Cooperative Design Lab
USB Full-Speed Bulk-Transfer Endpoint
AHB-Lite SoC Module
Final Report
ECE 337

Lab Section 4
Team Number 2
Nobelle Tay, Jihan Salsabila, Tsun-Lin Hsia, Mao Huan Huang
TA: Anirudh Sivakumar
December 9th, 2018

1. Design Decision Discussion

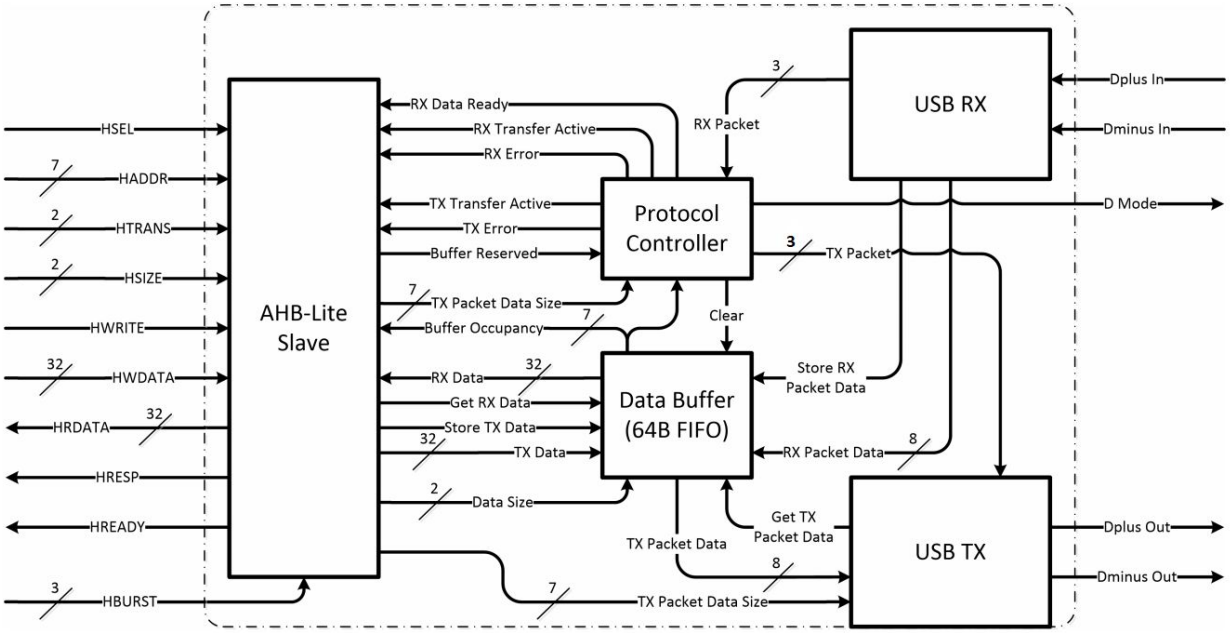


Figure 1: Architecture for USB Full-Speed Bulk-Transfer Endpoint AHB-Lite SOC Module

- **AHB-Lite slave** is responsible for all operations related to AHB-Lite Slave interface functionality which responds to requests that are directed to the bus, and handles the control and data signals for requests that are routed between the two devices involved in the bus transaction.
- **USB RX** is responsible for all operations related to receptions and validity checking of packets from USB host during Bulk Transfers.
- **USB TX** is responsible for all operations related to correct transmission of packets to USB host during Bulk Transfers.
- **Protocol Controller** is responsible for all operations related to correct execution of packet transfer sequence for Bulk Transfers and high-level error/status checking during transfers.
- **Data Buffer** is responsible for all operations related to buffering the data packets that should be transmitted during next endpoint-to-host data transfer or the data packets that was received during the most recent host-to-endpoint data transfer.

