**Requirements Document for UART Protocol with FIFO in VHDL**

**1. Project Overview**

The goal of this project is to design a Universal Asynchronous Receiver/Transmitter (UART) protocol with FIFO (First-In-First-Out) buffers for efficient data transmission and reception. This protocol will be implemented using VHDL for integration into FPGA or ASIC hardware systems.

**2. Scope of Work**

The UART protocol will facilitate asynchronous data communication between devices, supporting configurable baud rates, data bits, parity, and stop bits. The FIFO buffers will be used to manage incoming and outgoing data, preventing data loss during transmission and allowing for more efficient data handling.

**3. Functional Requirements**

**3.1 UART Transmitter (TX)**

* **Data Transmission**: The UART transmitter must transmit data asynchronously from the FIFO buffer to the output pin (TXD).
* **Baud Rate Configuration**: The transmitter must support different baud rates. A baud rate generator should be implemented to control the timing of data transmission.
* **Data Bits**: The transmitter should support configurable data bits (typically 5, 6, 7, or 8 bits per data frame).
* **Parity Bit**: The transmitter must support even, odd, and no parity.
* **Stop Bits**: The transmitter should support 1 or 2 stop bits for data framing.
* **FIFO Buffer**: The transmitter should use a FIFO buffer to hold outgoing data. If the FIFO is full, the transmitter should signal a 'buffer full' error or wait until space is available.

**3.2 UART Receiver (RX)**

* **Data Reception**: The UART receiver must receive data from the input pin (RXD) asynchronously and store it in a FIFO buffer.
* **Baud Rate Configuration**: Similar to the transmitter, the receiver must support the same configurable baud rates for proper synchronization.
* **Data Bits**: The receiver should support configurable data bits (5, 6, 7, or 8).
* **Parity Check**: The receiver should check the received data for parity errors (even, odd, or no parity).
* **Stop Bits**: The receiver should check the validity of the stop bits (1 or 2 stop bits).
* **FIFO Buffer**: The receiver should use a FIFO buffer to store incoming data. If the FIFO is full, the receiver should signal an error or drop data until the FIFO has space.

**3.3 FIFO Implementation**

* **Size**: The FIFO buffer size should be configurable (e.g., 8, 16, 32, or 64 entries).
* **Data Width**: The FIFO width must be the same as the configured data bits (5, 6, 7, or 8 bits).
* **Read/Write Pointer**: Implement pointers for reading and writing data to and from the FIFO. The write pointer increments when new data is written to the FIFO, and the read pointer increments when data is read.
* **Empty and Full Flags**: Provide flags for FIFO empty and full conditions, signaling when the FIFO has no data to read or no space to write.
* **Overflow/Underflow Protection**: Ensure the FIFO handles overflow (in case the FIFO becomes full during transmission) and underflow (when reading data from an empty FIFO).

**4. System Requirements**

**4.1 Clocking and Timing**

* **Clock Frequency**: The UART module should operate on a system clock, with a baud rate generator that derives the proper timing for transmission and reception.
* **Baud Rate Generator**: A clock divider or baud rate generator must be implemented to provide the appropriate timing for data transfer. The baud rate should be programmable based on the system clock frequency.

**4.2 Signal Interface**

* **TXD (Transmit Data)**: Output pin for transmitting data serially.
* **RXD (Receive Data)**: Input pin for receiving data serially.
* **TX Ready Flag**: An output flag that indicates when the UART transmitter is ready to transmit new data.
* **RX Ready Flag**: An output flag that indicates when the UART receiver has new data available.
* **FIFO Full Flag (TX and RX)**: A flag that signals when the FIFO is full and cannot accept new data.
* **FIFO Empty Flag (TX and RX)**: A flag that signals when the FIFO is empty and there is no data available for transmission or reception.

**4.3 Error Handling**

* **Parity Error**: The receiver must detect any parity errors and generate an interrupt or error flag.
* **Framing Error**: The receiver should detect invalid framing (incorrect stop bits) and generate an error flag.
* **Overrun Error**: The receiver should detect buffer overflow if data is lost due to a full FIFO.

