Chapter 2

UART Protocol Development

2.1 UART Protocol Description

UART stands for Universal Asynchronous Receiver / Transmitter. UART is also referred as Serial Port, RS-232 Interface or COM Port. A Properly Configured UART Setup can work with different types of Serial Protocols which involves Transmitting and Receiving Serial Data. Data is being Transferred Bit by Bit on Set of Wires thus can work on Low power which further reduce cost of Implementations.

Formally,UART is Communicates on Asynchronous Serial Communications with Configurable Speeds.Asynchronous itself means there is no requirement of Clock signal to Synchronize. In Order to Communicate UART Utilizes Baud Rate(Hz) it's Basically the rate at which Information is being Transferred. Note :- Both Rx & Tx needs to be at same Baud Rate in Order to Communicate. Failure to do so may affect the Transmission and Receiving of Data which may cause Discrepancies During Data Handling. Allowable Tolerance for Baud Rate is 10% before timing of the Data Bits gets Off. It is considered as Robust Communication Methods used for FPGA Debugging as it has minimum Protocol overheads.[3]

2.1.1 Data Frame Format

UART Transmission Takes Place in the form of Packet. A Packet , which Connects Transmitter and Receiver it mainly consists of Start Bit, Data Bits , Parity Bit & Stop Bit Refer Figure - 2.1.

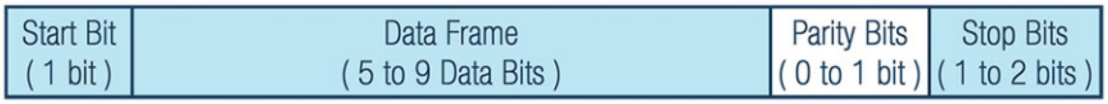


Figure 2.1: UART Frame Format [2]

- **Start Bit :-** When not in use, the UART data transmission line is typically maintained at a high voltage level. The transmitting UART pulls the transmission line from high to low for one (1) clock cycle to initiate the data transfer. The receiving UART starts reading the data frame's bits at the baud rate frequency as soon as it notices the change from high to low signal.[7]

- **Data Frame :-** The actual data being sent is contained in the data frame. If a parity bit is employed, its length can range from five (5) to eight (8) bits. The data frame can have a length of nine (9) bits if no parity bit is employed. The least important bit is often sent first when sending data.[7]

- **Parity Bit :-** The evenness or oddness of a number is referred to as parity. The receiving UART can determine whether any data has changed during transmission by using the parity bit. Long-distance data transmissions, mismatched baud rates, and electromagnetic radiation can all alter bits.

  Following the reading of the data frame, the receiving UART counts the number of bits that have a value of 1 and determines whether the sum is odd or even. The data frame's 1 or logic-high bit should add up to an even value if the parity bit is set to 0 (even parity). The 1 bit or logic highs in the data frame should add up to an odd number if the parity bit is a 1 (odd parity).

  The UART is aware that the transfer was error-free when the parity bit matches the data. However, the UART is aware that bits in the data frame have changed if the parity bit is a 0 and the total is odd, or if the parity bit is a 1 and the total is even.[7]

- **Stop Bit :-** The sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) in order to indicate the end of the data packet.[7]

2.2 Top Level Block Diagram Of UART Rx and Tx

A UART IP Core which accepts Serial Inputs , UART-Rx Block Which converts Serial Data into Parallel Form and same parallel data is feed into UART-Tx Block which Converts Parallel data into Serial UART Data Frame Format. This Loop back Mechanism can Easliy Implemented By inputing ASCII Characters from Keyboard Which gets into UART-Rx and LoopBack to UART-Tx which is connected to GTKterm and Should be able to see our typed character on GTKterm . This mechanism verifies Both Rx and Tx Cores. Refer Figure - 2.2

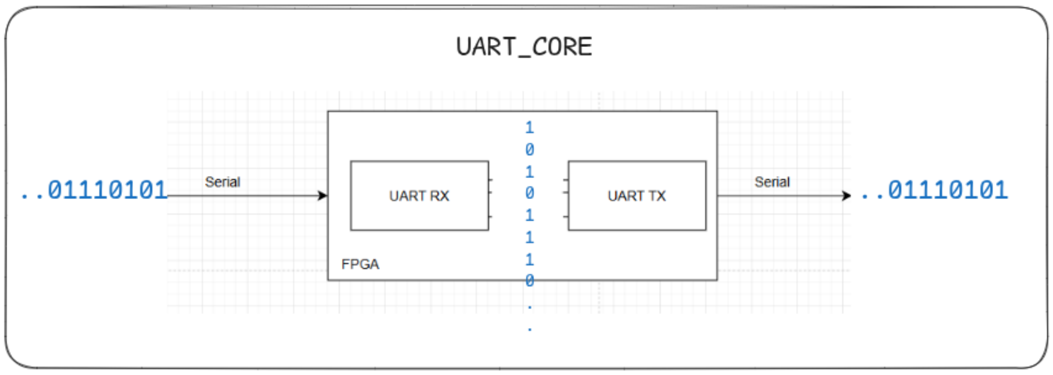


Figure 2.2: UART Block Level Diagram

2.3 Hardware Design Blocks and DataFlow

2.3.1 UART-RX

A state machine will be used by the UART receiver to record the data it receives. First, the receiver searches for the start bit's dropping edge. This signifies the transmission of a byte. After that, it waits for a half-second to align with the UART data's center. Because transitions are less likely to occur and a good sample of the UART data is more likely to be obtained, the center of the bits is the best place to use.[3]

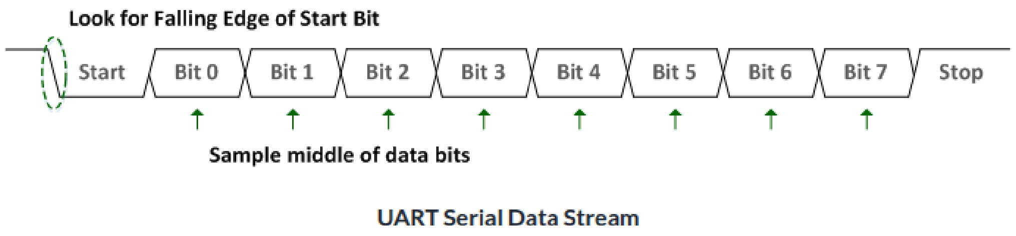


Figure 2.3: UART-RX Sampling Points [3]

Following center alignment Refer Figure - 2.3, the receiver takes a sample of the line's data and records each bit in a byte register. To keep track of which bit is being received, it advances an index. After receiving a stop bit and all eight data bits, it goes back to the IDLE Refer Figure - 2.4 state to await the subsequent data byte.