1.1 AHB-Lite Slave

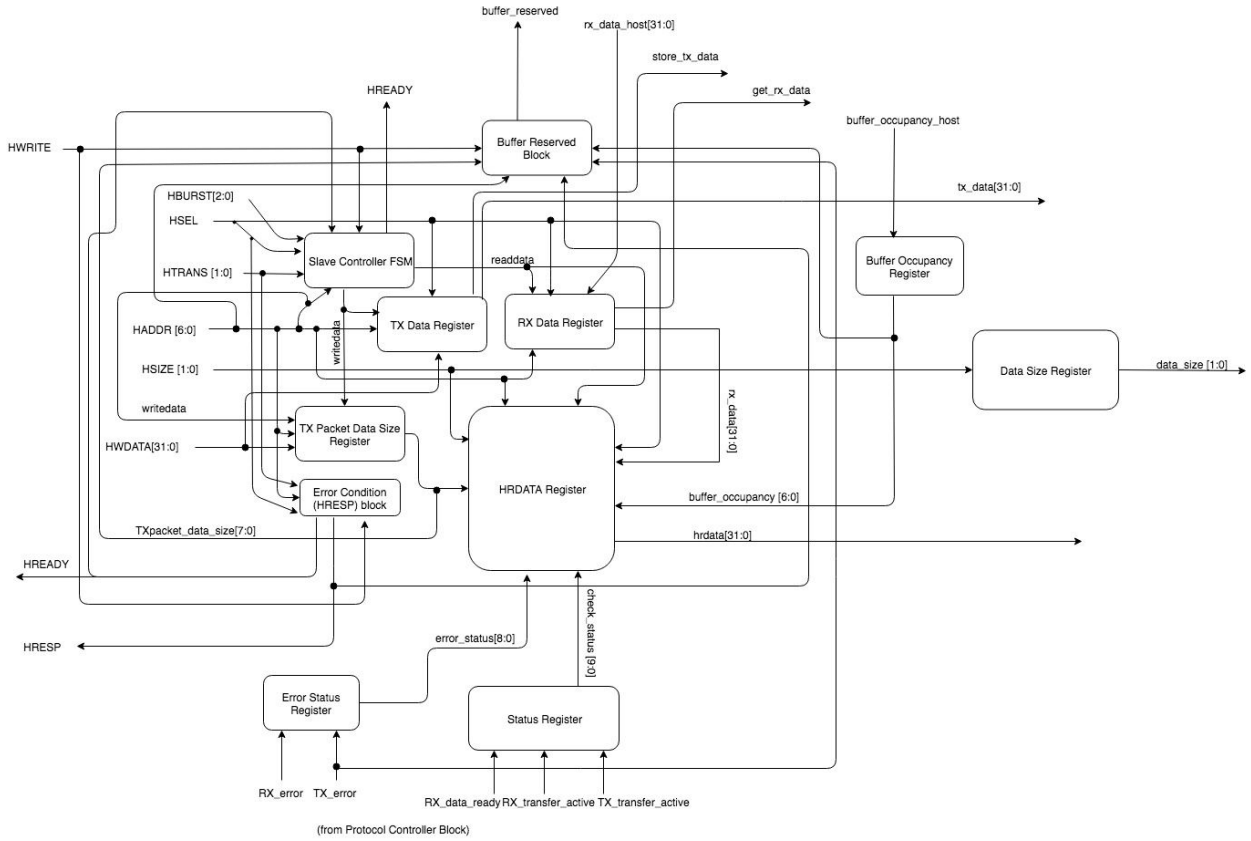


Figure 2: Architecture Block Diagram for AHB-Lite Slave

There are two major components to consider when designing the AHB-Lite Slave: the signals between the slave and the master, and the signals between the slave and the rest of the Endpoint (the other major modules). The behavior of the HRDATA and HRESP signals depending on the input from the master have generally been defined by the AMBA Protocol Specification for a standard AHB-Lite Slave. The HRDATA needs to be registered in order to prevent any bus synchronization issues; therefore, the logic block that determines HRDATA uses the address in the address phase so that HRDATA can be an output in the next clock cycle (data phase). A state machine was designed to enable read/write mode of the AHB as well as facilitate the different burst modes. It consists of an IDLE, READ, WRITE, ERROR, and BUSY state. The IDLE is the default state that the FSM is in when being reset and also acts as a starting point; READ and WRITE state are to enable the read mode (allowing HRDATA as an output signal) and to enable write mode (which will be stored into different registers depending on the address). The ERROR state is determined by HREADY signal (that comes from combinational logic block) and it needs to go to the IDLE state in the next clock cycle because as per the Protocol Specification, the bus should be in idle phase (cannot do read/write) when an erroneous transaction occurs. The BUSY state is necessary when the master decides to delay a variable length burst transaction, meaning that the BUSY state can only transition to READ or WRITE,

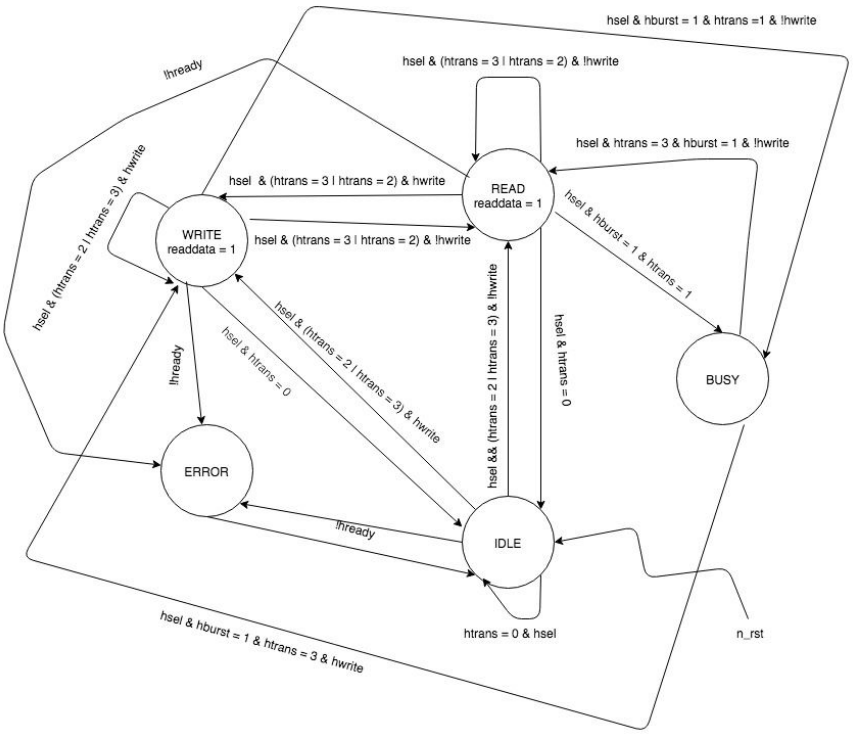


Figure 3: State Diagram for AHB Slave and Mode Controller

and similarly the FSM can only transition to BUSY state from READ or WRITE state. Both burst or single transaction modes from the bus utilize the same READ and WRITE state because the slave operation is identical and it will minimize the amounts of states required.

The buffer reserved signal is needed from the slave to the protocol controller so that the protocol controller can use it to control when RX and TX transactions happen. In our module the buffer reserved signal is high as soon as when the master decides to write into the TX Packet Data Size register, and it will be reserved during any writing transaction (since otherwise the master will only write to the tx_data register). The buffer will no longer be considered reserved when the master stops any writing transaction, which includes when there is a TX error signal, when there is a HRESP signal, or when the TX packet data size is equal to the buffer occupancy signal. The data size signal is a register from the master HSIZE input that will be sent to the data buffer in order to determine how much of data is requested from or being sent to the data buffer. It was decided that the Get RX Data Signal would be sent during the address phase and RX Data would be sent from the data buffer through a combinational block so that it can go through the HRDATA register during data phase. For TX Data operation, since HWDATA will only be available during the data phase, the Store TX Data signal will be available in the address phase, then HWDATA will go through a register during the data phase, delaying the TX Data Output by one clock cycle. This has been coordinated with the operation of the data buffer.

