MAIN PROJECT

ROUTER 1x3 VERIFICATION USING UVM

SUBMITTED BY
MELVIN RIHOHN T
26/03/2025

TESTBENCH

```
`include "uvm_macros.svh" // Include UVM macros
import uvm_pkg::*; // Import UVM package

interface router_if(input logic clk, input logic rst);
    // Data input signal
    logic [7:0] d_in;

    // Packet valid signal, indicates a valid packet is present
    logic pkt_valid;

    // Read enable signals for three output ports
    logic rd_en_0, rd_en_1, rd_en_2;

    // Valid output signals indicating data availability at each output port
    logic vld_out_0, vld_out_1, vld_out_2;

    // Error signal, indicates an invalid condition or error
    logic err;

    // Busy signal, indicates the router is processing data
    logic busy;

    // Data output signals for three output ports
    logic [7:0] dout_0, dout_1, dout_2;
endinterface



// Transaction class
class transaction extends uvm_sequence_item;
    // Randomized input data and header
    rand logic [7:0] d_in; // Data input
    rand bit [7:0] header; // Header containing routing information

    // Control signals
    logic pkt_valid; // Indicates if the packet is valid
    logic rd_en_0 = 1, rd_en_1 = 1, rd_en_2 = 1; // Read enable signals for three output
channels

    // Additional signals
    logic [7:0] parity = 0; // Parity bit for error detection
    logic vld_out_0, vld_out_1, vld_out_2; // Validity signals for output channels
    logic err, busy; // Error and busy flags
    logic [7:0] dout_0, dout_1, dout_2; // Output data for three channels
```

```
    // Constraints to ensure valid header values
    constraint con1 {
        header[1:0] != 2'b11; // Avoid using header values with lower 2 bits as '11'
        header[7:2] inside {[1:30]}; // Restrict header values to a valid range
    }

    // Registering the transaction class with UVM factory
    `uvm_object_utils_begin(transaction)
        `uvm_field_int(d_in, UVM_DEFAULT);
        `uvm_field_int(header, UVM_DEFAULT);
        `uvm_field_int(parity, UVM_DEFAULT);
        `uvm_field_int(pkt_valid, UVM_DEFAULT);
        `uvm_field_int(rd_en_0, UVM_DEFAULT);
        `uvm_field_int(rd_en_1, UVM_DEFAULT);
        `uvm_field_int(rd_en_2, UVM_DEFAULT);
        `uvm_field_int(vld_out_0, UVM_DEFAULT);
        `uvm_field_int(vld_out_1, UVM_DEFAULT);
        `uvm_field_int(vld_out_2, UVM_DEFAULT);
        `uvm_field_int(err, UVM_DEFAULT);
        `uvm_field_int(busy, UVM_DEFAULT);
        `uvm_field_int(dout_0, UVM_DEFAULT);
        `uvm_field_int(dout_1, UVM_DEFAULT);
        `uvm_field_int(dout_2, UVM_DEFAULT);
    `uvm_object_utils_end

    // Constructor function
    function new(string name = "TRANSACTION");
        super.new(name);
    endfunction
endclass


class sequences extends uvm_sequence #(transaction);
    // Declare a transaction object
    transaction trans;

    // Register the sequence with the UVM factory
    `uvm_object_utils(sequences)

    // Constructor
    function new(string name = "SEQUENCES");
        super.new(name);
    endfunction

    // Main sequence task
    virtual task body();
        bit [7:0] header;  // Variable to store the header
```

```
        int count;          // Number of payload bytes
        bit [7:0] parity = 0; // Parity calculation variable

        // Send Header
        trans = transaction::type_id::create("trans"); // Create transaction object
        trans.pkt_valid = 1;  // Indicate packet is valid
        start_item(trans);
        assert(trans.randomize()); // Randomize the transaction fields
        trans.d_in = trans.header; // Assign header to data input
        header = trans.header; // Store header value for later use
        finish_item(trans);

        count = header[7:2]; // Extract the count from the header (assuming upper bits
represent count)
        parity ^= header; // Initialize parity with header value

        // Generate Payload
        repeat (count) begin
            trans = transaction::type_id::create("trans"); // Create new transaction object
            trans.pkt_valid = 1; // Keep packet valid for payload transmission
            start_item(trans);
            assert(trans.randomize()); // Randomize the transaction fields

            parity ^= trans.d_in; // Update parity with payload data