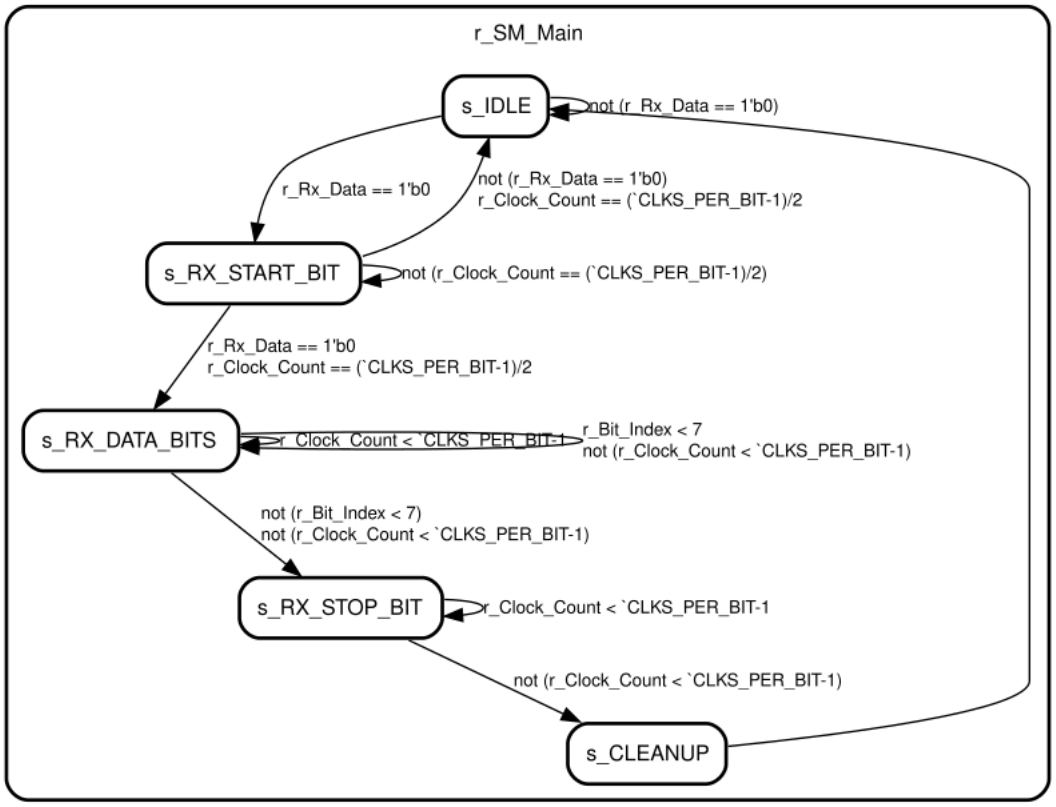

r_SM_Main
s_IDLE
not (r_Rx_Data == 1'b0)
r_Rx_Data == 1'b0
not (r_Rx_Data == 1'b0)
r_Clock_Count == (`CLKS_PER_BIT-1)/2
s_RX_START_BIT
not (r_Clock_Count == (`CLKS_PER_BIT-1)/2)
r_Rx_Data == 1'b0
r_Clock_Count == (`CLKS_PER_BIT-1)/2
s_RX_DATA_BITS
r_Clock_Count < `CLKS_PER_BIT-1
r_Bit_Index < 7
not (r_Clock_Count < `CLKS_PER_BIT-1)
not (r_Bit_Index < 7)
not (r_Clock_Count < `CLKS_PER_BIT-1)
s_RX_STOP_BIT
r_Clock_Count < `CLKS_PER_BIT-1
not (r_Clock_Count < `CLKS_PER_BIT-1)
s_CLEANUP

Figure 2.4: UART-RX FSM Diagram

2.3.2 UART-TX

To go through the data stream, the transmitter employs a state machine. The start bit comes first, followed by the data bits and finally the stop bit. Both where you are in the particular bit and where you are in the entire data stream must be monitored. That's what the state machine will do.

The process of returning a data stream to its original source without alteration is known as loopback. Characters received in this scenario will be echoed back to the computer via the transmitter. This enables the terminal screen to display the keys you press.

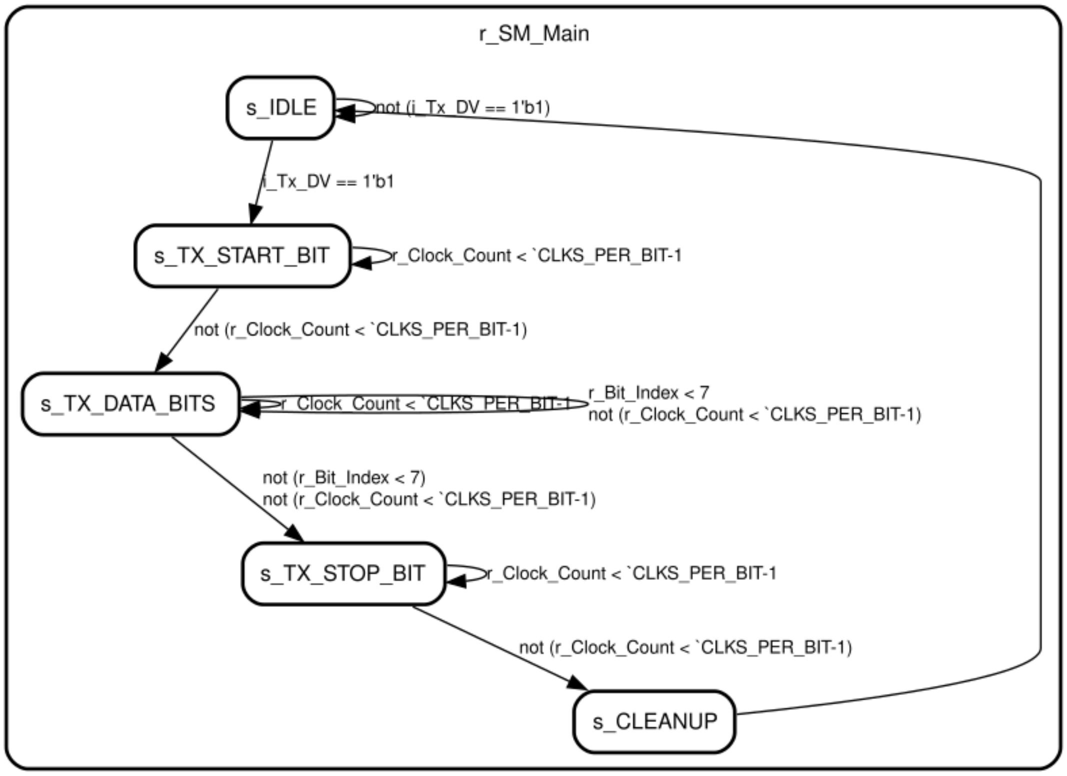

r_SM_Main
s_IDLE
not (i_Tx_DV == 1'b1)
i_Tx_DV == 1'b1
s_TX_START_BIT
r_Clock_Count < `CLKS_PER_BIT-1
not (r_Clock_Count < `CLKS_PER_BIT-1)
s_TX_DATA_BITS
r_Clock_Count < `CLKS_PER_BIT-1
r_Bit_Index < 7
not (r_Clock_Count < `CLKS_PER_BIT-1)
not (r_Bit_Index < 7)
not (r_Clock_Count < `CLKS_PER_BIT-1)
s_TX_STOP_BIT
r_Clock_Count < `CLKS_PER_BIT-1
not (r_Clock_Count < `CLKS_PER_BIT-1)
s_CLEANUP

Figure 2.5: UART-TX FSM Diagram

2.3.3 UART-TOP

UART-TOP module instantiates uart-tx & uart-rx for loopback mechanism as per our main block diagram . Top module inputs Serial data and outputs serial data. uart-tx has valid signal when asserted data is valid this uart-tx can start the process for Serialization based on UART Data Frame.