**5. Performance Requirements**

* **Data Throughput**: The system should support a minimum of 115200 baud rate, with scalability to higher baud rates (e.g., 1Mbps or more).
* **Latency**: The system should introduce minimal delay during data transmission and reception.

**6. Design Constraints**

* **VHDL Coding Standards**: The design should adhere to industry-standard VHDL coding practices, ensuring readability, maintainability, and synthesis compatibility.
* **Synthesis and Implementation**: The design must be synthesizable for FPGA or ASIC platforms.
* **Resource Utilization**: The design should be optimized to minimize resource usage (e.g., logic elements, memory) while still supporting the required functionality.

**7. Testbench and Verification**

* **Functional Verification**: A testbench must be created to simulate the UART functionality, ensuring that the data is correctly transmitted and received.
* **FIFO Testing**: Test the proper operation of the FIFO buffers, ensuring that data is written and read correctly, and that overflow/underflow conditions are handled.
* **Error Condition Testing**: Verify that error flags are triggered under the appropriate conditions (e.g., parity error, framing error, buffer overflow).

**8. Documentation**

* **VHDL Code Documentation**: The VHDL code should include comments explaining the functionality of key components, such as the transmitter, receiver, and FIFO buffer.
* **Block Diagram**: A high-level block diagram should be provided to illustrate the architecture of the UART system.
* **User Manual**: A user manual should be provided for configuring and operating the UART, including how to set the baud rate, data bits, parity, and stop bits.

**9. Testing and Debugging**

* **Simulation**: Perform extensive simulation to validate the design against the specifications, including edge cases like buffer overflow/underflow and parity errors.
* **Hardware Validation**: Once the design is synthesized, validate the system on a development board or FPGA platform.
* **Debug Interface**: Include a debug interface (e.g., UART monitoring) for testing and troubleshooting purposes.

**10. Timeline and Milestones**

* **Phase 1 - Design and Specification**: Complete by [Date].
* **Phase 2 - VHDL Implementation**: Complete by [Date].
* **Phase 3 - Testbench Development and Simulation**: Complete by [Date].
* **Phase 4 - Synthesis and Implementation on Hardware**: Complete by [Date].
* **Phase 5 - Final Testing and Validation**: Complete by [Date].

**11. References**

* [Xilinx UART IP Core Documentation](https://www.xilinx.com)
* [Intel UART IP Core Documentation](https://www.intel.com)

This document outlines the complete requirements for designing a UART protocol with FIFO buffers in VHDL, ensuring the system meets performance, resource, and functional criteria.

Here’s a more advanced **VHDL code for a UART** design that includes **FIFO buffers** for both the transmitter and receiver. The FIFO will manage data and help avoid data loss during transmission and reception, ensuring smoother communication at high data rates.

**VHDL Code for UART with FIFO**

-- UART\_with\_FIFO.vhdl

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

use IEEE.STD\_LOGIC\_ARITH.ALL;

use IEEE.STD\_LOGIC\_UNSIGNED.ALL;

entity UART\_FIFO is

Port (

clk : in STD\_LOGIC; -- System clock

rst : in STD\_LOGIC; -- Reset signal

rx : in STD\_LOGIC; -- RXD (Receive Data)

tx : out STD\_LOGIC; -- TXD (Transmit Data)

tx\_fifo\_wr : out STD\_LOGIC; -- Write enable for TX FIFO

tx\_fifo\_rd : in STD\_LOGIC; -- Read enable for TX FIFO

rx\_fifo\_wr : in STD\_LOGIC; -- Write enable for RX FIFO

rx\_fifo\_rd : out STD\_LOGIC; -- Read enable for RX FIFO

tx\_data : in STD\_LOGIC\_VECTOR(7 downto 0); -- Data to transmit (8 bits)

tx\_start : in STD\_LOGIC; -- Start transmission signal

rx\_data : out STD\_LOGIC\_VECTOR(7 downto 0); -- Received Data

rx\_ready : out STD\_LOGIC -- Flag indicating received data is ready

);

end UART\_FIFO;

architecture Behavioral of UART\_FIFO is

-- Constants for UART configuration

constant BAUD\_RATE : integer := 9600; -- Baud rate for transmission

constant CLOCK\_FREQ : integer := 50000000; -- System Clock Frequency (50 MHz)

constant DIVISOR : integer := CLOCK\_FREQ / BAUD\_RATE;