            finish_item(trans);
        end

        // Send Parity
        trans = transaction::type_id::create("trans"); // Create transaction for parity byte
        start_item(trans);
        trans.pkt_valid = 0; // Indicate end of packet
        trans.d_in = parity; // Assign computed parity value
        finish_item(trans);
    endtask
endclass


class driver extends uvm_driver #(transaction);
    // Register the driver as a UVM component
    `uvm_component_utils(driver)

    // Virtual interface handle to interact with DUT
    virtual router_if vif;

    // Transaction object to hold stimulus data
    transaction trans;

    int len = 0;    // Stores the length of the packet
```

```
int count = 0;  // Counter for tracking the number of transmitted transactions

// Constructor
function new(string name = "DRIVER", uvm_component parent = null);
    super.new(name, parent);
endfunction

// Build phase: Initialize transaction object and get virtual interface
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Create an instance of the transaction class
    trans = transaction::type_id::create("trans", this);

    // Retrieve the virtual interface from the UVM configuration database
    if (!uvm_config_db#(virtual router_if)::get(this, "", "vif", vif))
        `uvm_fatal("router_driver", "Virtual interface not set")
endfunction

// Run phase: Drive transactions to the DUT
task run_phase(uvm_phase phase);
    // Wait until the DUT is not busy
    wait(vif.busy == 0);
    @(negedge vif.clk) // Synchronize with the negative clock edge

    // Fetch the first transaction from the sequence
    seq_item_port.get_next_item(trans);

    // Drive initial packet information to the DUT
    vif.pkt_valid = trans.pkt_valid;
    vif.d_in = trans.d_in;

    // Extract packet length from the data field
    len = int'(trans.d_in[7:2] + 1);

    // Indicate that the transaction processing is done
    seq_item_port.item_done();
    @(negedge vif.clk)

    // Forever loop to drive subsequent transactions
    forever begin
        // Wait for DUT to be ready
        wait(vif.busy == 0);
        @(negedge vif.clk)

        // Fetch the next transaction from the sequence
        seq_item_port.get_next_item(trans);
```

```
            // Drive transaction data to the DUT
            vif.pkt_valid = trans.pkt_valid;
            vif.d_in = trans.d_in;
            vif.rd_en_0 = trans.rd_en_0;
            vif.rd_en_1 = trans.rd_en_1;
            vif.rd_en_2 = trans.rd_en_2;

            // Indicate that the transaction processing is done
            seq_item_port.item_done();

            // Increment the transaction counter
            count = count + 1;
        end
    endtask
endclass


// Monitor
class monitor extends uvm_monitor;
    // Register the monitor as a UVM component
    `uvm_component_utils(monitor)

    // Virtual interface handle to interact with DUT
    virtual router_if vif;

    // Transaction object to capture monitored data
    transaction trans;

    // Analysis port to send captured transactions to the scoreboard or other components
    uvm_analysis_port #(transaction) send;

    // Constructor
    function new(string name = "MONITOR", uvm_component parent = null);
        super.new(name, parent);

        // Create analysis port to send observed data
        send = new("send", this);
    endfunction

    // Build phase: Initialize transaction object and get virtual interface
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        // Create a transaction object to store captured data
        trans = transaction::type_id::create("trans");

        // Retrieve the virtual interface from the UVM configuration database
        if (!uvm_config_db#(virtual router_if)::get(this, "", "vif", vif))
```

```
            `uvm_fatal("monitor", "Virtual interface not set")
    endfunction

    // Run phase: Capture signals from the DUT continuously
    task run_phase(uvm_phase phase);
        forever begin
            @(negedge vif.clk); // Sample data on the negative clock edge

            // Capture DUT output signals into the transaction object
            trans.dout_0     = vif.dout_0;
            trans.dout_1     = vif.dout_1;
            trans.dout_2     = vif.dout_2;
            trans.vld_out_0  = vif.vld_out_0;
