AXI-4 Stream UART IP Core

FPGA IP Core Design with VHDL

by
Ronaldo Tsela

Revision History Table

Date	Version	Revision
13/8/2024	1.0	Initial release of this document.

Overview

This document presents the design and implementation of an AXI-4 Stream Universal Asynchronous Receiver Transmitter (UART) IP core in VHDL. This IP core offers an efficient solution for interfacing serial peripherals with AXI-4 Stream-compatible hardware accelerator units.

The design, simulation, implementation, and testing of this IP core were conducted using the AMD-Xilinx Vivado Suite (version 2022.2) and Vitis SDK (version 2022.2). The platform used for functional testing consists of the Zynq-7000 AP SoC with an Artix-7 FPGA (speed grade -1) integrated into the ZedBoard Evaluation and Development Kit. Additionally, the UART-to-USB bridge from Digilent [4] was utilized to validate the functionality of this IP core.

Key features of this IP core include:

- Efficient conversion between AXI-4 Stream and serial interface, supporting both directions.
- Full-duplex and simplex communication modes with fully decoupled transmitter and receiver operations.
- Configurable FIFO buffer width and depth for both receiver and transmitter.
- Configurable data bits per character for both receiver and transmitter.
- Adjustable baud rates for receiver and transmitter.
- Configurable clock frequencies for receiver and transmitter operations.
- Compact design with low hardware resource utilization (339 LUTs and 485 FFs) and zero control complexity.
- Fully custom component design in VHDL.

Design limitations:

- No parity check implemented.
- Tested only on Zynq-7000 SoC with Artix-7 FPGA (speed grade -1).
- No software drivers implemented.
- No benchmarks or performance measurements conducted.

This document is structured as follows:

- Chapter 1: Overview of the UART protocol.
- Chapter 2: Description of the Advanced Microcontroller Bus Architecture (AMBA®) AXI-4 Stream protocol.
- Chapter 3: Specifications of the developed AXI-4 Stream UART IP core.
- Chapter 4: Detailed description of the IP core's design and implementation in VHDL.
- Chapter 5: Evaluation and testing results demonstrating the functionality of the IP core.

Usage

All components necessary for synthesizing and using this IP core are contained in this repository. Simply download the VHDL source code and integrate it directly into your design or add it to your IP core repository for use in your projects. There are no external dependencies (to the best of my knowledge). **Table 1** lists the generic parameters used to configure the IP core along with their descriptions, while **Fig. 1** illustrates the IP core as displayed in the Block Design tool of Vivado.

Table 1 - The AXI-4 Stream UART IP core list of generics.

Generic Name	Description	Default Value
C_RX_BAUDRATE	Receiver baud rate	115,200
C_RX_CLOCK_FREQ	Receiver port clock frequency	100 MHz
C_RX_DATA_BITS	Receiver number of payload data bits	8
C_RX_FIFO_DEPTH	Receiver FIFO buffer depth	16
C_TX_BAUDRATE	Transmitter baud rate	115,200
C_TX_CLOCK_FREQ	Transmitter port clock frequency	100 MHz
C_TX_DATA_BITS	Transmitter number of payload data bits	8
C_TX_FIFO_DEPTH	Transmitter FIFO buffer depth	16

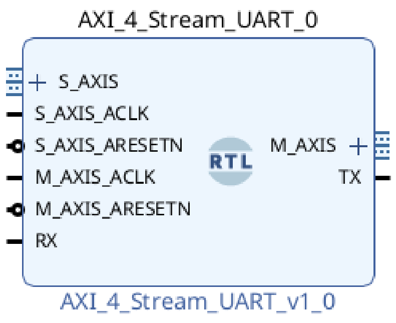

Fig. 1 - The AXI-4 Stream UART block diagram in Vivado.

Chapter 1 : UART Protocol

The Universal Asynchronous Receiver Transmitter (UART) protocol is one of the earliest and simplest communication protocols, originally developed by Gordon Bell at Digital Equipment Corporation in the 1960s. Despite its age, UART remains a widely used communication standard in various electronic applications, ranging from simple consumer electronics that communicate between peripherals on a single board to more complex device-to-device communication architectures [1].

