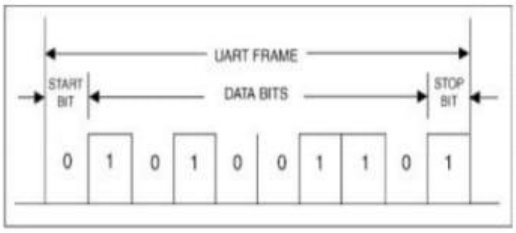
UART Protocol

UART (Universal Asynchronous Receiver-Transmitter), serves as a crucial asynchronous serial communication protocol predominantly employed in embedded systems. Its parameters such as BAUD rate, number of data bits, parity bits, stop bits, and flow control are adjustable, offering flexibility in communication setups. Given its asynchronous nature, both the transmitter and receiver must synchronize on the BAUD rate and adhere to the same frame structure for successful data transmission.



TX

a) 1 START bit
b) 5, 6, 7, or 8 data bits
c) 1 PARITY bit (optional)
d) 1, 1.5, or 2 STOP bits

RX

a) 1 START bit
b) 5, 6, 7, or 8 data bits
c) 1 PARITY bit (optional)
d) 1 STOP bit (any other STOP bits transferred with the above data are not detected)

Typically, during idle periods, the UART transmission line maintains a high state. When data transmission initiates, the line voltage transitions from high to low, signaling the receiver to commence sampling incoming data. This mechanism ensures efficient and reliable communication between devices.

Code

```
module uart #(
    parameter CLOCKS_PER_PULSE = 5208
)
(
    input logic [3:0] data_in,          // Input data to be transmitted
    input logic data_en,                 // Data enable signal
    input logic clk,                     // System clock
    input logic rstn,                    // Active low reset signal
    output logic tx,                     // Transmitted data output
    output logic tx_busy,                // Transmission status output
    input logic ready_clr,              // Ready clear signal for receiver
    input logic rx,                      // Received data input
    output logic ready,                  // Ready signal indicating data reception
completion
    output logic [3:0] led_out,          // Output for LED display
    output logic [6:0] display_out       // Output for 7-segment display
);
    logic [7:0] data_input;              // Input data for transmitter
    logic [7:0] data_output;             // Output data from receiver

    // Instantiate transmitter module
    transmitter #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE)) uart_tx (
        .data_in(data_input),
        .data_en(data_en),
        .clk(clk),
        .rstn(rstn),
        .tx(tx),
        .tx_busy(tx_busy)
    );
    // Instantiate receiver module
    receiver #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE)) uart_rx (
        .clk(clk),
        .rstn(rstn),
        .ready_clr(ready_clr),
        .rx(rx),
        .ready(ready),
        .data_out(data_output)
    );

    // Instantiate binary-to-7-segment converter module
    binary_to_7seg converter (
        .data_in(data_output[3:0]),
        .data_out(display_out)
    );

    // Connect input data to transmitter
    assign data_input = {4'b0, data_in};

    // Connect LED output to lower nibble of received data
    assign led_out = data_output[3:0];

endmodule
```

Transmitter

The Verilog module presented encapsulates a robust implementation of a serial transmitter. Operating within a concise framework, it leverages an internal state machine to orchestrate the synchronized transmission of data. Supporting an 8-bit data input, along with pivotal signals including data enable, clock, and asynchronous reset, the module establishes a solid foundation for its functionality. With operations spanning across four states—IDLE, START, DATA, and END—the module navigates through each phase with precision. Upon initiation of data transmission, it seamlessly progresses through these states, ensuring the transmission of individual bits in harmony with the clock signal. Furthermore, the module provides a crucial output signal, indicating the operational status of the transmitter, whether it's actively transmitting data or in an idle state. This Verilog code begins with the module declaration, followed by the definition of input and output ports. An enumeration is then established to define transmission states. Internal variables are initialized, and the core logic is implemented within an `always_ff` block, determining transmitter behavior based on state. Finally, an assignment statement sets the output signal indicating transmitter status. This systematic structure ensures a comprehensive and efficient implementation of the serial transmitter functionality.