            trans.vld_out_1  = vif.vld_out_1;
            trans.vld_out_2  = vif.vld_out_2;
            trans.err        = vif.err;
            trans.busy       = vif.busy;
            trans.d_in       = vif.d_in;

            // Send the captured transaction to other UVM components (e.g., scoreboard)
            send.write(trans);
        end
    endtask
endclass


// Scoreboard
class scoreboard extends uvm_scoreboard;
    // Register the scoreboard as a UVM component
    `uvm_component_utils(scoreboard)

    // Queues to store input and output streams
    int in_stream[$], out_stream[$];

    bit [7:0] header = 0; // Stores the packet header
    int int_parity = 0;   // Variable to store calculated parity

    int prev_d_in = 0, prev_d_out = 0; // Variables to track previous data values

    // Transaction object to hold received data
    transaction trans;

    // Analysis implementation port to receive transactions from the monitor
    uvm_analysis_imp #(transaction, scoreboard) recv;

    // Constructor
    function new(string name = "SCOREBOARD", uvm_component parent = null);
        super.new(name, parent);
```

```
    // Initialize analysis port to receive monitored transactions
    recv = new("send", this);
endfunction

// Build phase: Initialize transaction object
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Create a transaction instance
    trans = transaction::type_id::create("trans", this);
endfunction

// Write function: Called when monitor sends a transaction
virtual function void write(input transaction t);
    trans = t;

    // Call the function to check the received packet
    check_packet(trans);
endfunction

// Function to verify packet correctness
function void check_packet(transaction pkt);
    // Capture input data if it has changed
    if(pkt.d_in != prev_d_in) begin
        in_stream.push_back(pkt.d_in);
        prev_d_in = pkt.d_in;

        // Store the first byte as the header
        if(in_stream.size() == 1) header = pkt.d_in;
    end

    // Capture valid output data from output port 0
    if(pkt.rd_en_0 && pkt.vld_out_0 && (pkt.dout_0 != 8'b0)) begin
        out_stream.push_back(pkt.dout_0);
        prev_d_out = pkt.dout_0;
        if(out_stream.size() == header[7:2] + 1) checkParity();
    end
    // Capture valid output data from output port 1
    else if(pkt.rd_en_1 && pkt.vld_out_1 && (pkt.dout_1 != 8'b0)) begin
        out_stream.push_back(pkt.dout_1);
        prev_d_out = pkt.dout_1;
        if(out_stream.size() == header[7:2] + 1) checkParity();
    end
    // Capture valid output data from output port 2
    else if(pkt.rd_en_2 && pkt.vld_out_2 && (pkt.dout_2 != 8'b0)) begin
        out_stream.push_back(pkt.dout_2);
        prev_d_out = pkt.dout_2;
```

```
            if(out_stream.size() == header[7:2] + 1) checkParity();
        end
    endfunction

    function void checkParity();
        // Compute parity over received output data
        foreach(out_stream[i]) begin
            int_parity ^= out_stream[i];
        end

        // Compare computed parity with the last received input byte (expected parity)
        if(int_parity == in_stream[$]) begin
            out_stream.push_back(int_parity);

            // Display success message
            `uvm_info("SCOREBOARD",
"/*********************************************************************/", UVM_LOW);
            `uvm_info("SCOREBOARD", "/* Successfully verified Router 1x3", UVM_LOW);
            `uvm_info("SCOREBOARD", $sformatf("/* Input:  %0p", in_stream), UVM_LOW);
            `uvm_info("SCOREBOARD", $sformatf("/* Output: %0p", out_stream), UVM_LOW);
            `uvm_info("SCOREBOARD",
"/*********************************************************************/", UVM_LOW);
        end
        else begin
            // Display error message if parity check fails
            $display("Received corrupted data");
        end

    endfunction
endclass


class agent extends uvm_agent;
    // Register the agent as a UVM component
    `uvm_component_utils(agent)

    // Sequencer to generate and control stimulus transactions
    uvm_sequencer #(transaction) sqr;

    // Driver to drive stimulus into the DUT
    driver drv;

    // Monitor to capture DUT responses
    monitor mon;