The UART protocol defines the method for structuring and transmitting data packets (serial frames) between two electronic devices through a serial channel, typically consisting of two wires. The physical implementation of UART can be achieved using various electrical signaling standards, such as RS-232, RS-485, or raw TTL [2].

Traditionally, UART is implemented as an individual integrated circuit specialized for enabling and controlling serial communication between electronic devices. However, as serial communication has become a common feature in most electronics, UART controllers are often integrated into the same silicon die as the device's digital logic [2].

A typical UART system consists of five primary components, as depicted in **Fig. 2** and listed bellow:

1. Receiver first-in-first-out buffer (RX FIFO) : A memory buffer for storing the received data bits.
2. Transmitter first-in-first-out buffer (TX FIFO) : A memory buffer for storing the data to be transmitted.
3. Receiver unit (RX) : Implements the logic for receiving serial data and converting them into parallel packets.
4. Transmitter unit (TX) : Implements the logic for converting the parallel data packets into a serial form for transmission.
5. Baud rate generator (BDG) : Generates the synchronization signal for data transmission and reception based on a predefined baud rate value shared among the TX and RX of a communication channel.

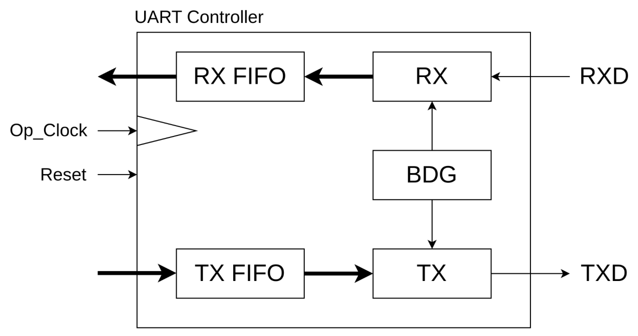

Fig. 2 - A high-level structural block diagram of a general UART controller

The TX unit reads data stored in the TX FIFO and transmits it serially, bit-by-bit. Conversely, the RX unit receives serial data, bit-by-bit, and stores it as a packet within the RX FIFO. As an asynchronous communication protocol, UART does not use any transmitted clock signal to synchronize the transmitter and receiver. Instead, synchronization is achieved through the use of the BDG unit, which generates local synchronization signals based on a predefined baud rate. The baud rate must be identical for both the TX and RX units of a communication channel, ensuring that data is transmitted and received at the same rate, independent of the devices' operational clock frequencies. Each bit is transmitted in the channel with frequency equal to F_{OP} / BD [Hz], where F_{OP} is the operational frequency of the receiver or transmitter and BD the pre-defined baud rate values. Therefore each bit in the data frame lasts for BD / F_{OP} [sec].

A UART data frame consists of 8 to 13 bits and it is structured as follows (see **Fig. 3**) [2]:

- Start bit (1 bit) : Indicates the beginning of a transmission, represented by a logical low value ('0').
- Data bits (5 to 9 bits) : The actual payload data of the transmission ranging from 5 to 9 bits.
- Parity bit (0 or 1 bit) : Optional error-checking bit used in odd or even mode. The parity bit is assigned a logical high value if the total number of logical high values in the payload including the parity bit is odd or even depending on the selected parity checking mode.
- Stop bit (1 or 2 bits) : Indicates the end of the transmission, represented by a logical high value ('1').

When no data transmission is occurring, the communication channel is held in an idle state represented by a logic high value ('1'). Once a transmission begins, a start bit is sent, followed by the data bits, an optional parity bit, and one or two stop bits forming the serial data frame. The encapsulation of data within start and stop bits, along with the consistent duration of each bit as aforementioned, enables the receiver and transmitter to remain synchronized and therefore to communicate successfully [2].

Start Bit (1)	Data Bits (5-9)						Parity Bit (0-1)	Stop Bit(1-2)
						…		

Fig. 3 - The structure of a UART data frame.