1.2 Protocol Controller

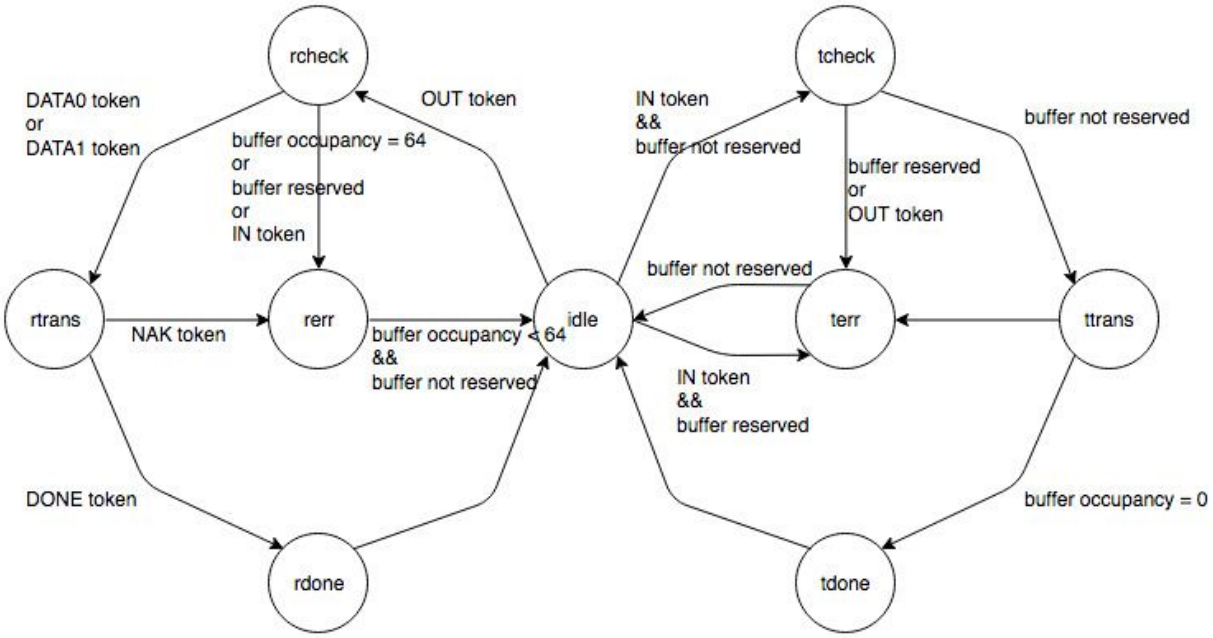


Figure 4: State diagram for protocol controller

The protocol control serves as the brain for the design; it handles data depending on given tokens between AHB-Lite slave, USB RX and USB TX. If given data or tokens at inopportune moments or inappropriate tokens, protocol control also handles errors within the system.

As demonstrated in Fig. 4, the state machine of protocol control handles two different cycles of transactions. Generally, excluding idle, both receive and transmit modes include four states. After receiving the appropriate token from USB RX(OUT for receive and IN for transmit), they go into their respective checking states. In receiving mode, since USB RX accepts data regardless of buffer availability, if buffer is reserved by AHB-Lite slave or is full, protocol control goes into its error state and responds to host by sending a NAK token to USB TX for transmission. However, if either DATA0 or DATA1 tokens are received, it may move on to the transfer state. At this point, data from the host is flowing into the buffer and if USB RX gets indication that said data is done, the DONE token would move protocol into its done state. It's worth noting that the DONE token was not specified in the manual, but instead a new signal agreed upon between protocol and USB RX for better communication. After one clock cycle, the state machine returns to idle. If at any point during the checking or transfer state, an error from host is perceived by USB RX, a NAK token would also send protocol into its error state. To escape from the error state in receive mode, buffer reserved has to go low and buffer occupancy must drop lower than 64(full buffer). If said conditions are met, the state machine returns to idle after one clock cycle.

In transfer mode, the checking state makes sure buffer is not reserved before transmission; if so, protocol goes into transfer mode and unloads the buffer byte by byte to USB TX, otherwise, if buffer is reserved or an OUT token is received indicating a receive cycle, protocol goes into its error state. Subsequently when buffer occupancy drops to zero, the data is assumed to be ready and the done state ensues. After one clock cycle the protocol returns to idle. If at any point the state machines becomes trapped in the error state, the buffer reserved must go low before protocol may return to idle.