    // Constructor
    function new(string comp = "AGENT", uvm_component parent = null);
        super.new(comp, parent);
```

```
    endfunction

    // Build phase: Create instances of sequencer, driver, and monitor
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        // Instantiate sequencer, driver, and monitor
        sqr = uvm_sequencer#(transaction)::type_id::create("sqr", this);
        drv = driver::type_id::create("drv", this);
        mon = monitor::type_id::create("mon", this);
    endfunction

    // Connect phase: Connect sequencer to driver
    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);

        // Connect the sequencer's transaction output to the driver's input
        drv.seq_item_port.connect(sqr.seq_item_export);
    endfunction

endclass


// Environment
class environment extends uvm_env;
    // Register the environment as a UVM component
    `uvm_component_utils(environment)

    // Declare agent and scoreboard components
    agent agt;      // Agent handles stimulus generation and DUT interaction
    scoreboard sbd; // Scoreboard verifies DUT output against expected results

    // Constructor
    function new(string name = "ENVIRONMENT", uvm_component parent = null);
        super.new(name, parent);
    endfunction

    // Build phase: Instantiate agent and scoreboard
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        // Create instances of the agent and scoreboard
        agt = agent::type_id::create("agt", this);
        sbd = scoreboard::type_id::create("sbd", this);
    endfunction

    // Connect phase: Connect the monitor's analysis port to the scoreboard
    virtual function void connect_phase(uvm_phase phase);
```

```
        super.connect_phase(phase);

        // Link the monitor's output (from agent) to the scoreboard's input
        agt.mon.send.connect(sbd.recv);
    endfunction
endclass

// Test
class test extends uvm_test;
    // Register the test class as a UVM component
    `uvm_component_utils(test)

    // Declare environment and sequence instances
    environment env; // Environment containing agent and scoreboard
    sequences seq;   // Sequence to generate stimulus for the DUT

    // Constructor
    function new(string name = "TEST", uvm_component parent);
        super.new(name, parent);
    endfunction

    // Build phase: Create instances of the environment and sequence
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);

        // Instantiate environment and sequence
        env = environment::type_id::create("env", this);
        seq = sequences::type_id::create("seq");
    endfunction

    // Run phase: Execute the test sequence
    virtual task run_phase(uvm_phase phase);
        // Raise an objection to keep the test running
        phase.raise_objection(this);

        // Start the sequence on the agent's sequencer
        seq.start(env.agt.sqr);

        // Wait for 100 time units to allow the test to complete
        #100;

        // Drop the objection to allow the test to end
        phase.drop_objection(this);
    endtask
endclass


module tb_top;
```

```
// Declare clock and reset signals
logic clk = 0;
logic rst;

// Clock generation: Toggles every 5 time units to create a 10-time-unit period
always #5 clk = ~clk;

// Instantiate the interface (router_if) and connect it to clock and reset
router_if tb_if (clk, rst);

// Instantiate the Design Under Test (DUT)
router DUT (
    tb_if.clk,        // Clock input
    tb_if.rst,        // Reset input
    tb_if.d_in,       // Data input
    tb_if.pkt_valid,  // Packet valid signal
    tb_if.rd_en_0,    // Read enable for output 0
    tb_if.rd_en_1,    // Read enable for output 1
    tb_if.rd_en_2,    // Read enable for output 2
    tb_if.vld_out_0,  // Valid output flag for output 0
    tb_if.vld_out_1,  // Valid output flag for output 1
    tb_if.vld_out_2,  // Valid output flag for output 2
    tb_if.err,        // Error signal
    tb_if.busy,       // Busy signal
    tb_if.dout_0,     // Data output 0
    tb_if.dout_1,     // Data output 1
    tb_if.dout_2      // Data output 2
);

// Initialize and run the UVM test
initial begin
    // Set the virtual interface for UVM testbench components