-- Signals for Transmitter

signal tx\_state : STD\_LOGIC\_VECTOR(2 downto 0) := "000"; -- State machine for TX

signal tx\_shift\_reg : STD\_LOGIC\_VECTOR(9 downto 0); -- 1 start bit + 8 data bits + 1 stop bit

signal tx\_bit\_cnt : integer range 0 to 9 := 0;

signal tx\_clock\_div : integer range 0 to DIVISOR := 0;

signal tx\_fifo\_empty : STD\_LOGIC := '1'; -- FIFO empty flag for TX

signal tx\_fifo\_data : STD\_LOGIC\_VECTOR(7 downto 0); -- Data to be sent from TX FIFO

-- Signals for Receiver

signal rx\_state : STD\_LOGIC\_VECTOR(2 downto 0) := "000"; -- State machine for RX

signal rx\_shift\_reg : STD\_LOGIC\_VECTOR(9 downto 0); -- 1 start bit + 8 data bits + 1 stop bit

signal rx\_bit\_cnt : integer range 0 to 9 := 0;

signal rx\_clock\_div : integer range 0 to DIVISOR := 0;

signal rx\_fifo\_full : STD\_LOGIC := '0'; -- FIFO full flag for RX

signal rx\_fifo\_data : STD\_LOGIC\_VECTOR(7 downto 0); -- Data received in RX FIFO

-- FIFO definitions for Transmitter and Receiver (Assume FIFO is 16 words deep)

type fifo\_type is array (0 to 15) of STD\_LOGIC\_VECTOR(7 downto 0); -- 8-bit data

signal tx\_fifo : fifo\_type := (others => (others => '0'));

signal rx\_fifo : fifo\_type := (others => (others => '0'));

signal tx\_fifo\_wr\_ptr, tx\_fifo\_rd\_ptr : integer range 0 to 15 := 0;

signal rx\_fifo\_wr\_ptr, rx\_fifo\_rd\_ptr : integer range 0 to 15 := 0;

begin

-- Transmitter Process

process(clk, rst)

begin

if rst = '1' then

tx\_state <= "000";

tx\_bit\_cnt <= 0;

tx\_fifo\_wr\_ptr <= 0;

tx\_fifo\_rd\_ptr <= 0;

tx\_clock\_div <= 0;

tx\_shift\_reg <= (others => '0');

tx\_fifo\_empty <= '1';

elsif rising\_edge(clk) then

-- Baud rate clock divider

if tx\_clock\_div < DIVISOR - 1 then

tx\_clock\_div <= tx\_clock\_div + 1;

else

tx\_clock\_div <= 0;

end if;

-- Transmitter state machine

if tx\_clock\_div = 0 then

case tx\_state is

when "000" => -- IDLE state

tx\_fifo\_empty <= '1'; -- Wait for data

if tx\_fifo\_rd\_ptr /= tx\_fifo\_wr\_ptr then -- FIFO is not empty

tx\_state <= "001"; -- Start transmission

tx\_shift\_reg <= "0" & tx\_fifo(tx\_fifo\_rd\_ptr) & "1"; -- Load data from FIFO

tx\_bit\_cnt <= 0;

end if;

when "001" => -- START state

if tx\_bit\_cnt < 1 then

tx\_bit\_cnt <= tx\_bit\_cnt + 1;

else

tx\_state <= "010"; -- Move to data state

tx\_bit\_cnt <= 0;

end if;

when "010" => -- DATA state

tx <= tx\_shift\_reg(0); -- Transmit LSB of data

if tx\_bit\_cnt < 8 then

tx\_shift\_reg <= '0' & tx\_shift\_reg(9 downto 1); -- Shift data

tx\_bit\_cnt <= tx\_bit\_cnt + 1;

else

tx\_state <= "011"; -- Move to stop state

tx\_bit\_cnt <= 0;

end if;

when "011" => -- STOP state

tx <= '1'; -- Send stop bit

if tx\_bit\_cnt < 1 then

tx\_bit\_cnt <= tx\_bit\_cnt + 1;

else

tx\_state <= "000"; -- Back to IDLE

tx\_fifo\_rd\_ptr <= (tx\_fifo\_rd\_ptr + 1) mod 16; -- Update read pointer

end if;

end case;

end if;

end if;

end process;