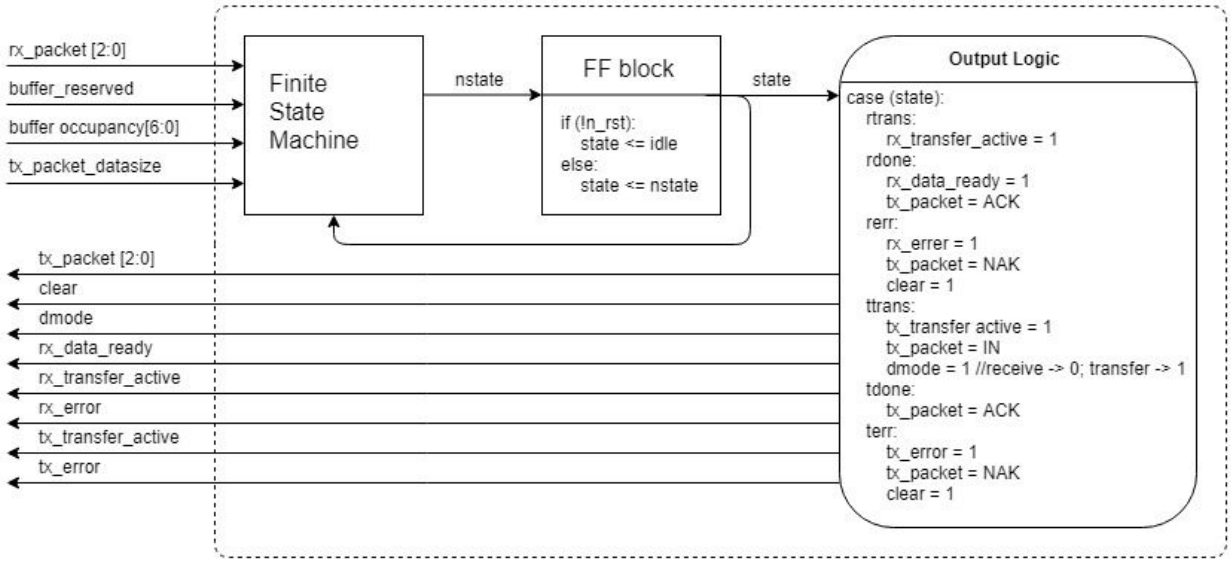


Figure 5: RTL diagram for protocol controller

Protocol Controller only outputs data during error, transfer, and done phases. Intuitively, during the transfer phase, transfer active signals are held high regardless during receive or transmit modes; D mode however is only held high during transfer. In the current design, D mode is held high for output/drive mode and held low for receiving/passive mode. Both done phases produce ACK tokens that's passed onto USB TX for transmission, but data ready is only clocked high for receive mode to indicate the data within buffer is ready. Lastly, in both error states, a NAK token is transmitted and respective errors are held high; a clear signal is outputted to buffer to flush out any remaining data. Tx packet data size was not utilized in protocol controller for this design.

1.3 Data Buffer

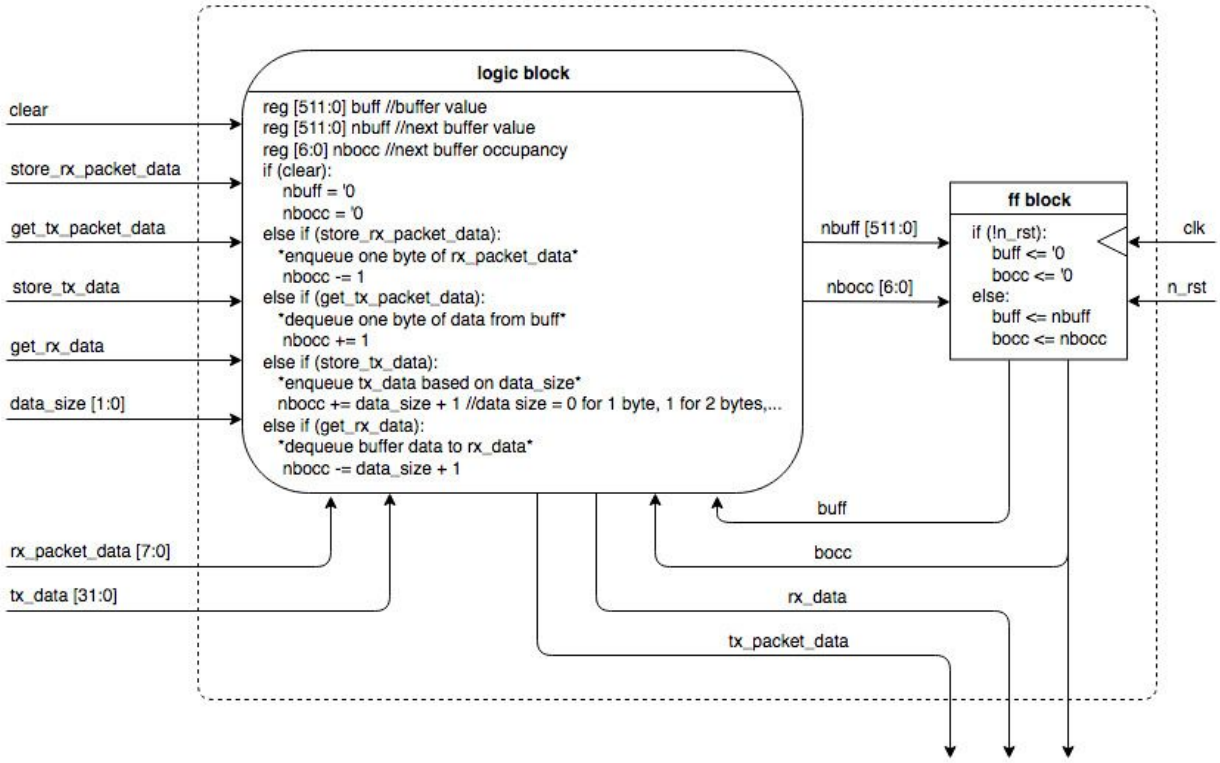


Figure 6: RTL diagram for data buffer

The data buffer is a FIFO buffer than enqueues and dequeues data coming in and out the system. It was not run via a state machine, hence the lack of a state diagram. At any point if a signal is given, the corresponding output would be available. This design allows AHB-Lite to send a flag and retrieve the needed data within the next clock cycle. The only registered data in data buffer are the buffer value itself and buffer occupancy.

During normal procedures, the store rx packet data signal notifies the buffer to read incoming data from USB RX in the rx packet data wire; the get tx packet data signal tells buffer to dequeue its data into USB TX via tx packet data. On the AHB-Lite slave side, store tx data and get rx data are requests to either dequeue data from buffer or enqueue data from AHB-Lite slave. Depending on data size, the buffer can either read or send one to four bytes from or to AHB-Lite slave. Lastly, the clear signal, aforementioned in protocol controller, clears the buffer of any data during the course of an error.