    uvm_config_db#(virtual router_if)::set(null, "uvm_test_top.env.agt*", "vif", tb_if);

    // Start the test named "test"
    run_test("test");
end

// Reset sequence
initial begin
    rst = 0;  // Assert reset
    #10 rst = 1;  // De-assert reset after 10 time units
end

// Dump waveform data for debugging
initial begin
    $dumpfile("dump.vcd");  // Specify the waveform file
    $dumpvars;              // Dump all variables
```

```
        #1000 $finish;          // Stop simulation after 1000 time units
    end
endmodule
```

DESIGN (ROUTER 1x3)

ROUTER TOP

```
/*************************************************************************/
/*      AUTHOR: METECH                                                   */
/*      FILE_NAME: router.sv                                             */
/*      DESCRIPTION: Top level module of a 1x3 router                    */
/*      DATE: 21/12/2024                                                 */
/*************************************************************************/

module router (
    input logic clk,                    // Clock input
    input logic rst,                    // Reset input
    input logic [7:0] d_in,             // Data input (8 bits)
    input logic pkt_valid,              // Packet validity signal
    input logic rd_en_0,                // Read enable for FIFO 0
    input logic rd_en_1,                // Read enable for FIFO 1
    input logic rd_en_2,                // Read enable for FIFO 2
    output logic vld_out_0,             // Valid output for FIFO 0
    output logic vld_out_1,             // Valid output for FIFO 1
    output logic vld_out_2,             // Valid output for FIFO 2
    output logic err,                   // Error signal
    output logic busy,                  // Busy signal indicating processing
    output logic [7:0] dout_0,          // Data output from FIFO 0
    output logic [7:0] dout_1,          // Data output from FIFO 1
    output logic [7:0] dout_2           // Data output from FIFO 2
);

    // Internal wire declarations for FIFO control signals and state management
    logic soft_rst_0, full_0, empty_0;
    logic soft_rst_1, full_1, empty_1;
    logic soft_rst_2, full_2, empty_2;
    logic fifo_full, detect_addr, ld_state, laf_state;
    logic full_state, lfd_state, rst_int_reg;
    logic parity_done, low_pkt_valid, wr_en_reg;
```

```
    logic [2:0] wr_en;              // Write enable for the FIFOs
    logic [7:0] din;                // Data input to FIFOs

    // Instantiate FIFOs
    fifo FIFO_0 (clk, rst, soft_rst_0, wr_en[0], rd_en_0, lfd_state, din, full_0,
empty_0, dout_0);
    fifo FIFO_1 (clk, rst, soft_rst_1, wr_en[1], rd_en_1, lfd_state, din, full_1,
empty_1, dout_1);
    fifo FIFO_2 (clk, rst, soft_rst_2, wr_en[2], rd_en_2, lfd_state, din, full_2,
empty_2, dout_2);

    // Instantiate synchronizer to manage input data and FIFO states
    synchronizer SYNC (
        clk, rst, d_in[1:0], detect_addr,
        full_0, full_1, full_2,
        empty_0, empty_1, empty_2,
        wr_en_reg, rd_en_0, rd_en_1, rd_en_2,
        wr_en, fifo_full,
        vld_out_0, vld_out_1, vld_out_2,
        soft_rst_0, soft_rst_1, soft_rst_2
    );

    // Instantiate registers to store data and manage errors
    register REG_0 (
        clk, rst, pkt_valid, d_in,
        fifo_full, detect_addr,
        ld_state, laf_state, full_state, lfd_state,
        rst_int_reg, din, err,
        parity_done, low_pkt_valid
    );

    // Instantiate FSM controller to manage router states and operations
    fsm_controller FSM (
        clk, rst, pkt_valid, fifo_full,
        empty_0, empty_1, empty_2,
        soft_rst_0, soft_rst_1, soft_rst_2,
        parity_done, low_pkt_valid,
        d_in[1:0], wr_en_reg,
        detect_addr, ld_state, laf_state,
        lfd_state, full_state,
        rst_int_reg, busy
    );
```

```
endmodule
```

FSM_CONTROLLER

```
/*****************************************************************/
/*      AUTHOR: METECH                                           */
/*      FILE_NAME: fsm_controller.sv                             */
/*      DESCRIPTION:  FSM Controller module                      */
/*      DATE: 21/12/2024                                         */
/*****************************************************************/

