is available in the buffer from the Host Value of '1' for bit-1 → An 'IN' token was received from the Host Value of '1' for bit-2 → An 'OUT' token was received from the Host Value of '1' for bit-3 → An 'ACK' was received from the Host Value of '1' for bit-4 → An 'NAK' was received from the Host Value of '1' for bit-8 → Currently receiving a packet from the Host Value of '1' for bit-9 → Currently sending a packet to the Host
0x6	2	R	Error Register: Value of '1' for bit-0 → Error happened during reception of a packet Value of '1' for bit-8 → Error happened during packet sending attempt
0x8	1	R	Buffer Occupancy: Current data buffer occupancy in bytes.
0xC	1	R/W	TX Packet Control Register: Writing a '1' value to this register causes the design to send a Data packet, if enough data is present in the data buffer. Writing a '2' value to this register causes the design to send an 'ACK'. Writing a '3' value to this register causes the design to send an 'NAK'. Writing a '4' value to this register causes the design to send an 'STALL'. All other values should have no functional effect. This must be cleared after the packet is sent.
0xD	1	R/W	Flush Buffer Control Register: Writing a '1' value to this register will clear the Data Buffer. This must be cleared after the buffer is cleared.

Writes to the data buffer address result in pushing a new value (with the amount of bytes pushed based on the HSIZE) into the data buffer. Reads to the data buffer address result in popping the amount of data requested (determined by the HSIZE) from the data buffer. All values are to be handled according to 'Little Endian' notation.

4.3 Valid Design Usage

4.3.1 Host-to-Endpoint Transfers

The valid flow of operations for using this design while expecting a Host-to-Endpoint Transfer is as follows:

1. The SoC core would periodically check the status register to see if a Host-to-Endpoint Token has been received.
2. The SoC core would then make sure the Data Buffer was empty and ready to receive a data packet.
3. The SoC core would then periodically check the status register to see if new data had arrived from a data packet.
4. Once data has been received from the Host, the SoC core will tell the design to send an 'ACK'
5. The SoC core will check the buffer occupancy register to find out how much data was received.
6. The SoC core will then retrieve the data via a sequence of reads.
7. Once all of the data has been retrieved from the buffer, the data-available flag in the status register must be cleared by the design.

4.3.2 Endpoint-to-Host Transfers

The valid flow of operations for using this design while expecting an Endpoint-to-Host Transfer is as follows:

1. The SoC core will make sure the data buffer is currently empty.
2. The SoC core will populate the data buffer with the data to send via a sequence of writes.
3. Once the data buffer currently holds as much data as set for the data payload size, the SoC core will tell the design to send a data packet.
4. The SoC core will periodically check to see when the data packet sending finishes.
5. The SoC core will periodically check that an 'ACK' was received, and perform any error handling if another packet type is seen first.

5 Specifications for the Blocks You Must Design and Implement

5.1 USB Full-Speed Bulk-Transfer Endpoint AHB-Lite SoC Module

5.1.1 Block Description

This is the full design module that connects the dedicated AHB-Lite-Slave interface, the USB specific modules, and the data buffer together to form the full peripheral.

Since there will not be an electronic grading script for this design, the exact names used for the filename and ports does not matter as long as they are logical and descriptive of their intended function.

The target clock rate for this design must be 100 MHz.

5.2 AHB-Lite-Slave Interface

This block handles all AHB-Lite specific work and must be composed of at least the following component modules:

Address Decoder This block should handle the decoding of raw addresses into value specific selection and write control signals.

RDATA Register The value for the HRDATA bus must be registered, this is to prevent critical path lengthening and bus synchronization issues for the AHB-Lite interconnect.

Value Registers Every accessible value that is not fully available via port on the module must be held in a dedicated register. These may be distinctly named or handled via an array of value registers.

5.3 USB RX

This block handles all work related to the reception of packets from the USB Host and must be composed of at least the following component modules:

Control FSM This component must be an FSM that controls the sequence of operations within the USB RX module.

Shift Register This should just be a wrapper or direct usage of your prior flexible serial-to-parallel shift register design.

Timer This component must control the sampling and packet field specific timing and is similar in role as the timer module from the UART lab design.

Decoder This component must handle the decoding of the data bits, as well as any idle and EOP conditions, that are sent to the RX module via the D+/D- signals.

5-bit CRC Checker BONUS This component checks if the CRC field of an arriving Token packet is valid for that packet.

16-bit CRC Checker BONUS This component checks if the CRC field of an arriving Data packet is valid for that packet.

Bit-Stuff Detector BONUS This component determines if a currently arriving bit should be considered a ’stuffed bit’ that should be ignored value wise.

5.4 USB TX

This block handles all work related to the sending of packets to the USB Host and must be composed of at least the following component modules:

Control FSM This component must be an FSM that controls the sequence of operations within the USB TX module.

Shift Register This should just be a wrapper or direct usage of your prior flexible parallel-to-serial shift register design.

Timer This component must control the data bit sending and packet field specific timing.

Encoder This component must handle the encoding of the data bits, as well as any idle and EOP conditions, that are sent from the TX module via the D+/D- signals.

16-bit CRC Generator BONUS This component generates the value for the CRC field of a Data packet that is being sent from the USB TX module.

Bit-Stuffer BONUS This component determines if a bit must be 'stuffed' into the outgoing bit-stream and pauses any data shifting for one bit period in order to allow the bit 'stuffing' to happen.

5.5 Data Buffer

This block handles the buffering of the data packet payloads in FIFO queue fashion. It must be naturally empty before any buffering of data from the Host is allowed, and must be cleared/flushed if there is any error the during the reception of a data packet from the Host. It should never be possible to have the FIFO's read and write pointers wrap around in the design and so the logic can be designed with this in mind.

5.6 Required Workload Distribution

All design and full module verification work must be distributed in the following manner:

- Team Member 1: [Regardless of team size]
  - Design, implement, and use bus model to test the AHB-Lite Slave
  - Design and implement top level test bench.
- Team Member 2:
  - Design, implement, and test the USB TX module and its components
  - Design, implement, and test the data buffer.
  - Set up USB TX test bench so that it can be used for USB RX testing
- Team Member 3:
  - Design, implement, and test the USB RX module and its components
  - Use USB TX test bench when available to test the USB RX

6 Closing Remarks

- It is strongly recommended that you review RTL diagrams, pseudo-code, list of test cases, etc. with course staff as you develop them. Even though these items will be graded, you will need feedback as soon as possible to stay on track.

Bibliography

[1] ARM. *AMBA 3 AHB-Lite Protocol Specification v1.0*. 2006.

[2] Compaq Intel Microsoft NEC. Specification, Universal Serial Bus, 1998.