-- Receiver Process

process(clk, rst)

begin

if rst = '1' then

rx\_state <= "000";

rx\_bit\_cnt <= 0;

rx\_fifo\_wr\_ptr <= 0;

rx\_fifo\_rd\_ptr <= 0;

rx\_clock\_div <= 0;

rx\_shift\_reg <= (others => '0');

rx\_ready <= '0';

elsif rising\_edge(clk) then

-- Baud rate clock divider

if rx\_clock\_div < DIVISOR - 1 then

rx\_clock\_div <= rx\_clock\_div + 1;

else

rx\_clock\_div <= 0;

end if;

-- Receiver state machine

if rx\_clock\_div = 0 then

case rx\_state is

when "000" => -- WAIT\_START state

if rx = '0' then -- Detect start bit

rx\_state <= "001"; -- Start receiving

rx\_bit\_cnt <= 0;

end if;

when "001" => -- RECEIVE state

rx\_shift\_reg <= rx & rx\_shift\_reg(9 downto 1); -- Shift in data bit

if rx\_bit\_cnt < 8 then

rx\_bit\_cnt <= rx\_bit\_cnt + 1;

else

rx\_state <= "010"; -- Move to stop state

rx\_bit\_cnt <= 0;

end if;

when "010" => -- STOP state

if rx = '1' then -- Check stop bit

-- Write data to RX FIFO

rx\_fifo(rx\_fifo\_wr\_ptr) <= rx\_shift\_reg(8 downto 1);

rx\_fifo\_wr\_ptr <= (rx\_fifo\_wr\_ptr + 1) mod 16;

rx\_ready <= '1';

end if;

rx\_state <= "000"; -- Go back to WAIT\_START

end case;

end if;

end if;

end process;

-- TX FIFO Write Enable

tx\_fifo\_wr <= tx\_state = "000" and tx\_fifo\_rd\_ptr /= tx\_fifo\_wr\_ptr;

-- RX FIFO Read Enable

rx\_fifo\_rd <= rx\_fifo\_wr\_ptr /= rx\_fifo\_rd\_ptr;

-- Output the data from RX FIFO when it's ready

rx\_data <= rx\_fifo(rx\_fifo\_rd\_ptr);

end Behavioral;

**Key Points:**

1. **FIFO Implementation**:
   * The FIFO for both the transmitter and receiver is implemented using two fifo\_type arrays, each with 16 entries.
   * The transmitter and receiver FIFO pointers are managed using tx\_fifo\_rd\_ptr, tx\_fifo\_wr\_ptr for TX and rx\_fifo\_rd\_ptr, rx\_fifo\_wr\_ptr for RX.
   * Data is written to the FIFO when it's available (via tx\_fifo\_wr for the transmitter and rx\_fifo\_wr for the receiver).
   * Data is read from the FIFO via tx\_fifo\_rd and rx\_fifo\_rd respectively.
2. **Transmitter**:
   * The transmitter has a state machine (IDLE, START, DATA, and STOP states) that processes the data from the FIFO for transmission.
   * The transmitter checks if data is available in the FIFO

and begins transmission once it has data.

1. **Receiver**:
   * The receiver detects the start bit and shifts in the received data bit-by-bit.
   * Once the data is fully received (including the stop bit), it is stored in the receiver FIFO.
2. **Baud Rate Control**:
   * Both the transmitter and receiver use a clock divider (tx\_clock\_div and rx\_clock\_div) to ensure data transmission and reception occur at the correct baud rate based on the system clock.

**Usage and Expansion:**

* **FIFO Size**: The FIFO is currently 16 words deep. You can increase the size of the FIFO by adjusting the array bounds.
* **FIFO Flags**: Flags like tx\_fifo\_empty, rx\_fifo\_full, rx\_ready are used to indicate the status of the FIFO and data availability.
* **Interrupts**: You can add interrupts or flags for signaling when the FIFO is full or empty, or when new data is available.

This design is a flexible base for creating UART communication with FIFO management, allowing for efficient and reliable data transmission and reception.