ChatGPT

UART Packetizer System Design

Packetizer FSM

The packetizer is a finite-state machine (FSM) that controls reading bytes from the FIFO and initiating UART transmission. It monitors the FIFO status and the external `tx_ready` signal, and generates a framed serial packet with start/data/stop bits. In the **IDLE** state, `tx_busy` is low; when the FIFO is non-empty and `tx_ready` is asserted, the FSM moves to **READ_FIFO**, asserts `fifo_rd_en`, and waits for the byte to become available. In **READ_FIFO** it captures the 8-bit data out of the FIFO. Next, the FSM enters **TRANSMIT**: it sets a `tx_start` pulse (beginning the UART block), loads the byte into the UART transmitter, and raises `tx_busy`. The UART module then shifts out the start bit, data bits, and stop bit(s) in sequence. The FSM monitors the transmission and, once the stop bit is sent, returns to **IDLE**, lowering `tx_busy` again.

- **IDLE**: `fifo_empty`=0 *and* `tx_ready`=1 → assert `fifo_rd_en`, go to READ_FIFO.
- **READ_FIFO**: Wait for `data_out_valid`; load `data_out` into a shift register.
- **TRANSMIT**: Set `tx_busy=1`, assert `tx_start` to UART block. UART sends 1 start bit (low), 8 data bits (LSB first), and 1 stop bit (high) [1] [2].
- **DONE**: After transmission completes (UART clears busy), set `tx_busy=0` and return to IDLE.

Each state is a Moore-style step: outputs (`fifo_rd_en`, `tx_start`, `tx_busy`) depend on the current state (and in IDLE on input conditions). The FSM ensures that each byte is only read from FIFO when the receiver is ready, framing it correctly before moving on.

```
module packetizer_fsm(
    input  wire       clk,        // system clock
    input  wire       rst,        // active-high synchronous reset
    input  wire       tx_ready,   // external ready signal
    input  wire       fifo_empty, // FIFO empty flag
    input  wire [7:0] fifo_data,  // data read from FIFO
    input  wire       data_out_valid, // FIFO data valid
    output reg        fifo_rd_en, // enable FIFO read
    output reg [7:0]  tx_data,    // data byte to transmit
    output reg        tx_start,   // pulse to start UART send
    output reg        tx_busy     // high during transmission
);
  // State encoding
  typedef enum logic [1:0] {IDLE, READ, SEND, WAIT_DONE} state_t;
  state_t state, next_state;
  always @(posedge clk) begin
    if (rst) state <= IDLE;
    else      state <= next_state;
  end
```

```
  always @(*) begin
    // Default outputs
    fifo_rd_en = 1'b0; tx_start = 1'b0; tx_busy = 1'b0; next_state = state;
    case (state)
      IDLE: begin
        if (!fifo_empty && tx_ready) begin
          fifo_rd_en = 1'b1;
          next_state = READ;
        end
      end
      READ: begin
        if (data_out_valid) begin
          // latch data and start transmission
          next_state = SEND;
        end
      end
      SEND: begin
        tx_start = 1'b1;
        tx_busy = 1'b1;
        next_state = WAIT_DONE;
      end
      WAIT_DONE: begin
        tx_busy = 1'b1;
        // Wait until UART clears busy; return to IDLE
        if (!tx_start) // assume UART deasserts start and busy after send
          next_state = IDLE;
      end
    endcase
  end

  // Latch FIFO data when valid
  always @(posedge clk) begin
    if (state == READ && data_out_valid) begin
      tx_data <= fifo_data;
    end
  end
endmodule
```

Asynchronous FIFO Buffer

We use an 8-bit wide, 16-depth FIFO to buffer incoming data. This FIFO decouples the data producer (writing bytes) from the consumer (UART packetizer). Asynchronous FIFOs are common when data must cross clock domains or operate at independent rates [3]. Here, we assume a single 50 MHz clock, but the design supports an independent read clock if needed. We implement dual-port RAM with separate read and

write logic. The FIFO uses internal read/write pointers (Gray-coded for safe clock-domain crossing) and generates `fifo_full` and `fifo_empty` flags.