The transmission operation is relatively straightforward as depicted in the state diagram shown in **Fig. 4**. When the TX unit has a data packet available, it initiates transmission by sending a start bit (logic low) to the channel. It then transmits the data bits serially, starting from the least significant bit (LSB) to the most significant bit (MSB). After transmitting the data bits, the TX unit sends one or two stop bits to complete the transmission. The timing of each transmitted signal is determined by the operation frequency of the TX unit (F_{OP}) and the baud rate (BD). The TX unit holds each signal for a duration of BD / F_{OP} [sec].

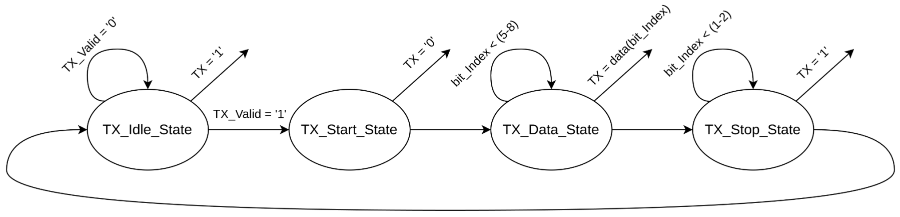

Fig. 4 - State diagram of UART transmitter operation.

The reception operation is depicted in the state diagram in **Fig. 5**. The reception operation is slightly more complex than the transmission operation. The complexity arises from how the RX unit captures the transmitted bits. The RX unit typically employs oversampling to accurately capture the transmitted bits, with the most common method being 16x oversampling. This method divides the bit time into 16 equal intervals and samples each bit at multiple positions. Another simpler method involves sampling the transmitted bit at the middle of its duration. Therefore the RX unit relies on the BDG unit to generate the necessary sampling ticks with frequency F_{OP} / (BD x 16) [Hz].

The RX unit remains idle until it detects a valid start bit. Upon detection, it transitions to the data reading state, where it sequentially captures each data bit transmitted on the channel, forming the complete data packet. Once all the data bits have been read, if parity checking is enabled, the RX unit reads the parity bit. If an error is detected, the data packet is discarded. Otherwise, the RX unit proceeds to the stop bit reception state, where it waits for one or two valid stop bits to complete the transmission.

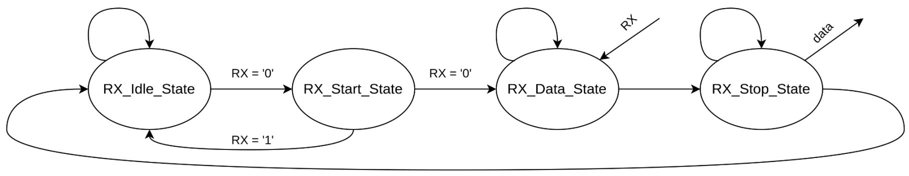

Fig. 5 - State diagram of UART receiver operation.

Chapter 2 : AMBA ® AXI-4 Stream Protocol

The Advanced eXtensible Interface (AXI) Stream protocol is a part of the Advanced Microcontroller Bus Architecture (AMBA®) family of interconnect protocols developed by ARM. It is specifically designed to facilitate high-speed data streaming between master and slave components in a system-on-chip (SoC) environment, making it ideal for applications requiring continuous data flow, such as video and image processing, and communication systems. The AXI-4 Stream protocol is widely used in hardware accelerator architectures designed for high-performance digital signal processing operations [5].

Unlike the traditional AXI protocols, which focus on memory-mapped transactions, the AXI-4 Stream protocol is dedicated to data streaming. It allows the transfer of data without the need for addressing, making it simpler and more efficient for certain types of data transmission. This protocol operates on a handshake mechanism to ensure reliable data transfer, supporting configurable data widths for flexible system designs [5].