module fsm_controller (
    input logic clk,              // Clock input
    input logic rst,              // Active-low reset signal
    input logic pkt_valid,        // Signal indicating a valid packet is present
    input logic fifo_full,        // Signal indicating FIFO is full
    input logic fifo_empty_0,     // Signal indicating FIFO 0 is empty
    input logic fifo_empty_1,     // Signal indicating FIFO 1 is empty
    input logic fifo_empty_2,     // Signal indicating FIFO 2 is empty
    input logic soft_rst_0,       // Soft reset signal for FIFO 0
    input logic soft_rst_1,       // Soft reset signal for FIFO 1
    input logic soft_rst_2,       // Soft reset signal for FIFO 2
    input logic parity_done,      // Signal indicating parity check is complete
    input logic low_pkt_valid,    // Signal indicating a low-valid packet
condition
    input logic[1:0] din,        // 2-bit input specifying the destination FIFO

    output logic wr_en_req,       // Write enable request signal
    output logic detect_addr,     // Signal to detect packet address
    output logic ld_state,        // Load data state indicator
    output logic laf_state,       // Load after full state indicator
    output logic lfd_state,       // Load first data state indicator
    output logic full_state,      // FIFO full state indicator
    output logic rst_int_reg,     // Reset internal register signal
    output logic busy             // Busy signal indicating FSM activity
);
```

```
  // State encoding for the FSM (1x3 router control)
  parameter DECODE_ADDRESS     = 3'b000; // State to decode packet address
  parameter LOAD_FIRST_DATA    = 3'b001; // State to load the first data word
  parameter LOAD_DATA          = 3'b010; // State to load subsequent data words
  parameter WAIT_TILL_EMPTY    = 3'b011; // Wait for the target FIFO to become
empty
  parameter CHECK_PARITY_ERROR = 3'b100; // State to check for parity errors
  parameter LOAD_PARITY        = 3'b101; // Load parity word
  parameter FIFO_FULL_STATE    = 3'b110; // State when FIFO is full
  parameter LOAD_AFTER_FULL    = 3'b111; // Load data after the FIFO becomes non-
full

  logic [2:0] PS, NS; // Current State (PS) and Next State (NS) registers

  // State transition logic triggered on the rising edge of the clock
  always @(posedge clk) begin
    if (!rst)
      PS <= DECODE_ADDRESS; // Reset state to DECODE_ADDRESS
    else if (soft_rst_0 || soft_rst_1 || soft_rst_2)
      PS <= DECODE_ADDRESS; // On any soft reset, transition to DECODE_ADDRESS
    else
      PS <= NS; // Transition to the next state
  end

  // Next state logic based on the current state and input conditions
  always @(*) begin
    NS = DECODE_ADDRESS; // Default next state
    case (PS)
      DECODE_ADDRESS: begin
        if ((pkt_valid && din == 0 && fifo_empty_0) ||
            (pkt_valid && din == 1 && fifo_empty_1) ||
            (pkt_valid && din == 2 && fifo_empty_2))
          NS = LOAD_FIRST_DATA; // Load first data if FIFO is empty
        else if ((pkt_valid && din == 0 && ~fifo_empty_0) ||
                 (pkt_valid && din == 1 && !fifo_empty_1) ||
                 (pkt_valid && din == 2 && !fifo_empty_2))
          NS = WAIT_TILL_EMPTY; // Wait if target FIFO is not empty
        else
          NS = DECODE_ADDRESS; // Stay in the current state
      end

      LOAD_FIRST_DATA: NS = LOAD_DATA; // Transition to LOAD_DATA state
```

```
  LOAD_DATA: begin
    if (fifo_full)
      NS = FIFO_FULL_STATE; // If FIFO is full, transition to full state
    else if (!fifo_full && !pkt_valid)
      NS = LOAD_PARITY; // If no more data, load parity
    else
      NS = LOAD_DATA; // Continue loading data
  end

  WAIT_TILL_EMPTY: begin
    if (fifo_empty_0 || fifo_empty_1 || fifo_empty_2)
      NS = LOAD_FIRST_DATA; // If any FIFO becomes empty, load first data
    else
      NS = WAIT_TILL_EMPTY; // Continue waiting
  end

  FIFO_FULL_STATE: begin
    if (!fifo_full)
      NS = LOAD_AFTER_FULL; // If FIFO is no longer full, load after full
    else
      NS = FIFO_FULL_STATE; // Stay in the full state
  end

  LOAD_AFTER_FULL: begin
    if (!parity_done && !low_pkt_valid)
      NS = LOAD_DATA; // If parity not done and valid, load data
    else if (!parity_done && low_pkt_valid)
