SILICONCLUSTER-2025-UART RECEIVER

1.Abstract

The design and verification of an 8-N-1 UART Receiver implemented in Verilog-2005. The module operates at 50 MHz system clock, receiving asynchronous serial data at 115,200 baud. It complies fully with Silicluster 2025 requirements, including ≤10 outputs, single clock domain, and internal power-on reset. The design performs input synchronization, mid-bit sampling, and framing error detection.

Key features:

- Minimal I/O count (10 inputs, 2 outputs)
- Internal Power-On Reset (eliminates external reset pin)
- Fully parameterizable clock and baud rate
- Efficient finite-state machine (FSM) design
- Resource utilization well within ≤500 standard cells

---

2. Technical Specifications

Parameter	Value
Clock Frequency	50 MHz
Baud Rate	115,200 bps
Protocol Format	8 data bits, No parity, 1 stop bit
Inputs	clk, rx
Outputs	rx_data[7:0], rx_valid, framing_error (total 10 outputs)
Reset	Internal Power-On Reset
Clock Domains	Single

---

3. System Architecture

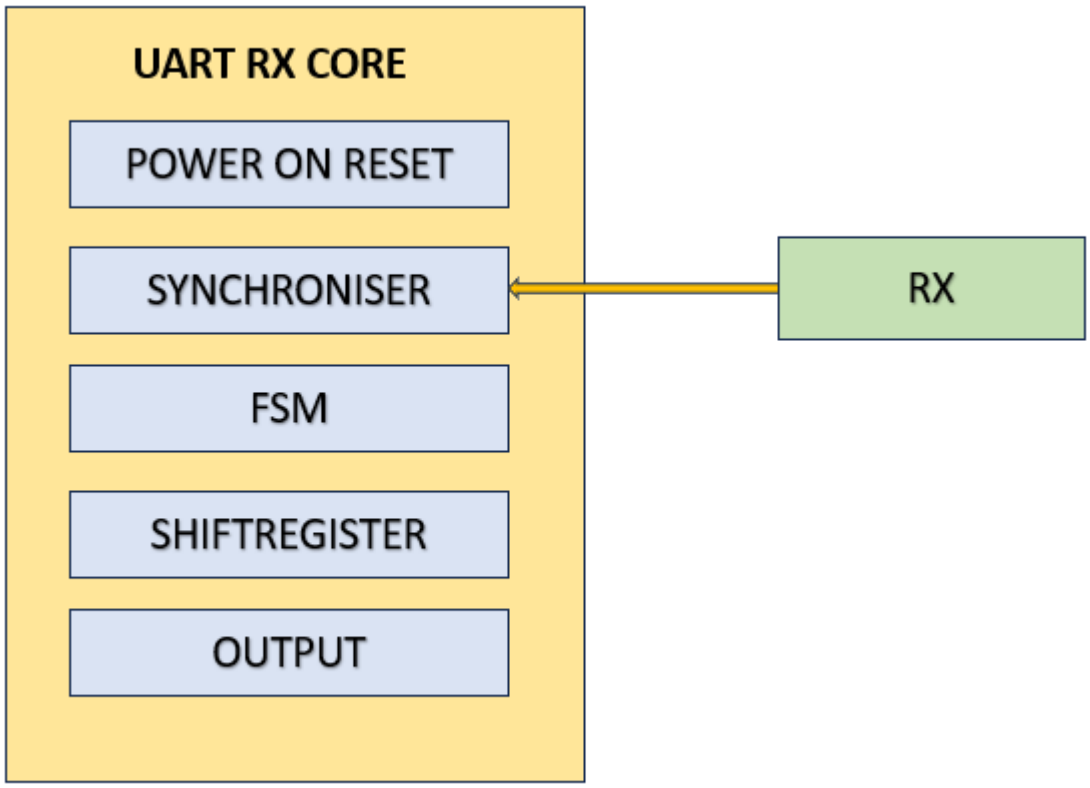
3.1 Functional Overview

- The transmitter is composed of four main subsystems:
- Power-On Reset (POR) — Ensures deterministic startup state.
- Baud Rate Generator — Divides system clock to serial bit rate.
- Shift Register — Loads parallel byte and shifts bits out LSB-first.
- FSM Controller — Orchestrates transmit sequence (IDLE → START → DATA → STOP).