The AXI-4 Stream protocol defines a set of signals that are used to control the flow of data between a master and a slave [5]. The key signals employed in a are illustrated in **Fig. 6** and are described as follows:

- AXIS_TVALID: The AXIS_TVALID signal is driven by the master to indicate that valid data is available on the data bus. The master sets this signal high when it has data ready for transmission.
- AXIS_TREADY: The AXIS_TREADY signal is driven by the slave to indicate that it is ready to receive data. When this signal is high, the slave is prepared to accept data from the master.
- AXIS_TDATA: The AXIS_TDATA signal carries the actual data being transmitted from the master to the slave. The width of the AXIS_TDATA bus can be configured based on the specific application requirements, allowing for scalable data transfer sizes.
- AXIS_TKEEP: The AXIS_TKEEP signal is used to indicate which bytes of the AXIS_TDATA bus are valid. This is particularly useful when the data width is greater than the actual number of bytes being transmitted, allowing for efficient packing of data.
- AXIS_TLAST: The AXIS_TLAST signal marks the end of a data stream or packet. This signal is asserted by the master when the last piece of data in a transaction is being transmitted, signaling to the slave that the current stream or packet is complete.
- AXIS_TSTRB: The AXIS_TSTRB signal operates similarly to AXIS_TKEEP but indicates which bytes of the AXIS_TDATA bus should be written to memory. It is used to control byte-level write operations, ensuring that only the intended data is written to the destination.

In most applications where the AXI-4 Stream protocol is used, the signals described above are sufficient to design a complete interface. However, additional signals are available that provide enhanced capabilities and greater flexibility for DSP designs. In the case of the AXI-4 Stream UART IP core presented in this document, a simplified version of the AXI-4 Stream protocol is employed instead, utilizing only the essential signals: AXIS_TVALID, AXIS_TREADY, and AXIS_TDATA.

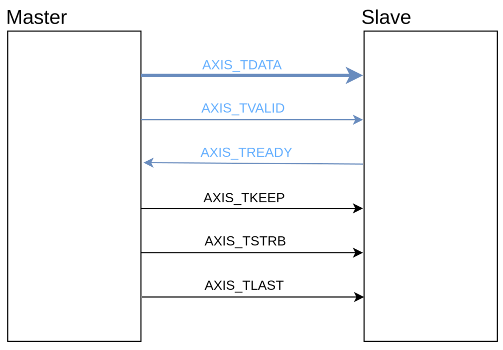

Fig. 6 - The AXI-4 Stream signals. In blue the essential signals are depicted

The AXI-4 Stream protocol relies on a simple yet robust handshake mechanism to control the flow of data between the master and slave. Data transfer only occurs when both the AXIS_TVALID signal from the master and the AXIS_TREADY signal from the slave are asserted high [5]. This ensures that data is transmitted only when the slave is ready to receive it, preventing data loss and ensuring reliable communication. The handshake mechanism also allows for flow control, where the slave can deassert the AXIS_TREADY signal to pause data transmission if it is not ready to process more data.

Chapter 3 : IP Core Specifications

The primary objective of designing this IP core is to facilitate the interface between serial peripherals and hardware accelerators using a simplified variant of the AXI-4 Stream protocol. Consequently, the design must effectively implement both the UART interface and the simplified AXI-4 Stream variant described in the previous chapters. **Fig. 7** provides a high-level overview of the IP core's internal structure. On the left side, the AXI Stream buses are depicted (both master and slave), while on the right side, the serial channels for RX (Receiver) and TX (Transmitter) are shown. Conceptually, this IP core can be viewed as a combination of two distinct units: one that receives serial data from the RX channel and forwards the parallel data packets to a slave AXI-4 Stream peripheral, and another that receives data packets from an AXI-4 Stream master peripheral, serializes them, and transmits them through the TX channel.