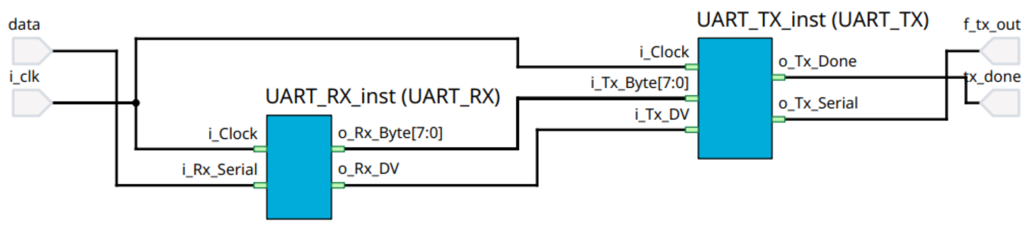
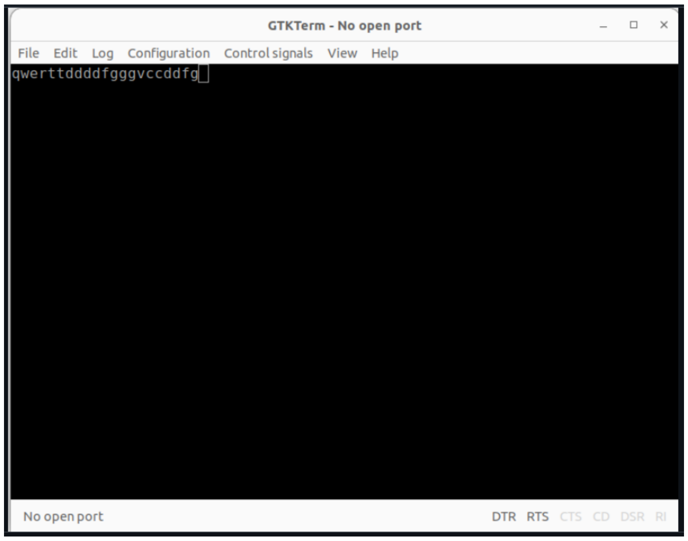


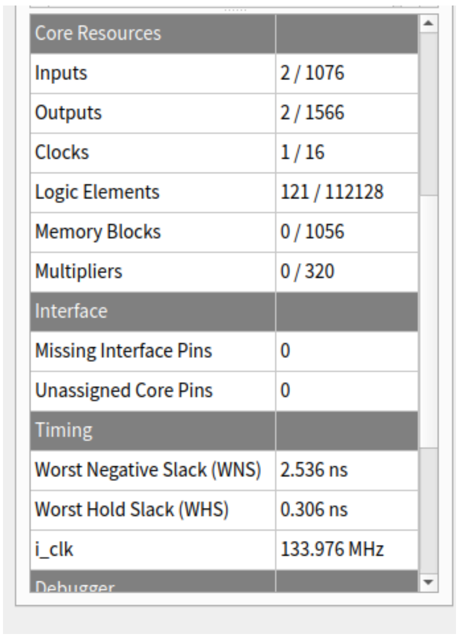
Figure 2.6: UART-TOP Diagram

2.4 Results and Logic Usage



```
qwerttddddfgggvccddfg
```

Figure 2.7: Output from UART-TOP Module on GTKTerm As per given Inputs from Keyboard



Core Resources	
Inputs	2 / 1076
Outputs	2 / 1566
Clocks	1 / 16
Logic Elements	121 / 112128
Memory Blocks	0 / 1056
Multipliers	0 / 320
Interface	
Missing Interface Pins	0
Unassigned Core Pins	0
Timing	
Worst Negative Slack (WNS)	2.536 ns
Worst Hold Slack (WHS)	0.306 ns
i_clk	133.976 MHz
Debugger	

Figure 2.8: Logic Elements used & Timing Measurement (-ve Slack Should be Avoided)

Chapter 3

APB Protocol Development

3.1 APB Protocol Description

The APB protocol's architecture aims to provide low power consumption and latency while preserving simplicity. GPIO, UART, timers, SPI, and other slow-speed modules are examples of low-performance peripherals that typically use APB. APB is a crucial component of many SoCs since it acts as a bridge between these low-speed peripherals and higher-speed buses like AHB or AXI.[8]

3.1.1 Key Features of APB Protocol

- **Single-Clock Cycle Transfers:** Transfers occur in one clock cycle, ensuring low-latency operation. Transfers occur in one clock cycle, ensuring low-latency operation.
- **No Pipeline Stages:** APB's non-pipelined design reduces complexity and ensures simple timing control.
- **Low Power Consumption:** Ideal for peripherals where power consumption is a concern.
- **Simple Control Signals:** APB uses a minimal set of control signals, making integration easy.

3.1.2 Signal Descriptions of APB Protocol

1. Clock and Reset Signals

- PCLK: The clock signal that keeps everything on the bus in sync.
- PRESETn: An active-low reset signal that sets up all of the bus's peripherals.

2. Address and Control Signals

- PADDR: The address bus that indicates the transaction's target register or peripheral memory location.

- PWRITE: A control signal that establishes the data transfer's direction. A write is denoted by a high, and a read by a low.
- PSELx: To identify which peripheral (or slave) the master is aiming at, APB uses PSELx.
- PENABLE: The access phase begins with this signal. It guarantees that data transfer takes place at the appropriate time.

3. Data Signals

- PWDATA: transports the information during a write transaction from the master to the peripheral.
- PRDATA: During a read transaction, the peripheral puts the data on this bus.

4. Handshake and Status Signals

- PREADY: The slave sends out a handshake signal when it is prepared to finish the data transfer.
- PSLVERR: A signal indicating whether a transaction error occurred. Transfers that are incorrect or mismatched may be flagged.

3.1.3 Example: Write and Read Transactions in APB

Write Transaction

Consider a scenario where a microcontroller (master) intends to write data to a timer peripheral (slave) using the APB protocol. The transaction occurs in two phases:

Setup Phase:

- The master asserts `PSELx` to activate the target timer peripheral.
- The address of the timer register is placed on the `PADDR` bus.
- The signal `PWRITE` is set to high, indicating a write operation.