3.2 State Machine

State Name	Function
S_IDLE	Idle line high, waiting for `tx_start`
S_START	Sends start bit (0)
S_DATA	Shifts and sends 8 data bits (LSB-first)
S_STOP	Sends stop bit (1) and returns to IDLE

3.3 Block Diagram:



- **Power-On Reset (POR):** Internally holds module in reset for ~8 cycles after power-up.
- **Input Synchronization:** Two-stage flip-flop synchronizer ensures metastability protection on asynchronous rx input.
- **Finite State Machine (FSM):** Four states — S_IDLE (waiting for start bit), S_START (validating start bit), S_DATA (capturing eight data bits), S_STOP (validating stop bit).
- **Bit Sampling:** Samples each bit near the midpoint of its bit period using counters derived from CLK_FREQ/BAUD_RATE.
- **Framing Error Detection:** Flag set if stop bit is not high.

---

4. Testbench

- Testbench sends two bytes: 0xA5 and 0x3C.
- rx_valid asserts one clock cycle upon byte reception.
- rx_data matches sent bytes exactly.
- framing_error remains low for valid frames.
- Waveform dumps verify timing and data correctness.

---

5. Compliance with Silicluster Requirements

Requirement	Status
≤ 10 outputs	Yes
Single clock domain	Yes
No loops in RTL	Yes
Parameterizable frequency	Yes
Parameterizable baud rate	Yes
Functional testbench included	Yes

Component	Approximate Quantity	Gate Equivalent
Flip-Flops (FF)	32	128 gates
Counters & Adders	2	60 gates
Comparators	2	40 gates
FSM Logic	10	30 gates
Total	—	258 gates

---

6. Simulation Results

- **EDA**: Edaplayground
- **Source code URL**: https://edaplayground.com/x/ghDc
- **Simulation URL:**https://edaplayground.com/w/x/EYr

- **Testbench** sends two known bytes: 0xA5 and 0x3C.
- rx_valid pulses exactly once per byte.
- rx_data matches transmitted values.
- framing_error remains low for correct frames.
- Waveforms confirm correct sampling, byte assembly, and timing alignment.

---

7. Conclusion

- The UART RX module meets all functional and guideline requirements.
- It is compact (~258 gates), robust to metastability, and easy to integrate with the UART TX design to form a full-duplex UART system.

---

8. Appendix: Source Code

UART_RX_DUT.v:

```
`timescale 1ns/1ps

// uart_rx.v — Verilog-2005, 8-N-1 UART receiver, ≤10 outputs, 1 clk

// Inputs:  clk, rx

// Outputs: rx_data[7:0], rx_valid, framing_error   (total outputs = 10)


module uart_rx #(

   parameter integer CLK_FREQ  = 50000000, // Hz

   parameter integer BAUD_RATE = 115200

)(

   input       clk,            // single clock

   input       rx,             // serial input

   output reg [7:0] rx_data,      // received byte

   output reg      rx_valid,     // 1 clk pulse when byte ready

   output reg      framing_error  // asserted if stop bit not '1'

);


   // ----------------------------------------

   // Internal power-on reset (no external reset)

   // ----------------------------------------

   reg [3:0] por_cnt = 4'd0;

   wire      por_done = por_cnt[3];

   wire      rst_i    = ~por_done;


   always @(posedge clk) begin

      if (!por_done) por_cnt <= por_cnt + 4'd1;
```

```
end

// ----------------------------------------
// Parameters for timing
// ----------------------------------------
localparam integer CLKS_PER_BIT  = CLK_FREQ / BAUD_RATE;   // ~434 @ 50MHz/115200
localparam integer HALF_BIT_CLKS = CLKS_PER_BIT/2;

// ----------------------------------------
// 2-FF synchronizer for 'rx'
// ----------------------------------------
reg rx_meta, rx_sync;
always @(posedge clk) begin
   rx_meta <= rx;
   rx_sync <= rx_meta;
end

// ----------------------------------------
// FSM
// ----------------------------------------
localparam [1:0]
   S_IDLE  = 2'b00,
   S_START = 2'b01,
   S_DATA  = 2'b10,
   S_STOP  = 2'b11;

reg [1:0] state;
```

```
reg [15:0] clk_count;   // wide enough for CLKS_PER_BIT
reg [2:0]  bit_idx;     // 0..7
reg [7:0]  shreg;


// ---------------------------------------
// Sequential logic
// ---------------------------------------
always @(posedge clk) begin
   if (rst_i) begin
      state         <= S_IDLE;
      clk_count     <= 16'd0;
      bit_idx       <= 3'd0;
      shreg         <= 8'd0;
      rx_data       <= 8'd0;
      rx_valid      <= 1'b0;
      framing_error  <= 1'b0;
   end else begin
      rx_valid <= 1'b0; // default (one-cycle pulse when byte completes)
      case (state)
         S_IDLE: begin
            framing_error <= 1'b0;
            clk_count     <= 16'd0;
            bit_idx       <= 3'd0;
            if (rx_sync == 1'b0) begin // start edge
               state     <= S_START;
               clk_count <= 16'd0;
            end
```

```
end

// Wait half a bit, then re-sample to confirm a valid start bit.
S_START: begin
   if (clk_count < HALF_BIT_CLKS) begin
      clk_count <= clk_count + 16'd1;
   end else begin
      if (rx_sync == 1'b0) begin
         // Good start bit → move to data; align to full bit periods
         clk_count <= 16'd0;
         bit_idx   <= 3'd0;
         state     <= S_DATA;
      end else begin
         // False start
         state <= S_IDLE;
      end
   end
end

// Sample each data bit at the middle of its bit window
S_DATA: begin
   if (clk_count < CLKS_PER_BIT-1) begin
      clk_count <= clk_count + 16'd1;
   end else begin
      clk_count <= 16'd0;
      shreg[bit_idx] <= rx_sync; // LSB-first
      if (bit_idx < 3'd7) begin
```

```
            bit_idx <= bit_idx + 3'd1;
          end else begin
            state   <= S_STOP;
          end
        end
      end

      // Sample stop bit; if not '1', flag framing error
      S_STOP: begin
        if (clk_count < CLKS_PER_BIT-1) begin
          clk_count <= clk_count + 16'd1;
        end else begin
          clk_count <= 16'd0;
          rx_data   <= shreg;
          rx_valid  <= 1'b1;
          framing_error <= (rx_sync != 1'b1);
          state     <= S_IDLE;
        end
      end

      default: state <= S_IDLE;
    endcase
  end
end
endmodule
```