      NS = LOAD_PARITY; // If low packet valid, load parity
    else if (parity_done)
      NS = DECODE_ADDRESS; // If parity done, decode next address
  end

  LOAD_PARITY: NS = CHECK_PARITY_ERROR; // Transition to parity check

  CHECK_PARITY_ERROR: begin
    if (fifo_full)
      NS = FIFO_FULL_STATE; // If FIFO is full, go to full state
    else
      NS = DECODE_ADDRESS; // Otherwise, decode next address
  end
endcase
```

```
  end

  // Output assignments based on the current state
  assign detect_addr = (PS == DECODE_ADDRESS); // Detect address in decode state
  assign wr_en_req = (PS == LOAD_DATA || PS == LOAD_PARITY || PS ==
LOAD_AFTER_FULL); // Write enable in specific states
  assign full_state = (PS == FIFO_FULL_STATE); // Indicate FIFO full state
  assign lfd_state = (PS == LOAD_FIRST_DATA); // Indicate load first data state
  assign busy = (PS == LOAD_FIRST_DATA || PS == LOAD_PARITY || PS ==
FIFO_FULL_STATE || PS == LOAD_AFTER_FULL || PS == WAIT_TILL_EMPTY || PS ==
CHECK_PARITY_ERROR); // Indicate FSM is busy
  assign ld_state = (PS == LOAD_DATA); // Indicate load data state
  assign laf_state = (PS == LOAD_AFTER_FULL); // Indicate load after full state
  assign rst_int_reg = (PS == CHECK_PARITY_ERROR); // Reset internal register
during parity check

endmodule
```

REGISTER

```
/////////////////////////////////////////////////////*
/*      AUTHOR: METECH                               */
/*      FILE_NAME: register.sv                        */
/*      DESCRIPTION:  register module                */
/*      DATE: 21/12/2024                             */
/////////////////////////////////////////////////////*

module register (
    input logic clk,                // Clock input
    input logic rst,                // Active-low reset signal
    input logic pkt_valid,          // Packet valid signal indicating data
validity
    input logic [7:0] din,          // 8-bit data input
    input logic fifo_full,          // Signal indicating if the FIFO is full
    input logic detect_addr,        // Signal for address detection
    input logic ld_state,           // Signal indicating load state is active
    input logic laf_state,          // Signal indicating load after full state
    input logic full_state,         // Signal indicating the full state of the
system
    input logic lfd_state,          // Signal indicating load first data state
    input logic rst_int_reg,        // Signal to reset the internal register
    output logic [7:0] dout,    // 8-bit data output
    output logic err,           // Error signal output
```

```
    output logic parity_done,   // Parity check completion flag
    output logic low_pkt_valid  // Signal indicating low packet validity
);

  // Internal registers to store data, parity, and intermediate values
  logic [7:0] header, int_reg, int_parity, ext_parity;

  // PARITY DONE LOGIC: Controls when the parity check is marked as done
  always @(posedge clk) begin
    if (!rst)
      parity_done <= 0;       // Reset parity done flag
    else if (detect_addr)
      parity_done <= 0;       // Reset if address detection occurs
    else if ((ld_state && (~fifo_full) && (~pkt_valid)) ||
             (laf_state && low_pkt_valid && (~parity_done)))
      parity_done <= 1;       // Set parity done if conditions are met
  end

  // LOW PACKET VALID LOGIC: Manages the `low_pkt_valid` flag
  always @(posedge clk) begin
    if (!rst)
      low_pkt_valid <= 0;     // Reset low packet valid flag
    else if (rst_int_reg)
      low_pkt_valid <= 0;     // Reset if internal register is reset
    else if (ld_state && ~pkt_valid)
      low_pkt_valid <= 1;     // Set if in load state and no valid packet
  end

  // DATA OUT LOGIC: Controls the data output based on various states
  always @(posedge clk) begin
    if (!rst) begin
      dout <= 0;              // Reset data output
      header <= 0;            // Reset header register
      int_reg <= 0;           // Reset internal register
    end else if (detect_addr && pkt_valid && din[1:0] != 2'b11)
      header <= din;          // Capture header if address is detected and packet
is valid
    else if (lfd_state)
      dout <= header;         // Output header if in load first data state
    else if (ld_state && ~fifo_full)