Code

```
module transmitter #(
    parameter CLOCKS_PER_PULSE = 16
)
(
    input logic [7:0] data_in,      // Input data to be transmitted
    input logic data_en,             // Data enable signal
    input logic clk,                 // System clock
    input logic rstn,                // Active low reset signal
    output logic tx,                 // Transmitted data output
    output logic tx_busy             // Transmission status output
);
    enum {TX_IDLE, TX_START, TX_DATA, TX_END} state; // Define transmission states

    logic [7:0] tx_data = 8'b0;       // Data to be transmitted
    logic [2:0] bit_counter = 3'b0;    // Bit counter to keep track of transmitted
bits
    logic [$clog2(CLOCKS_PER_PULSE)-1:0] pulse_counter = 0; // Counter for clock
pulses

    always_ff @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            // Reset all variables and state to initial values
            pulse_counter <= 0;
            bit_counter <= 0;
            tx_data <= 0;
            tx <= 1'b1;           // Set default value for tx signal
            state <= TX_IDLE;     // Set initial state to TX_IDLE
        end else begin
            case (state)
                TX_IDLE: begin
                    if (~data_en) begin
                        state <= TX_START;
                        tx_data <= data_in;       // Load data to be transmitted
                        bit_counter <= 3'b0;      // Reset bit counter
                        pulse_counter <= 0;       // Reset pulse counter
                    end else tx <= 1'b1;           // Keep transmitter idle if no
data to send
                end
                TX_START: begin
                    if (pulse_counter == CLOCKS_PER_PULSE-1) begin
                        state <= TX_DATA;         // Move to TX_DATA state after
start bit transmission
                        pulse_counter <= 0;       // Reset pulse counter
                    end else begin
                        tx <= 1'b0;               // Start bit transmission
                        pulse_counter <= pulse_counter + 1; // Increment pulse
counter
                    end
                end
                TX_DATA: begin
                    if (pulse_counter == CLOCKS_PER_PULSE-1) begin
                        pulse_counter <= 0;       // Reset pulse counter
                        if (bit_counter == 3'd7) begin
                            state <= TX_END;      // Move to TX_END state after all
data bits transmitted
                        end else begin
                            bit_counter <= bit_counter + 1; // Move to the next bit
                            tx <= tx_data[bit_counter];    // Transmit the next bit
                        end
                    end else begin
                        tx <= tx_data[bit_counter];    // Transmit the current bit
                        pulse_counter <= pulse_counter + 1; // Increment pulse 
counter
                    end
                end
                TX_END: begin
                    if (pulse_counter == CLOCKS_PER_PULSE-1) begin
                        state <= TX_IDLE;           // Move back to TX_IDLE state 
after stop bit transmission
                        pulse_counter <= 0;         // Reset pulse counter
                    end else begin
                        tx <= 1'b1;                 // Transmit stop bit
                        pulse_counter <= pulse_counter + 1; // Increment pulse 
counter
                    end
                end
                default: state <= TX_IDLE;          // Default state transition to 
TX_IDLE
            endcase
        end
    end
    assign tx_busy = (state != TX_IDLE);           // Output busy signal when not in 
idle state

endmodule
```

Receiver

The Verilog module provided is a UART (Universal Asynchronous Receiver-Transmitter) receiver designed for implementation on an FPGA (Field-Programmable Gate Array). It operates using a state machine to manage the reception of serial data. Upon receiving a start bit, it transitions through states to receive each data bit, and finally, detects the stop bit. The module includes counters to manage timing and synchronization logic to ensure proper reception of data. Upon successful reception of a byte, it sets a "ready" flag and outputs the received data. Overall, it provides a fundamental building block for UART communication within an FPGA-based system.

The below Verilog code represents a receiver module designed to receive data serially using a clock signal. It utilizes an internal state machine to control the reception process. The module synchronizes the incoming data signal, captures the data bit by bit, and stores it in a temporary register. Once all the bits are received, the module sets the ready signal and outputs the received data. The receiver transitions between states based on the clock cycles and resets to the idle state after data reception is complete.