Access Phase:

- The master drives `PENABLE` high to initiate the data transfer.
- The write data is placed on the `PWDATA` bus by the master.
- The slave asserts `PREADY` once it successfully receives the data.
- The master then deasserts `PENABLE`, completing the write transaction.

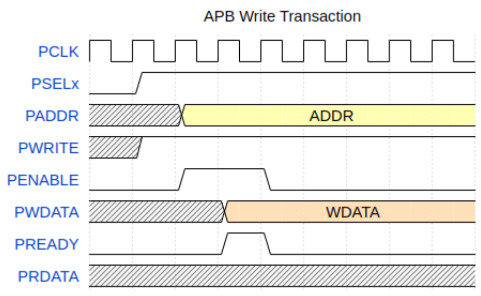


Figure 3.1: APB Write Transaction

Read Transaction

In a read operation, the master requests data from a peripheral, and the peripheral responds over the `PRDATA` bus.

Setup Phase:

- The master activates `PSELx` to select the desired peripheral.
- The read address is driven on the `PADDR` line.
- The `PWRITE` signal is set low to indicate a read operation.

Access Phase:

- The master asserts `PENABLE` to begin the read cycle.
- The slave places the requested data onto the `PRDATA` bus.
- The slave sets `PREADY` high to signal that the data is valid.
- The master deasserts `PENABLE`, finalizing the transaction.

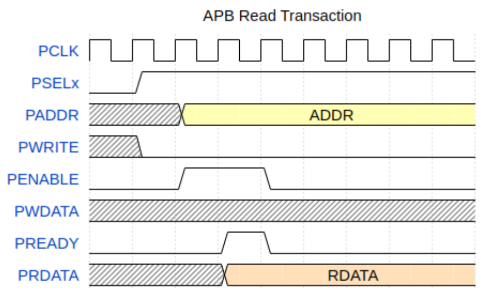


Figure 3.2: APB Read Transaction

3.2 Top Level Diagram of APB Protocol

The APB system architecture implemented a typical master-slave communication where the master initiates the transaction based on `PSEL` and `PENABLE` assertions.

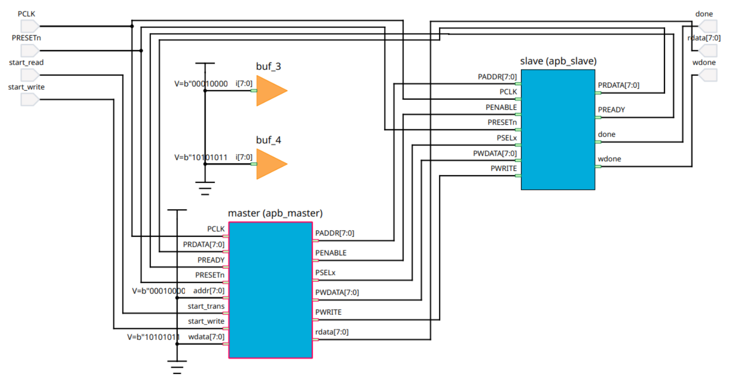


Figure 3.3: APB Top Level Diagram

3.3 DataFlow & Hardware Blocks

The APB system was divided into:

- **APB Master Module**: The apb-master module is designed to initiate APB transactions—either read or write—to communicate with peripheral devices using the AMBA APB protocol. It operates in three states (IDLE, SETUP, ACCESS) and is driven by a simple finite state machine (FSM). This master initiates the transaction only when start-trans is low, and it chooses between read and write operations based on the start-write signal.
- **APB Slave Module**: The slave responds to valid APB transfer phases—when PSELx and PENABLE are asserted. It uses the PWRITE signal to differentiate between write and read operations. The module outputs transaction status flags (PREADY, wdone, done) and read data (PRDATA) accordingly.
- **Top Module**: Instantiates slave and Master Module .

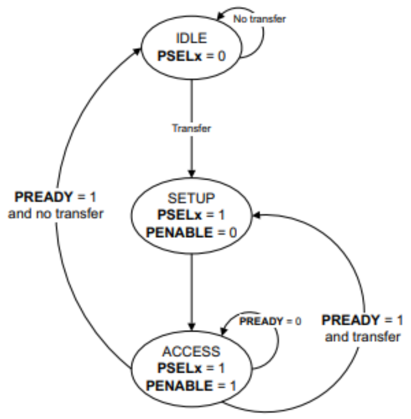


Figure 3.4: APB State Diagram.[4]

3.4 Results & Logic Usage

Simulations verified transaction sequences using GtkWave and Iverilog. Hardware testing on Vaaman FPGA confirmed correct operation.

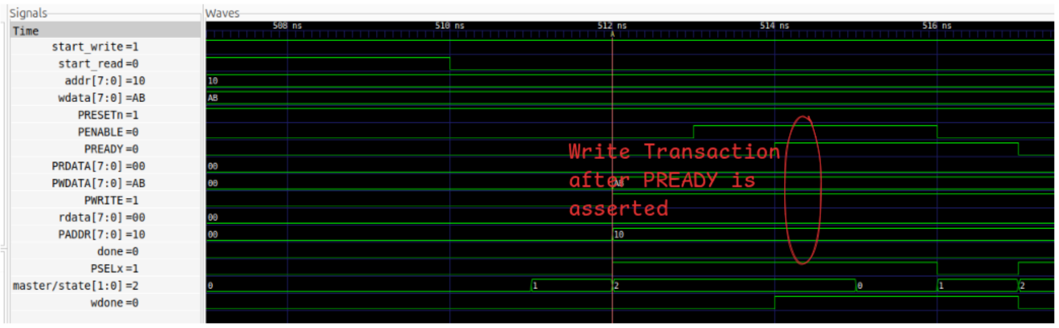
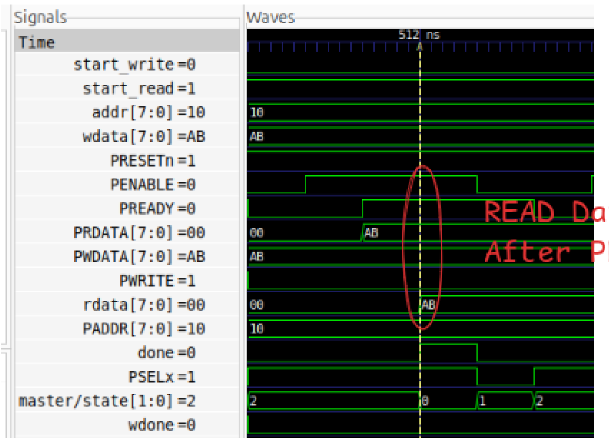
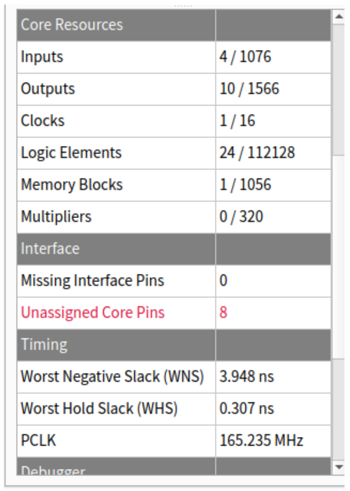


Figure 3.5: APB Write Transaction



(a) APB READ Transaction.