- **Data Width & Depth:** 8 bits wide, 16 entries (a power-of-2 size simplifies Gray-code pointer arithmetic [4]).
- **Write Side:** On `data_valid` (pulsed high) and when `!fifo_full`, the input byte is written into memory, and the write pointer increments.
- **Read Side:** When `fifo_rd_en` is asserted by the FSM and `!fifo_empty`, the next byte is presented at `data_out`; a `data_out_valid` pulse indicates valid data. Read pointer increments accordingly.
- **Flags:** `fifo_full` goes high when the FIFO is full (write pointer one step behind read pointer in a cycle), and `fifo_empty` when no data remains. These flags let the FSM know when it can or cannot read [3].

```
module async_fifo (
    input  wire       clk_wr,      // write-side clock (sys clk)
    input  wire       clk_rd,      // read-side clock (can be same or separate)
    input  wire       rst,         // active-high reset
    input  wire       wr_en,       // write enable
    input  wire [7:0] wr_data,     // data in
    input  wire       rd_en,       // read enable
    output reg  [7:0] rd_data,     // data out
    output reg        fifo_full,
    output reg        fifo_empty,
    output reg        data_out_valid
);
  localparam DEPTH = 16;
  reg [3:0] wr_ptr=0, rd_ptr=0;
  reg [7:0] mem [0:DEPTH-1];
  reg [4:0] count=0;

  // Write logic
  always @(posedge clk_wr) begin
    if (rst) begin
      wr_ptr <= 0;
      fifo_full <= 0;
    end else begin
      if (wr_en && !fifo_full) begin
        mem[wr_ptr] <= wr_data;
        wr_ptr <= wr_ptr + 1;
      end
      fifo_full <= (count == DEPTH);
    end
  end

  // Read logic
  always @(posedge clk_rd) begin
```

```
    if (rst) begin
      rd_ptr <= 0;
      fifo_empty <= 1;
      rd_data <= 0;
      data_out_valid <= 0;
    end else begin
      data_out_valid <= 0;
      if (rd_en && !fifo_empty) begin
        rd_data <= mem[rd_ptr];
        rd_ptr <= rd_ptr + 1;
        data_out_valid <= 1;  // one-cycle pulse when data is valid
      end
      fifo_empty <= (count == 0);
    end
  end

  // Occupancy count
  always @(posedge clk_wr or posedge clk_rd) begin
    if (rst) count <= 0;
    else begin
      // This simplistic count assumes clk_wr == clk_rd. For true async, Gray
pointers and CDC logic needed.
      if (wr_en && !fifo_full && !(rd_en && !fifo_empty)) count <= count + 1;
      else if (rd_en && !fifo_empty && !(wr_en && !fifo_full)) count <= count -
1;
    end
  end
endmodule
```

> **Note:** In a full asynchronous FIFO, the write and read clocks may differ. Gray-code pointers and synchronizers are used to cross domains without metastability [5] [4]. Here we assume a common clock for simplicity.

UART Serial Output Block

The UART transmitter converts an 8-bit parallel byte into a serial bitstream with framing bits. A simple **baud-rate generator** divides the 50 MHz clock down to the desired bit rate (e.g. 9600 or 115200 baud). When `tx_start` is asserted, the module loads the byte into a shift register. It then asserts a low **start bit** for one bit period, shifts out the 8 data bits (least-significant bit first [1]), and finally asserts a high **stop bit** for one bit period. The line idles high when not transmitting [2], and the `tx_busy` output remains high throughout the start/data/stop sequence.

The diagram above illustrates a typical UART frame (8N1 format). It has a logic-low start bit, followed by 8 data bits (LSB first), and a logic-high stop bit [1] [2]. During idle the line is high; pulling it low signals the start of transmission [2]. After all bits are sent, driving the line high for one bit time serves as the stop bit [6]. The entire framed packet is thus transmitted serially at the configured baud rate [7].