1.4 USB RX

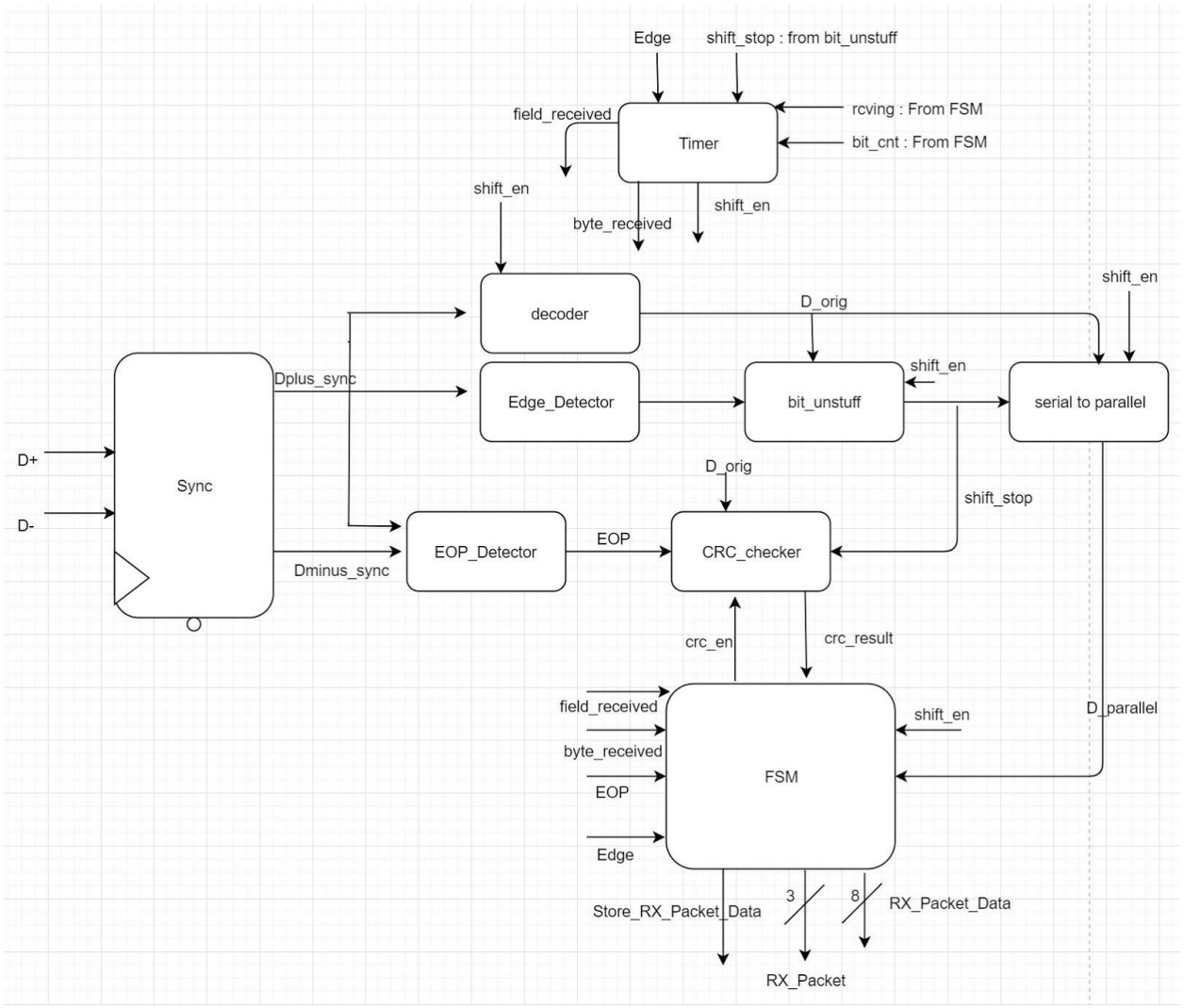


Figure 7, Architecture for USB RX Module

Input D+ and D - come in and synchronized before processing. D+_sync goes to decoder and check whether D+ is the same as the previous D+, if they're the same then D_orig will be 1, otherwise will be 0. There are two detector, which are Edge_Detector and EOP_Detector. Edge_Detector checks every time the data goes from 0 to 1 or 1 to 0 and EOP_Detector checks if D+ and D- is 0 then it's EOP. Bit_unstuff checks if there are six 1's and it will send a shift_stop to Timer that will skip one bit at a time. Sr_8bit is to combine 8 bits from serial to 8 bits parallel by using flex_stp shift register.

Timer controls timing. In Timer, shift_enable control everything to work at the same pace, so it goes to every block except detectors and CRC_checker. By receiving status from FSM, Timer will decide if FSM should go to the next state. To be more specific, the signal that will send it from FSM to Timer are "receiving" and "bit_cnt", which tells Timer that in this state, FSM is still receiving data, and bit_cnt shows how many bits in that states so that Timer can use it in flex_counter to count.

CRC_Ckecker is to check the CRC values when the state in FSM goes to Chk_CRC, FSM will send crc_en, which will start to check CRC.

The major part of the design is FSM. FSM go through data's, check and sort what type of data and then send out to buffer and protocol controller. Started from IDLE, first check if it's a right sync byte and right pid, and then check what kind of packet it is. If it's token packet, then check the 7-bit address, 4-bit endpoint, and 5-bit CRC. If it's a data packet, then start to store bytes, and after that, check 16-bit CRC. Else, the packet will send out with ACK, NAK, or STALL. If there is pid_error, EOP_error, invalid address, they will be label as STALL. The output of USB RX will be Store_RX_Packet_Data, 8-bit RX_Packet_Data, and 3-bit RX_Packet. Store_RX_Packet_Data tells Data Buffer if it should start to store RX_Packet_Data, and RX_Packet goes to protocol controller. If RX_Packet is 3'b000, it will be IDLE state, if it's 3'b001 or 3'b010, it will be OUT or IN token packet, if it's 3'b011 or 3'b100, it will one of two available data packet, if it's 3'b101 or 3'b110, it will be ACK or NAK, and the rest will be Error.