Core Resources	
Inputs	4 / 1076
Outputs	10 / 1566
Clocks	1 / 16
Logic Elements	24 / 112128
Memory Blocks	1 / 1056
Multipliers	0 / 320
Interface	
Missing Interface Pins	0
Unassigned Core Pins	8
Timing	
Worst Negative Slack (WNS)	3.948 ns
Worst Hold Slack (WHS)	0.307 ns
PCLK	165.235 MHz
Debugger	

(b) Logic - Timing Reports

Figure 3.6: APB READ Transaction & Logic Usage

Chapter 4

USB 1.0 Communication Protocols

4.1 USB 1.0 Protocol Description

Major tech firms like Compaq, Intel, Microsoft, and NEC worked together to design the Universal Serial Bus (USB) specification; Hewlett-Packard, Lucent, and Philips eventually participated. These businesses established the USB Implementers Forum, Inc. (USB-IF), a nonprofit group in charge of regulating the development of USB and publishing its specifications.

Addressing the intricacy and constraints of the current peripheral connections to PCs was the main objective of USB. Before USB, PCs used a variety of ports, each with its own connector type and communication protocol, including serial, parallel, keyboard, mouse, joystick, and MIDI interfaces. These outdated ports frequently needed to be manually configured and used up important system resources like DMA channels and hardware interrupts.

Main Idea Behind USB and its Need:-

- Use a single interface to standardize peripheral connections.
- Remove the requirement for particular DMA or interrupt resources.
- Allow devices to be attached or disconnected without restarting the system by supporting hot-plugging.
- Plug-and-play features make it easier to install devices by allowing for automatic driver installation and identification.

4.1.1 USB Versions & their Data Speeds

- **Low Speed:** 1.5 Mbit/s, intended for devices like mice that require minimal bandwidth. Low-speed cables are thinner and more flexible.
- **Full Speed:** 12 Mbit/s, originally specified for a wide range of devices beyond low-speed peripherals.

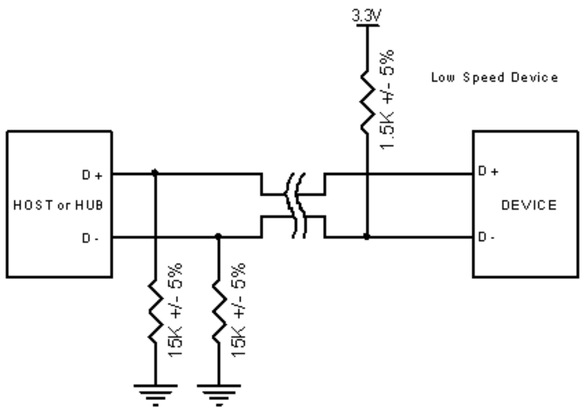


Figure 4.1: USB Low speed Identification[5]

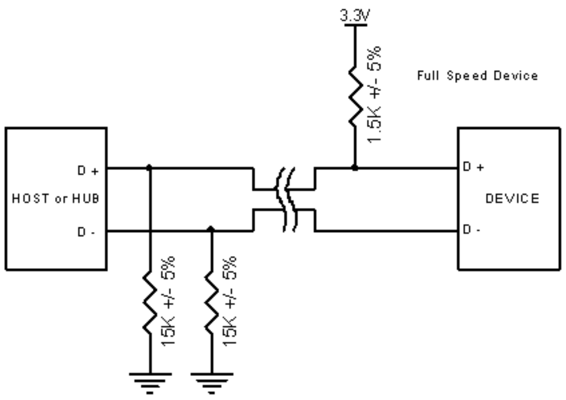


Figure 4.2: USB Full speed Identification[5]

- **High Speed:** 480 Mbit/s, introduced with USB 2.0 to accommodate devices requiring higher data throughput, such as external storage devices and video equipment.

  As you can see, we have left the High Speed mode's speed designation out. In the beginning, high-speed devices will connect as full-speed devices (1.5k to 3.3V). After it is attached, if the hub allows it, it will make a high-speed chirp during reset and create a high-speed connection. The pull-up resistor is eliminated to balance the line while the gadget is in high speed mode.

It's important to note that these speeds represent the maximum signaling rates; actual data throughput is lower due to protocol overhead.

4.1.2 Usb Line States

Bus State	**D+ Level**	**D- Level**	**Description**
Differential '1'	High	Low	Logical 1 (J state at Full-speed, K state at Low-speed)
Differential '0'	Low	High	Logical 0 (K state at Full-speed, J state at Low-speed)
SE0	Low	Low	Single-Ended Zero – indicates end of packet or reset
SE1	High	High	Illegal state – not used in normal USB operation
Data J State			
- Low-speed	Low	High	Represents differential '0'
	High	Low	Represents differential '1'
Data K State			
- Full-speed	High	Low	Represents differential '1'
	Low	High	Represents differential '0'
Idle State			
- Low-speed	High	Low	Line remains in J state (differential '1')
	High	Low	Same as low-speed; D+ high indicates full-speed idle
Resume	—		Transition to K state from idle
SOP (Start of Packet)	—		Line switches from idle (J) to K state
EOP (End of Packet)	Low	Low	SE0 for 2 bit times followed by J state for 1 bit time
Disconnect	Low	Low	SE0 held for greater than equal to 2 µs (line pulled low)
Connect	Idle	Idle	Idle line state held for 2.5 µs
Reset	Low	Low	SE0 held continuously for greater than equal to 2.5 µs

Table 4.1: USB Bus State Table

4.1.3 USB Packet Types

USB communication is structured around different types of packets, each fulfilling a distinct role in ensuring accurate and efficient data transfer.

1. Token Packets

These packets initiate communication by addressing the destination device and indicating the direction of data flow:

- **OUT:** Used when the host sends data to a USB device.
- **IN:** Used when the host requests data from a device.
- **SETUP:** Initiates control transfers, typically for device configuration.

2. Data Packets

Data packets are responsible for transferring the actual payload:

- **DATA0 and DATA1:** Used alternately to support error checking.
- **DATA2 and MDATA:** Employed in high-speed isochronous transfers for more complex data flows.

Alternating sequences like DATA0 and DATA1 help identify packet loss and ensure data integrity.