Key parts of the UART module include: - **Clock Divider:** A counter that generates a 1-clock pulse (`bit_tick`) every `CLK_FREQ/BAUD_RATE` system clocks.
- **Shift Register:** On `tx_start`, captures the 8-bit `tx_data`. Each `bit_tick` shifts out one bit onto `serial_out`.
- **State/Counter:** Tracks which bit is being sent. Typically: start bit (0), bits 0–7 of data, then stop bit (1).
- **Busy Flag:** `tx_busy` is asserted when sending starts and deasserted only after the stop bit. This lets the FSM know transmission is in progress.

```
module uart_tx #(
    parameter CLK_FREQ = 50000000,
    parameter BAUD = 115200
)(
    input  wire       clk,
    input  wire       rst,
    input  wire       tx_start,   // pulse to start transmission
    input  wire [7:0] tx_data,    // byte to send
    output reg        serial_out, // UART TX line
    output reg        tx_busy     // high while sending
);
  localparam DIV = CLK_FREQ/BAUD;
  reg [15:0] baud_cnt=0;
  reg        bit_tick=0;
  reg [3:0]  bit_pos=0;    // 0=start, 1-8=data, 9=stop
  reg [7:0]  shift_reg=8'hFF;

  // Baud rate generator
  always @(posedge clk) begin
    if (rst) begin
      baud_cnt <= 0; bit_tick <= 0;
    end else if (tx_busy) begin
      if (baud_cnt == DIV-1) begin
        baud_cnt <= 0;
        bit_tick <= 1;
      end else begin
        baud_cnt <= baud_cnt + 1;
        bit_tick <= 0;
      end
    end else begin
      baud_cnt <= 0;
      bit_tick <= 0;
    end
  end

  // Transmission state machine
  always @(posedge clk) begin
    if (rst) begin
      serial_out <= 1;  // idle high
```

```
      tx_busy <= 0;
      bit_pos <= 0;
    end else begin
      if (tx_start && !tx_busy) begin
        // Begin transmission
        tx_busy <= 1;
        bit_pos <= 0;
        shift_reg <= tx_data;
        serial_out <= 0; // start bit
      end else if (tx_busy && bit_tick) begin
        if (bit_pos < 8) begin
          // Send data bits LSB first
          serial_out <= shift_reg[0];
          shift_reg <= {1'b1, shift_reg[7:1]}; // shift right, fill with 1s for
stop
        end else if (bit_pos == 8) begin
          // Send stop bit (line high)
          serial_out <= 1;
        end
        bit_pos <= bit_pos + 1;
        if (bit_pos == 9) begin
          // Done after stop bit
          tx_busy <= 0;
          bit_pos <= 0;
        end
      end
    end
  end
endmodule
```

Top-Level Module Integration

The top module instantiates and connects the FIFO, FSM, and UART blocks. Input bytes and validity pulses feed into the FIFO; the FSM reads from the FIFO and feeds the UART. The system outputs the FIFO full flag, the serial line, and the busy flag. For example:

```
module top_uart_packetizer (
    input  wire       clk,        // 50 MHz system clock
    input  wire       rst,        // active-high sync reset
    input  wire [7:0] data_in,    // input byte
    input  wire       data_valid, // pulse indicating data_in is valid
    input  wire       tx_ready,   // external ready handshaking
    output wire       serial_out, // UART TX output
    output wire       fifo_full,
    output wire       tx_busy
);
```

```
  // FIFO instantiation
  wire [7:0] fifo_data;
  wire       fifo_empty, data_out_valid;
  async_fifo #(.WIDTH(8), .DEPTH(16)) fifo_inst (
    .clk_wr(clk), .clk_rd(clk), .rst(rst),
    .wr_en(data_valid), .wr_data(data_in),
    .rd_en(fifo_rd_en), .rd_data(fifo_data),
    .fifo_full(fifo_full), .fifo_empty(fifo_empty),
    .data_out_valid(data_out_valid)
  );

  // Packetizer FSM instantiation
  wire fifo_rd_en;
  wire [7:0] tx_data;
  wire tx_start;
  packetizer_fsm fsm_inst (
    .clk(clk), .rst(rst), .tx_ready(tx_ready),
    .fifo_empty(fifo_empty), .fifo_data(fifo_data),
    .data_out_valid(data_out_valid),
    .fifo_rd_en(fifo_rd_en), .tx_data(tx_data),
    .tx_start(tx_start), .tx_busy(tx_busy)
  );

  // UART transmitter instantiation
  uart_tx uart_inst (
    .clk(clk), .rst(rst), .tx_start(tx_start),
    .tx_data(tx_data), .serial_out(serial_out), .tx_busy(/* internal busy */)
  );
  // Output tx_busy from packetizer FSM or UART as system tx_busy
  // (Assume FSM drives tx_busy or connect UART busy here.)
endmodule
```

Signal routing ensures that `data_in`/`data_valid` go into the FIFO, `fifo_full` is driven from the FIFO, and the `tx_busy` output reflects when the UART is sending bits (from either FSM or UART module). This top module ties everything together so that incoming bytes are queued, packetized, and serialized.

Verilog Testbench

A comprehensive testbench applies stimulus to verify the design. The testbench should generate a 50 MHz clock and toggle `rst` at start. It pulses `data_valid` with various bytes, simulating writes to the FIFO. It also controls `tx_ready` to model the external receiver. Key checks include: - When `data_valid` pulses, the FIFO should accept data until full (`fifo_full` goes high) or data runs out. - When `fifo_empty` is false and `tx_ready`=1, the FSM should start a transmission (`tx_busy` high) and `serial_out` should show the correct start/data/stop bits for each byte. - After each transmission, `tx_busy` should return low and the FSM should proceed to the next byte.