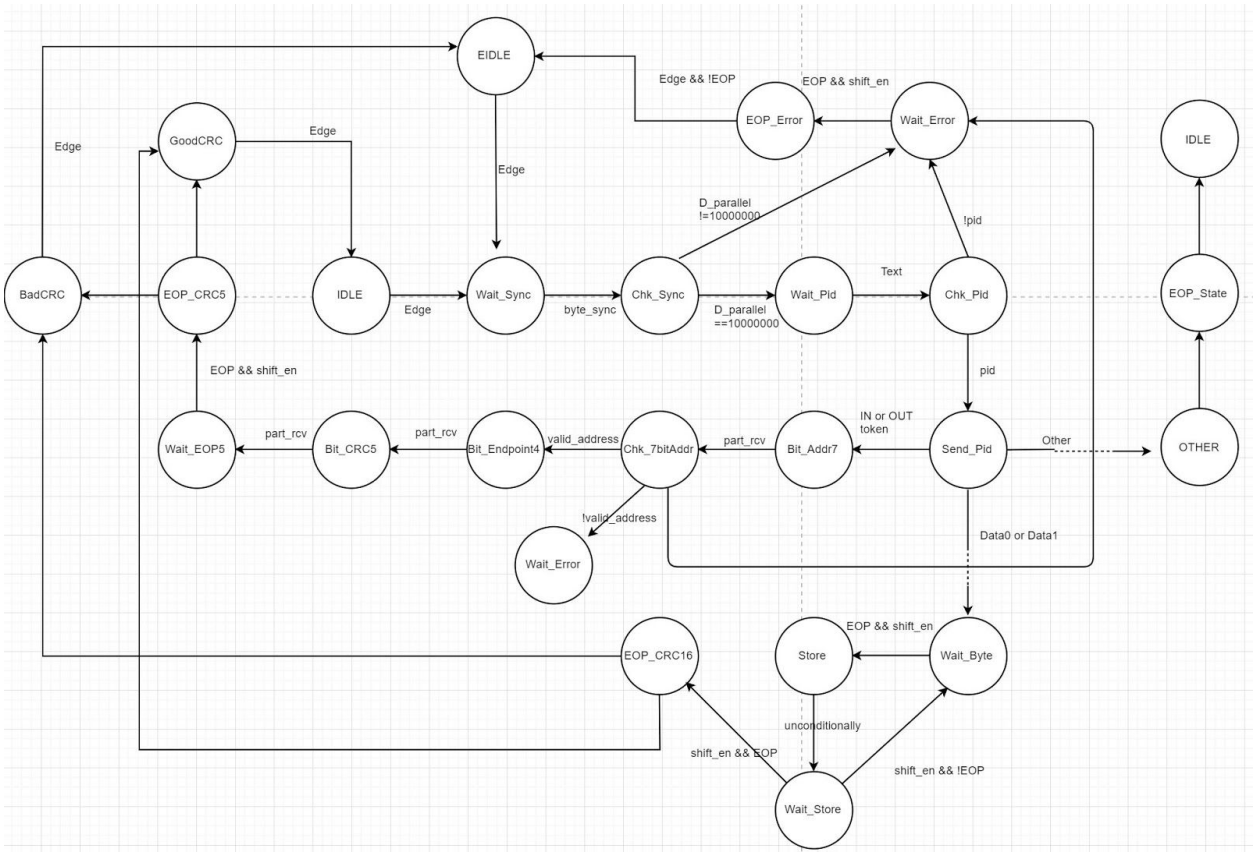


Figure 8: State Diagram of FSM submodule

1.5 USB TX

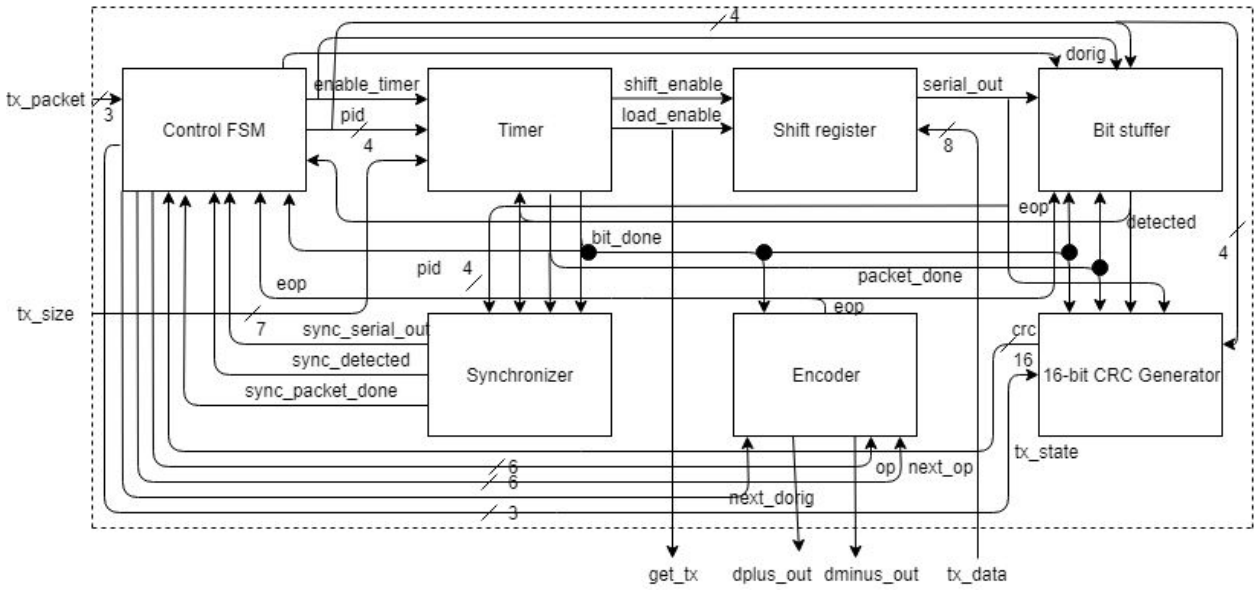


Figure 7: Architecture for USB TX Module

The Control FSM controls the sequence of operations within the USB TX module. It needs an input signal called **tx_packet** from Protocol Controller module**,** which can be IDLE_STATE, DATA0_STATE, DATA1_STATE, ACK_STATE and NAK_STATE to know the packet type it is going to transmit. The **tx_packet** signal can be both pulse signal or continuous signal as the state of the USB TX module, output signal **tx_state** will only depend on it again when it is in IDLE_STATE, or when it has finished sending the packets (when **eop** input signal is asserted). When the USB TX module is in any state other than IDLE_STATE, the output signal **enable_timer** will be asserted and the output signal **pid** will be the packet ID corresponding to the packet type it is transmitting. During transmission of packets from endpoint to host, the input signal **bit_done** tells the Control FSM to transmit another bit whereas the output signals **op** and **next_op** controls the field type it is transmitting in the packet, and the output signals **dorig** and **next_dorig** represent the bit to be encoded and transmitted. When the USB TX module is transmitting data packets, since the sync byte and pid byte, which are 16 bits altogether, have to be transmitted prior to the transmission of data field, the Shift Register module that shifts the data to be transmitted will still start working once the USB TX module knows it is going to transmit data packet (tx_state = DATA0_STATE or tx_state = DATA1_STATE), but the serial_out signal from Shift Register will be delayed for 16 bits to synchronize the serial_out signal to when it is supposed to transmit data field, and this synchronized signal is the input signal **sync_serial_out** from Sync Data module. Similarly, the detected signal from Bit Stuffer module and the packet_done signal from Timer module will be synchronized to when it is supposed to transmit data field, and the synchronized signals are the input signal **sync_detected** and **sync_packet_done** from Sync Data module respectively. Despite that, the input signal **detected** is still needed for the Control FSM module to stuff bits during the transmission of **crc.**