3. Handshake Packets

These packets convey the outcome of a data transaction:

- **ACK:** Confirms successful data reception.
- **NAK:** Indicates the device is temporarily unable to handle the request.
- **STALL:** Signals an error condition that prevents further processing.
- **NYET:** Used in high-speed mode to denote that the device is not yet ready.

4. Special Packets

Specialized packets are used for synchronization and legacy device compatibility:

- **SOF (Start of Frame):** Synchronizes data flow by marking the start of each frame.
- **PRE (Preamble):** Alerts the bus that a low-speed device will communicate.

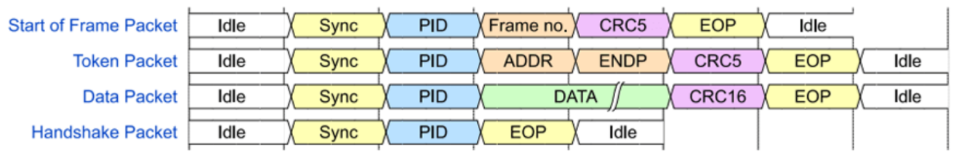


Figure 4.3: USB Packet Structure

4.1.4 Packet Structure

Each USB packet follows a defined structure consisting of the following components:

- **Sync:** A predefined pattern that aligns the receiver with the incoming data stream.
- **PID (Packet Identifier):** Specifies the type and purpose of the packet.
- **Data:** Contains the payload when applicable.
- **CRC (Cyclic Redundancy Check):** Used to detect transmission errors.
- **EOP (End of Packet):** Indicates the conclusion of the packet transmission.

4.2 Top Level Diagram of USB 1.0 Protocol.

The design architecture included:

- **USB PHY Layer**: The electrical signaling between devices and the host is managed by the USB PHY Layer. In order to ensure correct signal transmission via USB lines, it controls bit stuffing, clock recovery, speed detection, and data encoding/decoding (such as NRZI).

- **Protocol Layer**: This layer controls CRC-based error checking, packet framing, and decoding. In order to guarantee protocol compliance and dependable communication between the host and devices, it arranges data into token, data, and handshake packets.

- **Host Controller Layer**:The Host Controller manages data transfer, token issuance, handshake answers, and the scheduling and control of USB transactions. Additionally, it controls power distribution to linked peripherals, bandwidth distribution, and device enumeration.

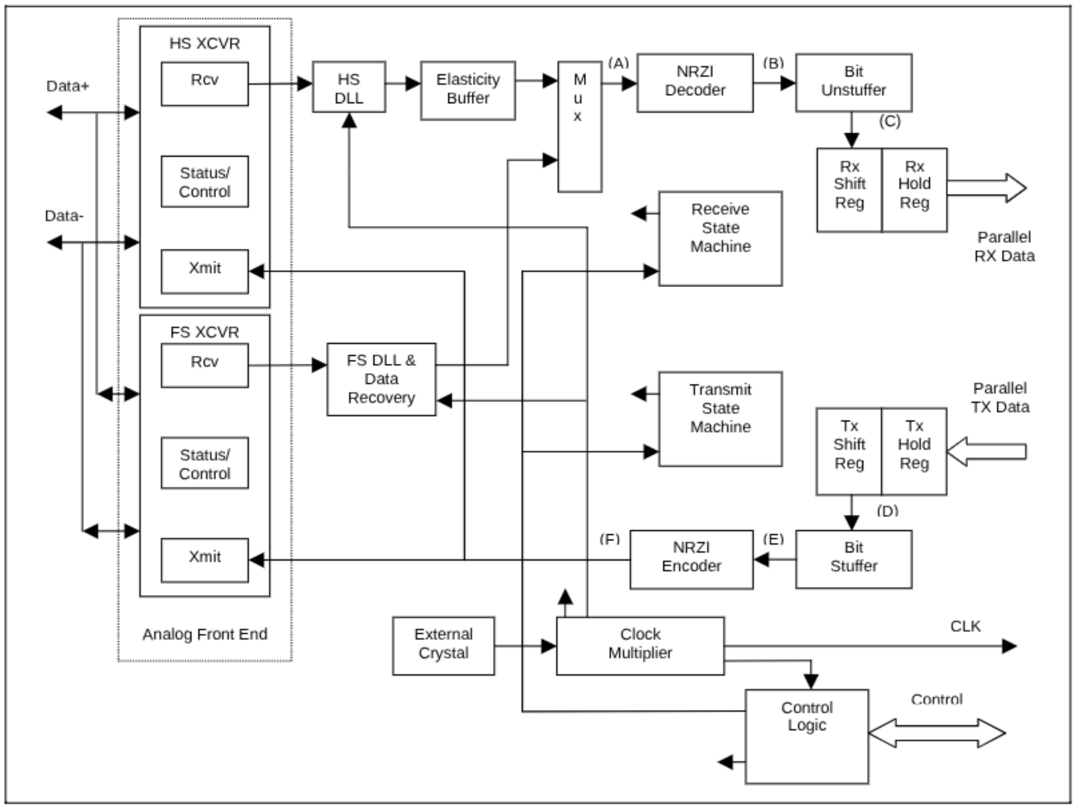


Figure 4.4: Universal Transceiver Macrocell Interface (UTMI Architecture)[6]

4.3 Hardware Design Blocks and DataFlow

The USB system comprised the following modules:

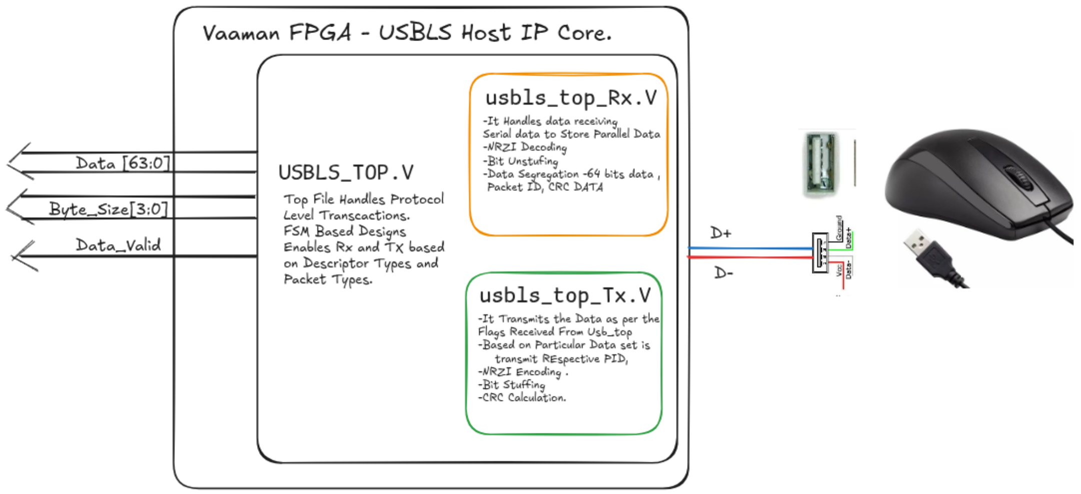

Vaaman FPGA - USBLS Host IP Core.
Data [63:0]
Byte_Size[3:0]
Data_Valid
USBLS_TOP.V
Top File Handles Protocol Level Transcactions. FSM Based Designs Enables Rx and TX based on Descriptor Types and Packet Types.
usbls_top_Rx.V
-It Handles data receiving Serial data to Store Parallel Data
-NRZI Decoding
-Bit Unstufing
-Data Segregation -64 bits data , Packet ID, CRC DATA
usbls_top_Tx.V
-It Transmits the Data as per the Flags Received From Usb_top
-Based on Particular Data set is transmit REspective PID,
-NRZI Encoding .
-Bit Stuffing
-CRC Calculation.
D+
D-
Ground
Data+
Vcc
Data-