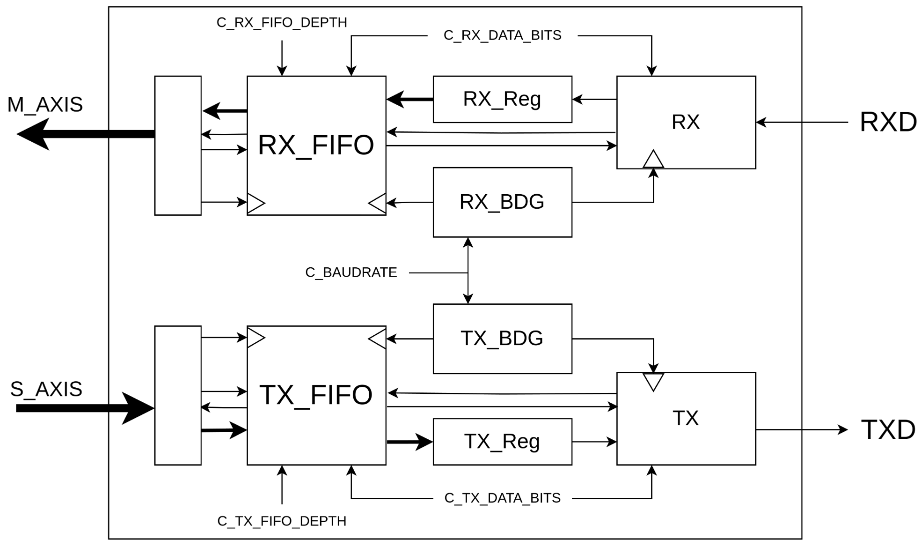

Fig. 7 - The AXI-4 Stream UART controller architecture.

The architecture of the IP core is intentionally kept simple to minimize hardware resource utilization while maintaining efficient functionality. There is no dedicated block designed to provide exclusive control over the IP core's operations. Instead, control is inherently managed by the protocols themselves, specifically through the TX and RX FSMs and the AXI-4 Stream - compatible FIFOs. This design choice eliminates the need for additional control logic, thereby reducing complexity and ensuring low resource usage.

One of the key features of this IP core is its high degree of configurability, making it adaptable to a wide range of applications of hardware accelerator designs. The IP core supports several configuration parameters, including:

- FIFO Depth: The depth of both the RX and TX FIFO buffers can be adjusted to accommodate different data buffering requirements.
- Data Packet Width: The number of bits per character in the data packets can be configured, allowing the core to handle various data formats. This also configures the AXI-4 Stream bus data-width.
- Baud Rate: The baud rate for both the receiver and transmitter can be specified to match the desired serial communication speed.
- Operation Clock Frequency: The operating clock frequency for both the receiver and transmitter can be independently set, enabling the RX and TX ends to operate with different timing.

The ability to configure the receiver and transmitter ends independently allows the IP core to operate in both full-duplex and simplex communication mode as well, where the RX and TX channels function separately. This flexibility is particularly useful in applications where different communication settings are required for transmitting and receiving data.

It is important to note that the configuration of the IP core is based on generics. The use of generics provides a powerful mechanism for tailoring the IP core to specific application needs, but it also requires that the desired configuration be determined and applied before the hardware is synthesized.

Chapter 4 : IP Core Design and Implementation

The AXI-4 Stream UART IP core is designed entirely from scratch using VHDL. Each functional unit within the core is mapped to a corresponding digital component. Based on the previous chapters, we can identify these components as follows: the baud rate generator is implemented as a simple frequency divider unit, the FIFO buffers are dual-clock, AXI-4 Stream-compatible FIFO memories (also designed from scratch), and the UART receiver (RX) and UART transmitter (TX) units are simple finite state machine controllers. The TX_Reg and RX_Reg are simple registers that function as shift registers by modifying the bit addressing index on each iteration controlled by the baud rate generator signaling.

The design process begins with the TX and RX FSM control units. Both FSMs are modeled as Moore FSMs with four states, as shown in **Fig. 4** and **Fig. 5** respectively. Both FSMs are clocked using the frequency divider (BDG). Specifically by dividing the clock frequency by BD x 16, the synchronization clock for the TX and RX Control FSM is obtained. The RX is designed to sample each received bit in the middle instead of performing any other oversampling technique.

The top-level entity of the TX FSM control includes the signals: Baud_Tick, Reset, TX, TX_Ready, TX_Valid, and TX_Data and the RX FSM control entity includes respectively the signals: Baud_Tick, Reset, RX, RX_Valid, RX_Ready, and RX_Data. The shift registers (RX_Reg and TX_Reg) are implemented internally as simple registers clocked at F_{OP} / (BD x 16) [Hz] with variable indexing that changes every bit-interval. This indexing allows the RX FSM controller to iterate over the data reception state and the TX FSM controller to manage data transmission, thereby achieving the shifting operation as predefined.

The design of the UART FSM controllers is the core of this IP. Once these components are validated and their functionality is proven through simulation-based testing, the next step is to integrate them with the remaining components: the two FIFO buffers and the two frequency dividers.