For example:

```
module tb_uart_packetizer;
  reg clk=0, rst, data_valid=0, tx_ready=0;
  reg [7:0] data_in=0;
  wire serial_out, fifo_full, tx_busy;

  // Instantiate top-level design
  top_uart_packetizer dut (
    .clk(clk), .rst(rst), .data_in(data_in),
    .data_valid(data_valid), .tx_ready(tx_ready),
    .serial_out(serial_out), .fifo_full(fifo_full), .tx_busy(tx_busy)
  );

  // Clock generation: 50 MHz
  always #10 clk = ~clk;

  initial begin
    // Reset
    rst = 1; #50; rst = 0;
    tx_ready = 0;

    // Write two bytes into FIFO
    @(posedge clk); data_in = 8'hA5; data_valid = 1;
    @(posedge clk); data_valid = 0;
    @(posedge clk); data_in = 8'h5A; data_valid = 1;
    @(posedge clk); data_valid = 0;

    // Assert ready and observe transmission
    #100 tx_ready = 1;
    // Wait enough time for two full packets to transmit...
    #20000 $finish;
  end

  // Optional: Monitor signals
  initial begin
    $dumpfile("waveform.vcd"); $dumpvars(0, tb_uart_packetizer);
    $display("Time    fifo_full tx_busy serial_out");
    $monitor("%0dns  %b         %b      %b", $stime, fifo_full, tx_busy,
serial_out);
  end
endmodule
```

This testbench writes two specific bytes (0xA5, 0x5A) into the FIFO, then sets `tx_ready=1`. The simulation should show `fifo_full` low (unless FIFO overflows), `tx_busy` pulsing high during each 10-bit transmission, and `serial_out` toggling through the frame bits (observe an initial low start bit followed by

each data bit and a final high stop bit for both bytes). Waveform viewers (like GTKWave or ModelSim) can be used to visually inspect `serial_out`, `fifo_full`, and `tx_busy`. The state transitions can be inferred by when `tx_busy` asserts and deasserts for each packet.

Documentation and Explanation

- **FSM State Diagram:** The FSM has states *IDLE, READ_FIFO, TRANSMIT,* and *DONE*. In *IDLE*, it waits for `!fifo_empty` and `tx_ready`; *READ_FIFO* reads the byte; *TRANSMIT* sends the framed byte; *DONE* deasserts `tx_busy` and returns to *IDLE*. Transitions occur on FIFO flags and the handshake `tx_ready`. A state diagram would show transitions based on `fifo_empty`, `data_out_valid`, and completion of sending (internal counter).

- **FIFO Usage:** The FIFO buffers incoming bytes when `data_valid` pulses, storing up to 16 bytes. It decouples the input rate from the UART transmission rate. When `data_valid`=1 and `fifo_full`=0, the FIFO writes the byte. The FSM only reads (`fifo_rd_en`) when `!fifo_empty` and the transmitter is ready. The signals `fifo_full` and `fifo_empty` indicate buffer status; `data_out_valid` signals when a byte is available on the output bus. This allows the packetizer to never read from an empty FIFO and prevents writes into a full buffer, ensuring no data loss as long as reads keep up with writes on average.

- **Asynchronous FIFO Justification:** An asynchronous FIFO is used because it safely handles different clock domains or independent timing between the writer and reader [3]. Here, even with a single clock, it provides a clean interface: the producer (driving `data_valid`) can write bursts of data without waiting for the transmitter. The consumer (UART FSM) reads at the serial bit rate, which may be slower. Using gray-coded read/write pointers and synchronizers (common practice in asynchronous FIFOs) avoids metastability when clocks differ [4] [5]. Even if both sides use the 50MHz clock, organizing the logic as an asynchronous FIFO makes the design more robust and easily extensible. Moreover, the FIFO's `fifo_full` flag back-pressures the data source, and `fifo_empty` prevents reading when no data is available [3].

Overall, this design cleanly separates concerns: the FIFO handles buffering and flow control; the FSM handles packet framing logic; the UART block handles bit-level serialization. The provided Verilog modules and testbench ensure correct packet transmission with 1 start bit, 8 data bits, and 1 stop bit each time [1] [2]. All protocols (FIFO handshake and UART framing) are supported by the signals and state transitions described above.

Sources: Concepts of UART framing and buffering are standard [1] [2] [8], and asynchronous FIFO design is well-established for clock-domain crossing [3] [4].

---

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

[2] [6] [7] [8] UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter | Analog Devices
https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html

3 GitHub - dpretet/async_fifo: A dual clock asynchronous FIFO written in verilog, tested with Icarus Verilog
https://github.com/dpretet/async_fifo

4 Dual-Clock Asynchronous FIFO in SystemVerilog - Verilog Pro
https://www.verilogpro.com/asynchronous-fifo-design/

5 Asynchronous FIFO - VLSI Verify
https://vlsiverify.com/verilog/verilog-codes/asynchronous-fifo/