Figure 4.5: USB 1.0 Block Level Diagram

- **USB RX Module**: It Handles data receiving Serial data to Store Parallel Data ,NRZI Decoding, Bit Unstufing, Data Segregation, 64 bits data , Packet ID, CRC DATA.
- **USB TX Module**: It Transmits the Data as per the Flags Received From Usb-top , Based on Particular Data set is transmit Respective PID, NRZI Encoding , Bit Stuffing, CRC Calculation. It contains many sub modules for Each function.
- **CRC Modules**: Real-time USB CRC5 and USB CRC16 computation.
- **Top Controller**: Top File Handles Protocol Level Transcactions. Enables Rx and TX based on Descriptor Types and Packet Types.Handles ACK and NAK logic when CRC are Correct.

4.4 Results & Simulations

Hardware tests on Vaaman FPGA demonstrated flawless USB HID detection and multiple stages of usb transactions includes Various Successful Descriptors Communcation then we can transition into Data Stage where HID Devices and Continuosly send the ASCII Values(Keyboards) or X,Y Movement(Mouse).

Such our Designed USB 1.0 Host Core was able to Handle Various Such Descriptors & Data Stage. It also has ACK/NAK Mechanism when CRC not matching.

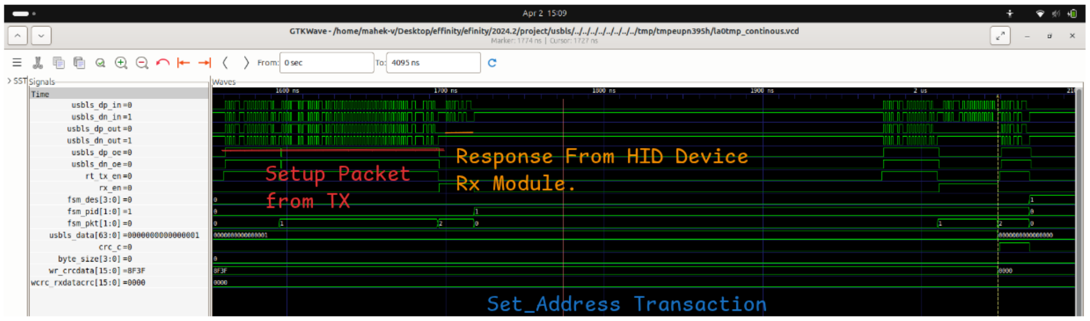
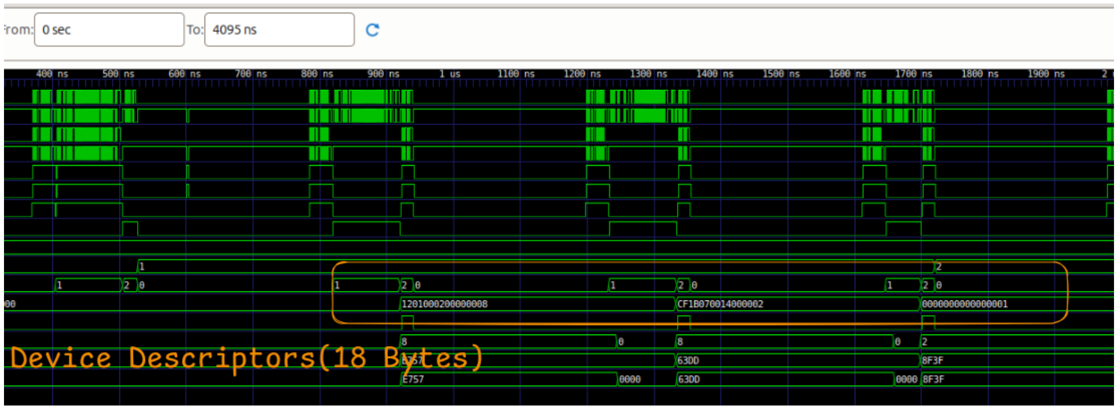
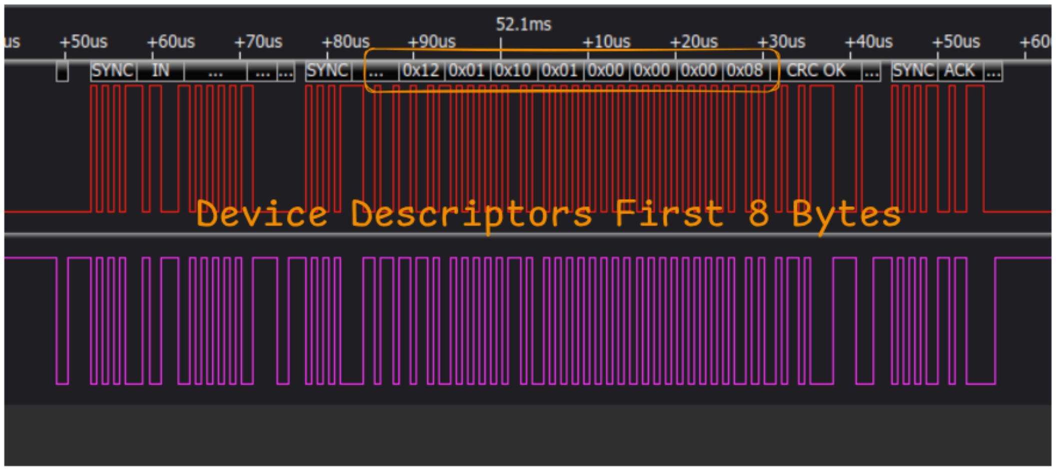


Figure 4.6: Set Address Snippet for USB 1.0 Transaction



(a) Tested on Vaaman Hardware DEvice Descriptor



(b) Capture of Device Descriptor Data when HID Communicating With Laptop

Figure 4.7: Device Descriptors Comparison with Design Core and Capture of Real time HID Communication with PC

4.5 Integration of USB 1.0 Host Core with Periplex.

Integration of USB 1.0 Host Core with Existing Periplex Design (On Demand Hardware Peripheral Generator) involves Architecting Bridge Which Takes 64 Bits Inputs to Periplex Propretiary Frame Size and goes through RAH(Proprietary Real time Application Handler) Towards CPU . Below Involves The Block level DIagram for it and DATA Receiving from CPU Snippet.