Code

```
module receiver #(
    parameter CLOCKS_PER_PULSE = 16
)
(
    input logic clk,                   // System clock
    input logic rstn,                  // Active low reset signal
    input logic ready_clr,             // Ready clear signal
    input logic rx,                    // Received data input
    output logic ready,                // Ready signal indicating data reception
completion
    output logic [7:0] data_out        // Received data output
);

    enum {RX_IDLE, RX_START, RX_DATA, RX_END} state; // Define reception states

    logic [2:0] bit_counter;            // Bit counter to keep track of received bits
    logic [$clog2(CLOCKS_PER_PULSE)-1:0] pulse_counter; // Counter for clock pulses

    logic [7:0] temp_data;              // Temporary storage for received data
    logic rx_sync;                      // Synchronized received data

    always_ff @(posedge clk or negedge rstn) begin
        if (!rstn) begin
            // Reset all variables and state to initial values
            pulse_counter <= 0;
            bit_counter <= 0;
            temp_data <= 8'b0;
            ready <= 0;
            state <= RX_IDLE;
        end else begin
            rx_sync <= rx;  // Synchronize the input signal using a flip-flop

            case (state)
                RX_IDLE : begin
                    if (rx_sync == 0) begin
                        state <= RX_START; // Move to RX_START state upon detection
of start bit
                        pulse_counter <= 0; // Reset pulse counter
                    end
                end
                RX_START: begin
                    if (pulse_counter == CLOCKS_PER_PULSE/2-1) begin
                        state <= RX_DATA; // Move to RX_DATA state after half a clock
period
                        pulse_counter <= 0; // Reset pulse counter
                    end else
                        pulse_counter <= pulse_counter + 1; // Increment pulse
counter
                end
                RX_DATA : begin
                    if (pulse_counter == CLOCKS_PER_PULSE-1) begin
                        pulse_counter <= 0; // Reset pulse counter
                        temp_data[bit_counter] <= rx_sync; // Store received bit
                        if (bit_counter == 3'd7) begin
                            state <= RX_END; // Move to RX_END state after all data
bits received
```

```
                            bit_counter <= 0; // Reset bit counter
                        end else bit_counter <= bit_counter + 1; // Move to the next
bit
                    end else pulse_counter <= pulse_counter + 1; // Increment pulse
counter
                end
                RX_END : begin
                    if (pulse_counter == CLOCKS_PER_PULSE-1) begin
                        ready <= 1'b1; // Indicate readiness to output data
                        state <= RX_IDLE; // Move back to RX_IDLE state after stop
bit reception
                        pulse_counter <= 0; // Reset pulse counter
                    end else pulse_counter <= pulse_counter + 1; // Increment pulse
counter
                end
                default: state <= RX_IDLE; // Default state transition to RX_IDLE
            endcase
        end
    end
    assign data_out = temp_data; // Output received data
endmodule
```

Testbench

The testbench provided sets up a simulation environment for the UART module, initializing crucial signals like clock, reset, and data control signals. It configures and connects the UART module, ensuring smooth communication by setting parameters appropriately. The testbench generates data for transmission and closely monitors the received data. It meticulously compares the received data with the transmitted data to ensure accuracy. If any discrepancies are detected, it promptly flags errors and terminates the simulation. Conversely, upon successful verification of all transmitted bytes, it concludes the simulation, confirming the proper functionality of the UART module. This testbench serves as a reliable tool for evaluating the UART module's performance in a simulated environment.

Code

```
`timescale 1ns/1ps

module testbench();

    localparam CLOCKS_PER_PULSE = 4;
    logic [3:0] data_in = 4'b0001;
    logic clk = 0;
    logic rstn = 0;
    logic enable = 1;

    logic tx_busy;
    logic ready;
    logic [3:0] data_out;
    logic [7:0] display_out;

    logic loopback;
    logic ready_clr = 1;

    // Instantiate UART module for testing
    uart #(.CLOCKS_PER_PULSE(CLOCKS_PER_PULSE))
            test_uart(.data_in(data_in),
                        .data_en(enable),
                        .clk(clk),
                        .tx(loopback),
                        .tx_busy(tx_busy),
                        .rx(loopback),
                        .ready(ready),
                        .ready_clr(ready_clr),
                        .led_out(data_out),
                        .display_out(display_out),
                        .rstn(rstn)
                        );

    // Toggle clock
    always begin
        #1 clk = ~clk;
    end

    // Test initialization and data transmission
    initial begin
        $dumpfile("testbench.vcd");
        $dumpvars(0, testbench);
        rstn <= 1;                  // Assert reset initially
        enable <= 1'b0;             // Disable data transmission initially
        #2 rstn <= 0;               // Deassert reset
        #2 rstn <= 1;               // Assert reset again to initialize UART
        #5 enable <= 1'b1;          // Enable data transmission after initialization
    end

    // Monitor ready signal and check received data
    always @(posedge ready) begin
        if (data_out != data_in) begin
            $display("FAIL: rx data %x does not match tx %x", data_out, data_in);
            $finish();
        end else begin
            if (data_out == 4'b1111) begin // Check if received data is 11111111
                $display("SUCCESS: all bytes verified");
                $finish();
            end
            #10 rstn <= 0;              // Assert reset for next data transmission
            #2 rstn <= 1;               // Deassert reset
            data_in <= data_in + 1'b1;  // Increment data for next transmission
            enable <= 1'b0;             // Disable data transmission temporarily
            #2 enable <= 1'b1;          // Enable data transmission for the next
round
        end
    end
endmodule
```