The Timer module controls the shifting of the data bits and the timing of sending the bits of the entire packet. The Timer module only functions when the input signal **enable_timer** is asserted. The output signal **bit_done** is asserted every 8 clock cycles once the Timer module starts working to have transmission frequency around 12 MHz when the system clock frequency is 100 MHz. The input signal **tx_size** represents the number of data bytes to be transmitted and the output signal **packet_done** will be asserted once all the data bytes has been transmitted. The shifting of the data bits will only start if the USB TX module is in DATA0 or DATA1 state, as indicated by the input signal **pid**. During the shifting of the data bits, the output signal **load_enable** will be asserted everytime first bit of the data byte is supposed to be shifted, whereas the output signal **shift_enable** will be asserted everytime other bits of the data byte is supposed to be shifted. The shifting of the data bits delays for one bit when the input signal **detected** is asserted so that 0 bit can be stuffed in the transmission of data bits.

The Shift Register module shifts the data bits during transmission of data bits. When the input signal **load_enable** is asserted, a new data byte, **tx_data**, is loaded and its least significant bit is continuously shifted out as the output signal **serial_out** when the input signal **shift_enable** is asserted.

The Bit Stuffer module determines if a bit must be stuffed into the outgoing bit-stream and outputs a signal to pause data shifting for one bit period in order to allow the bit stuffing to happen. The Bit Stuffer module only functions when the input signal **bit_done** is asserted and it is reset when the packet is done being processed (**eop** is asserted). When the shifting of data bits has just started (**enable_timer** changes from low to high), the **pid** input signal is needed to indicate the number of ones that are accumulated before the shifting of the data bits. Bit Stuffer module processes the input signal **serial_out** when the shifting of data bits is happening (**packet_done** is low) and processes the input signal **dorig** when all the data bits have been shifted out (**packet_done** is high). The only output from Bit Stuffer module is the **detected** signal which will be asserted if bit stuffing is necessary.

The 16-bit CRC Generator module generates the value for the CRC field of the data packet that is being transmitted. The 16-bit CRC Generator module only shifts in the input signal **serial_out** when the packet type is DATA0 or DATA1, as indicated by the input signal **pid**, when the input signal **bit_done** is high and when bit stuffing in data bits is not happening (**detected** is low). After all the data bits have been shifted out (**packet_done** is high), it shifts in another 16 zeros. Then, the output signal **crc** holds the computed CRC value until the USB TX module is idle again (**tx_state** = IDLE_STATE), where it resets to zeros (initial register value).

The Encoder module handles the encoding of the data bits, as well as any idle and eop conditions, that are transmitted from the USB TX module via the D+/D- signals. The Encoder module only encodes the input signal **next_dorig** using NRZI encoding when the input signal **bit_done** is high and holds the value until the **bit_done** signal is high again. When the bit type to be encoded is idle bit (**next_op** = IDLE), the output signal **dplus_out** should be high whereas the output signal **dminus_out** should be low. When the bit type to be encoded is eop bit (**next_op** = EOP1 or EOP2), the output signals **dplus_out** and **dminus_out** should be both low. The output pulse signal **eop** is asserted when second eop bit is transmitted (**next_op** == EOP2).

2. Summary of Success Criteria Status

2.1 General Criteria

1. Test benches exist for all top-level components and the entire design. The test benches for the entire design can be demonstrated or documented to cover all of the functional requirements given in the design specific success criteria.
    a. Status: Partially Satisfied
    b. Rationale: There were working test benches for AHB-Lite Slave Module, Protocol Controller Module, Data Buffer Module and USB TX Module to cover all the related functionalities. There are working test benches for USB RX Module but the code doesn't function correctly.
2. Entire design synthesizes completely, without any inferred latches, timing arcs, and, sensitivity list warnings.
    a. Status: Partially Satisfied
    b. Rationale: AHB-Lite Slave Module, USB TX Module, and USB RX Module synthesized completely without any inferred latches, timing arcs, and, sensitivity list warnings. The Protocol Controller and Data Buffer could not be mapped for unknown reasons.
3. Source and mapped version of the complete design behave the same for all test cases. The mapped version simulates without timing errors except at time zero.
    a. Status: Partially Satisfied
    b. Rationale: AHB-Lite Slave Module, USB TX Module, and USB RX Module has the source and mapped version behaving the same for all test cases with only time zero timing errors. The Protocol Controller and Data Buffer could not be mapped, however the synthesized version works with all test cases.

2.2 Functional Criteria

1. Demonstrate via test bench driven simulation that the overall design correctly allows an attached AHB-Lite SoC to retrieve data payloads from a USB Host via the USB 1.1 interface and respective Bulk Transfer protocol.
    a. Status: Partially Satisfied
    b. Rationale: The AHB-Lite Slave module part can correctly function and protocol controller can function as well, but protocol controller is not function detail enough that the overall result doesn't function correctly.
2. Demonstrate via test bench driven simulation that the overall design correctly allows an attached AHB-Lite SoC to supply data payloads to a USB Host via the USB 1.1 interface and respective Bulk Transfer protocol.
    a. Status: Partially Satisfied