The FIFO buffers are designed to be compatible with the simplified AXI-4 Stream interface, featuring two ports: one for writing and one for reading as we expect. In addition these FIFOs support dual-clock functionality to accommodate the different operating frequencies of the serial interface FSM controllers (F_{OP} / (BD x 16) [Hz]) and the AXI Stream interface (F_{OP} [Hz]). Without dual-clock support, bridging the AXI-4 Stream interface with the serial interface would lead to synchronization errors during read and write operations.

These FIFOs include internal control signals (FIFO_Full and FIFO_Empty), which are not exposed to the external interface. Instead, the AXI-4 Stream interface manages the transaction control effectively through the inherent handshaking mechanism. Specifically when we say that the FIFOs are AXI-4 Stream compatible we mean that they can only be written to if both S_AXIS_TVALID and S_AXIS_TREADY are high, and it can only be read from if both M_AXIS_TVALID and M_AXIS_TREADY are high respectively. The S_AXIS_TREADY signal is asserted high only when the FIFO is not full, and deasserted otherwise. Similarly, M_AXIS_TVALID is high only when the FIFO is not empty. This design ensures that the entire system is controlled through these dual-clock AXI-4 Stream FIFO buffers and the AXI-4 interface instead of relying on additional control unit.

All components are ultimately wrapped into a top-level AXI-4 Stream-compatible module that includes four parallel channels: AXI-4 Stream master, AXI-4 Stream slave, serial receiver, and serial transmitter. The elaborated design schematic is depicted in **Fig. 8**.

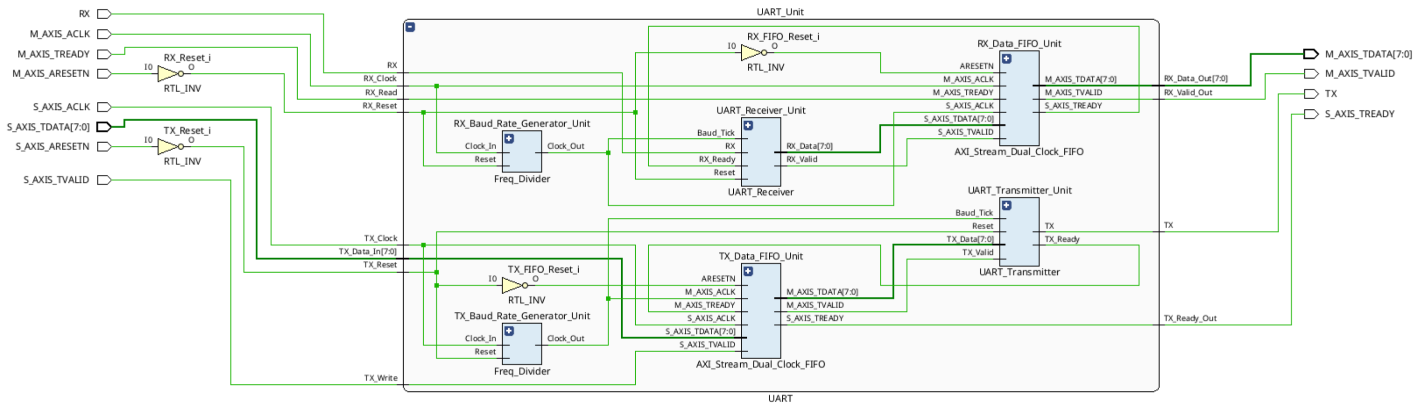

Fig. 8 - The elaborated design schematic diagram of the AXI-4 Stream UART IP core.

Chapter 5 : IP Core Validation and Testing

The validation and testing process of the AXI-4 UART IP core is divided into two main stages: simulation-based validation and hardware testing on the ZedBoard Evaluation Board platform. Initially testbench VHDL scripts are designed and employed with the Vivado simulation tools to verify the correct operation of the IP core and its components under various conditions. In addition a top level testbench which is provided along with the VHDL model of this IP core is used to simulate both the UART and AXI-4 Stream interfaces evaluating their behavior. The simulation tests assess the IP core's ability to accurately transmit and receive data between the UART and AXI-4 Stream interfaces in various configurations, such as different baud rates, data bit lengths, and FIFO depths. Additionally, edge cases are examined, including the handling of FIFO overflow or underflow conditions, as well as the core's behavior during intermediate resets.