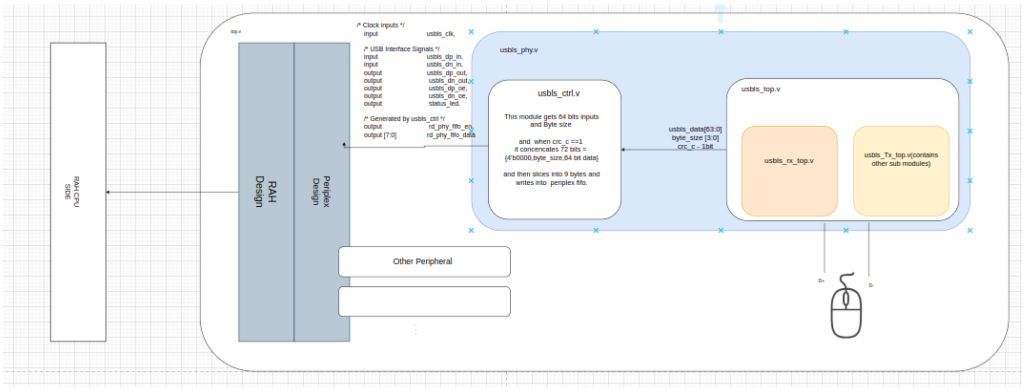
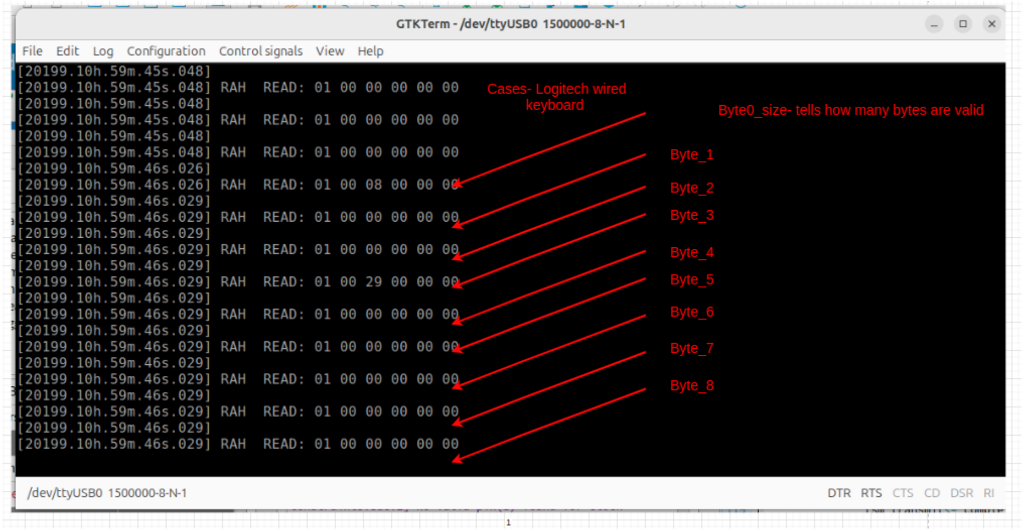


Figure 4.8: Block Level Diagram For USB Integration



GTKTerm - /dev/ttyUSB0 1500000-8-N-1
File Edit Log Configuration Control signals View Help
Cases- Logitech wired keyboard
Byte0_size- tells how many bytes are valid
Byte_1
Byte_2
Byte_3
Byte_4
Byte_5
Byte_6
Byte_7
Byte_8
/dev/ttyUSB0 1500000-8-N-1
DTR RTS CTS CD DSR RI

```
[20199.10h.59m.45s.048]
[20199.10h.59m.45s.048] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.45s.048]
[20199.10h.59m.45s.048] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.45s.048]
[20199.10h.59m.45s.048] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.026]
[20199.10h.59m.46s.026] RAH  READ: 01 00 08 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 29 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
[20199.10h.59m.46s.029]
[20199.10h.59m.46s.029] RAH  READ: 01 00 00 00 00 00
```

Figure 4.9: CPU Side Data Received Successfully

Chapter 5

Conclusion

The project effectively illustrates the creation of USB 1.0 host cores, APB, and UART that are designed for FPGA deployment. Every protocol was created from the ground up and validated using the Vaaman board's hardware and simulation. The USB Host Core is the most intricate and significant of them all, demonstrating the capacity to handle real-time HID data, descriptor interpretation, and device enumeration. Modular testing was made possible by the layered architecture, which also guaranteed protocol compliance across the token, data, and handshake stages of USB. The system's ability to scale and adapt to real-world embedded contexts was validated through integration with the Periplex framework. All things considered, the initiative prepares the key IPs for wider deployment by bridging the gap between academic knowledge and industrial protocol development.

5.1 Future Work

Upgrading the USB Host Core to accommodate USB 1.1 and USB 2.0 protocols is the next stage of development. By enhancing signal integrity and error management while preserving backward compatibility with USB 1.0, USB 1.1 expands on the current architecture. More importantly, USB 2.0 adds the ability to transmit data at high speeds (480 Mbps), necessitating improvements in buffer management, protocol layer speed negotiation, and PHY encoding. Design will need to use dual-packet management techniques for split transactions, chirp sequence detection, and efficient NRZI decoding in order to support USB 2.0. Further application domains will become available if Mass Storage and Audio Class devices beyond HID are supported. The IP will become commercially viable after the existing design is modified for the UTMI+ interface and USB-IF certification standards are met.

Bibliography

[1] Vicharak Computers LLP, “Vicharak Documentation,” 2024. Accessed: Apr. 29, 2025.

[2] A. Devices, “Figure: UART Timing Diagram – UART: A Hardware Communication Protocol,” 2021. Accessed: Apr. 29, 2025.

[3] R. Merrick, “Project 7: UART – Part 1 – Receive Data from a Computer,” 2014. Accessed: Apr. 29, 2025.

[4] Arm Ltd., “AMBA® APB Protocol Specification (IHI 0024D),” 2021. Accessed: Apr. 29, 2025.

[5] C. Peacock, “USB in a Nutshell – Chapter 1: Introduction,” 2002. Accessed: Apr. 29, 2025.

[6] I. Corporation, “Usb 2.0 transceiver macrocell interface (utmi) specification,” Tech. Rep. Version 1.05, Intel Corporation, 2001. Accessed: Apr. 30, 2025.

[7] C. Lippincott, “UART: A Hardware Communication Protocol,” 2021. Accessed: Apr. 29, 2025.

[8] Life is a SoC, “A Comprehensive Guide to the APB Protocol: Design and Implementation,” 2023. Accessed: Apr. 29, 2025.