    b. Rationale: The AHB-Lite Slave module part can correctly function and protocol controller can function as well, but protocol controller is not function detail enough that the overall result doesn't function correctly.
3. Demonstrate via test bench driven simulation that the USB RX module correctly handles the reception of the USB packet types received by an endpoint during USB bulk transfer sequences.
    a. Status: Partially Satisfied
    b. Rationale: RX CRC didn't check correctly, so there is no totally correct packet demonstrated.
4. Demonstrate via test bench driven simulation that the USB TX module correctly handles the sending of the USB packet types sent by an endpoint during USB bulk transfer sequences, excluding the 'STALL' packet type.
    a. Status: Satisfied
5. Demonstrate via test bench driven simulation that the AHB-Lite Interface module correctly handles singleton and burst transfers according the AHB-Lite standard, including proper transfer-level error handling, and implements the address mapping/value access rules outlined in the design manual.
    a. Status: Satisfied
6. Demonstrate via test bench driven simulation that the Data Buffer (FIFO) correctly stores data provided by the SoC (via the AHB-Lite interface) and correctly provides that data to the USB TX module during endpoint-to-host data transfers.
    a. Status: Partially Satisfied
    b. Rationale: USB TX functions well but Data Buffer doesn't provide the data on the right time.
7. Demonstrate via test bench driven simulation that the Data Buffer (FIFO) correctly stores data from the USB RX module during host-to-endpoint data transfers and correctly provides that data to the SoC (via the AHB-Lite interface).
    a. Status: Partially Satisfied
    b. Rationale: USB RX didn't check that well, and data buffer didn't communicate with SoC perfectly.

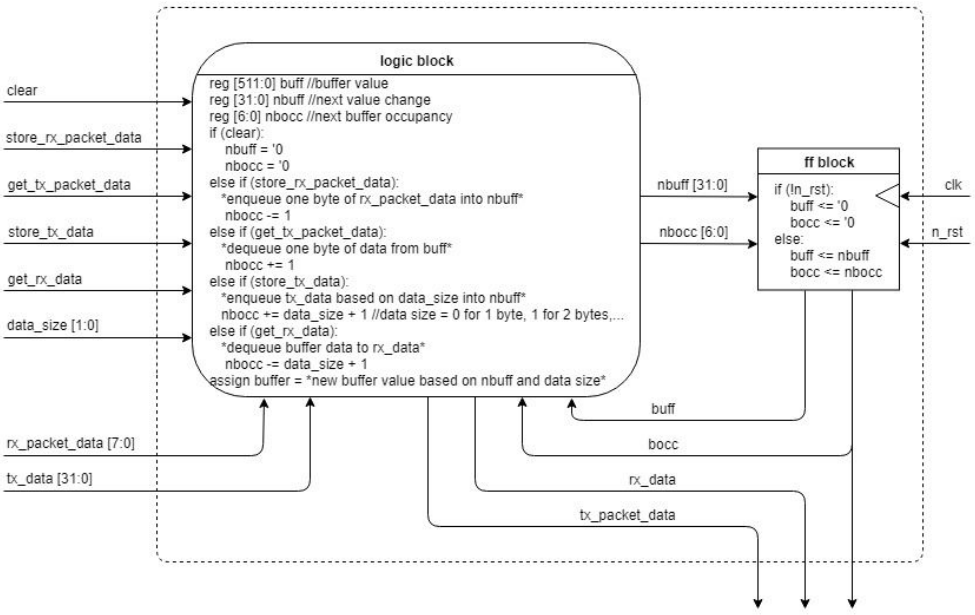
3. Design Area Analysis

Area Summaries of Each Section from Synthesis Report

Module Name	Total Cell Area (µm^2)	Flip Flop Cell Area (µm^2)	Combinational Cell Area (µm^2)
USB RX	217278.00	99792.00	117486.00
USB TX	335160.00	185328.00	149832.00
Protocol Controller	27495.00	6336.00	21159.00
Data Buffer	4395483.00	822096.00	3573873.00
AHB-Lite Slave	213840.00	153648.00	60192.00

As the design areas get bigger, it doesn't efficiently use the chip so the chip will have less space to fit more function, also the power consuming will be higher because of the large area. This will have an effect on the performance and efficiency of the design.

Currently, the largest contributor to cell area is the data buffer; that's mainly the result of stuffing the entire data buffer into registers. Instead of doing so, the same effect, given current timing constraints, can be still be achieved by only storing the changing value into an 8 bit wide register. The register will be updated every clock cycle, and the value stored within can be updated to the buffer at last byte given by buffer occupancy.



4. Timing Analysis

Critical Path Summaries of Each Section from Synthesis Report

Module Name	Delay Values (ns)	Comb. Cell Count (ns)	Start Point	Endpoint
USB RX	3.30	464	FSM/state_reg[2]	receiving(output port)
USB TX	2.45	609	state[0] (rising edge-triggered flip-flop)	get_tx (output port)
Protocol Controller	2.21	90	state_reg[0](rising edge-triggered flip-flop)	Rx_ready (output port)
Data Buffer	3.40	9435	Store_rx_packet_data	rxdata[31]
AHB-Lite Slave	3.32	238	haddr[3] (input port)	get_rx_data (output port)

According to the critical path summaries (from the mapped reports), the Data Buffer has the longest critical path of 3.40 ns, which means that this is the path that has the largest potential of a timing violation. Depending on if our propagation delay is smaller than 3.40 ns, this can cause the circuit to have a propagation delay - which is bad in many ways, including metastability and chip failure as a worst case scenario.

There are a few ways to optimize that path. First, we can reduce the combinatory logic blocks of the Data Buffer RTL to be as short as possible, i.e. possibly implementing more registers where there were initially none in that path. For example, the signal store_rx_packet_data can be registered instead of leaving it as a combinatory signal, and that would decrease the critical path. However, that would delay the signal by a clock cycle which could potentially have other effects in how the code works; therefore, the decision to register a signal should include meticulous attention to how it interacts with other signals and blocks.

References Cited

ARM. AMBA 3 AHB-Lite Protocol Specification v1.0. 2006.

"ASIC Design Laboratory Cooperative Design Lab (CDL) USB Full-Speed Bulk-Transfer Endpoint AHB-Lite SoC Module Design Manual (4-Person Team)." ECE 337 Fall 2018 Course Website, 2018, mycourses.purdue.edu/bbcswebdav/pid-12242945-dt-content-rid-90994013_1/courses/wl_21734.201910/Lab%20Documents/CDL/usb_bulk_manual_4person.pdf.