A timing analysis is also conducted to examine the critical paths and determine the core's range of operational frequencies. After performing this timing analysis (post-implementation), the critical path is found to be 6.659 ns, which corresponds to a maximum achievable frequency of 150.17 MHz.

Following successful simulation, the IP core's bitstream is deployed onto the ZedBoard platform to validate the core's functionality in a real-world environment and ensure its robustness under actual operating conditions. The IP core is tested in loopback mode, where the received data stored in the RX FIFO is fed directly to the TX FIFO through the AXI-4 master-slave buses. This setup allows for straightforward verification of the core's operation by comparing the transmitted and received data. The hardware architecture used for testing is depicted in the block diagram shown in **Fig. 9**. A Clocking Wizard IP core is utilized to generate a 100 MHz operation clock for the AXI-4 Stream IP core from an input clock coming from the crystal oscillator of the platform.

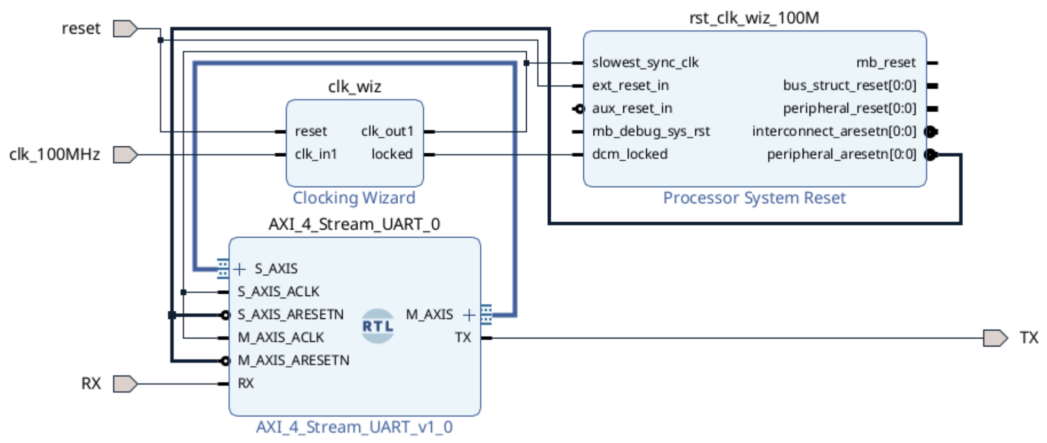

Fig. 9 - The experimentation setup in the Block Diagram tool of Vivado.

The input and output signals for this testing design and their mapping to the platform's pins are listed in **Table 2**. The experimental setup also includes a personal computer connected to the FPGA fabric of the ZedBoard, where the IP core is deployed, via a USB port which communicates with the IP core through a USB-to-UART bridge circuit from Digilent, connected to pins JB2 and JB3. Data coming from the computer are fed back again and depicted in the screen indicating the proper operation of the IP core.

Table 2 - Pinout assignment for the testing setup.

Signal Name	Chip Pin Name	Board Connector Name	Description
reset	T18	BTNU	Active high system reset signal.
clk_100MHz	Y9	GCLK	100 MHz operation clock.
RX	V10	JB3	UART serial receiver channel.
TX	W11	JB2	UART serial transmitter channel.

References

[1] Prodigy Technovations. UART Protocol. https://www.prodigytechno.com/uart-protocol

[2] Wikipedia. Universal asynchronous receiver-transmitter. https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

[3] Analog Devices. Eric Pena, Mary Grace Legaspi. UART: A Hardware Communication Protocol - Understanding Universal Asynchronous Receiver/ Transmitter. https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

[4] Digilent. Pmod USBUART: USB to UART Interface. https://digilent.com/shop/pmod-usbuart-usb-to-uart-interface/

[5] AMD. Technical Information Portal. UG1399. How AXI-Stream Works. https://docs.amd.com/r/en-US/ug1399-vitis-hls/How-AXI4-Stream-Works