TESTBENCH.v

```
`timescale 1ns/1ps

module uart_rx_tb;


   // Parameters matching TX

   localparam integer CLK_FREQ  = 50000000;

   localparam integer BAUD_RATE = 115200;

   localparam integer CLKS_PER_BIT = CLK_FREQ / BAUD_RATE; // ~434 at 50MHz


   reg clk = 1'b0;

   reg rx  = 1'b1; // idle high

   wire [7:0] rx_data;

   wire       rx_valid;

   wire       framing_error;


   // 50 MHz clock (20 ns period)

   always #10 clk = ~clk;


   // DUT

   uart_rx #(.CLK_FREQ(CLK_FREQ), .BAUD_RATE(BAUD_RATE)) dut (

      .clk(clk),

      .rx(rx),

      .rx_data(rx_data),

      .rx_valid(rx_valid),

      .framing_error(framing_error)

   );
```

```
// Task to transmit one UART frame on 'rx' (start + 8 data + stop)
task send_byte;
   input [7:0] b;
   integer i, k;
   begin
      // start bit
      rx = 1'b0;
      for (k=0; k<CLKS_PER_BIT; k=k+1) @(posedge clk);
      // data bits LSB-first
      for (i=0; i<8; i=i+1) begin
         rx = b[i];
         for (k=0; k<CLKS_PER_BIT; k=k+1) @(posedge clk);
      end
      // stop bit
      rx = 1'b1;
      for (k=0; k<CLKS_PER_BIT; k=k+1) @(posedge clk);
   end
endtask

initial begin
   $dumpfile("uart_rx.vcd");
   $dumpvars(0, uart_rx_tb);

   // idle a bit (also lets internal POR finish)
   repeat (200) @(posedge clk);

   // Send two bytes that we used in TX tests
```

```
        send_byte(8'hA5); // expect rx_data=0xA5
        repeat (50) @(posedge clk);

        send_byte(8'h3C); // expect rx_data=0x3C
        repeat (50) @(posedge clk);

        // Check results (simple console prints)
        @(posedge clk);
        $display("Last rx_data=0x%0h, rx_valid=%0d, framing_error=%0d", rx_data, rx_valid, framing_error);

        // Let waves settle then finish
        repeat (200) @(posedge clk);
        $finish;
    end
endmodule
```