      dout <= din;            // Output data if in load state and FIFO is not
full
```

```
    else if (ld_state && fifo_full)
      int_reg <= din;         // Store data in internal register if FIFO is full
    else if (laf_state)
      dout <= int_reg;        // Output internal register data if in load after
full state
  end

  // PARITY CALCULATE LOGIC: Computes the internal parity for error checking
  always @(posedge clk) begin
    if (!rst)
      int_parity <= 0;      // Reset internal parity
    else if (detect_addr)
      int_parity <= 0;      // Reset if address detection occurs
    else if (lfd_state && pkt_valid)
      int_parity <= int_parity ^ header; // XOR with header data if packet is
valid
    else if (ld_state && pkt_valid && ~full_state)
      int_parity <= int_parity ^ din; // XOR with data input if in load state
    else
      int_parity <= int_parity; // Hold current parity value
  end

  // ERROR LOGIC: Checks if there is a parity error
  always @(posedge clk) begin
    if (!rst)
      err <= 0; // Reset error flag
    else if (parity_done) begin
      if (int_parity == ext_parity)
        err <= 0; // No error if internal and external parity match
      else
        err <= 1; // Set error if parities do not match
    end else
      err <= 0; // Hold error as 0 if parity is not done
  end

  // EXTERNAL PARITY LOGIC: Stores the external parity value
  always @(posedge clk) begin
    if (!rst)
      ext_parity <= 0; // Reset external parity
    else if (detect_addr)
      ext_parity <= 0; // Reset if address detection occurs
    else if ((ld_state && !fifo_full && !pkt_valid) ||
```

```
            (laf_state && ~parity_done && low_pkt_valid))
      ext_parity <= din; // Store data input as external parity if conditions are
met
  end
endmodule
```

FIFO

```
/*************************************************************************/
/*      AUTHOR: METECH                                                   */
/*      FILE_NAME: fifo.sv                                               */
/*      DESCRIPTION:  16x9 Fifo Module                                   */
/*      DATE: 21/12/2024                                                 */
/*************************************************************************/

module fifo (
    input logic clk,             // Clock input signal
    input logic rst,             // Active-low reset signal
    input logic soft_reset,      // Soft reset signal to clear certain outputs
    input logic wr_en,           // Write enable signal
    input logic rd_en,           // Read enable signal
    input logic lfd_state,       // State indicating the first data word (header)
    input logic [7:0] din,       // 8-bit input data to write into the FIFO
    output logic full,           // Full flag indicating the FIFO is full
    output logic empty,          // Empty flag indicating the FIFO is empty
    output logic [7:0] dout  // 8-bit output data from the FIFO
);

  logic [4:0] rd_ptr, wr_ptr;    // Read and write pointers (5 bits to track
overflow)
  logic [6:0] intCount;          // Counter for tracking multi-byte packet size
  logic [8:0] mem[15:0];         // Memory array with 9-bit width for data +
header bit                    //
  logic lfd_state_t;             // Temporary register to hold the lfd_state
signal

  // Latching lfd_state signal with each clock cycle
  always @(posedge clk) begin
    if (!rst)
      lfd_state_t <= 0;        // Reset lfd_state_t when reset is active
    else
      lfd_state_t <= lfd_state; // Store the current lfd_state value
```

```
  end

  // Managing data output based on read enable and empty status
  always @(posedge clk) begin
    if (!rst)
      dout <= 8'b0;            // Reset dout to 0 on reset
    else if (soft_reset)
      dout <= 8'bz;            // Set dout to high impedance on soft reset
    else if (rd_en && !empty)
      dout <= mem[rd_ptr[3:0]][7:0]; // Read data from memory if enabled and not
empty
    else if (intCount == 0)
      dout <= 8'bz;            // High impedance when no data to output
  end

  // Memory write logic: Writing input data into the FIFO
  always @(posedge clk) begin
    if (!rst || soft_reset) begin
      // Reset all memory locations on reset or soft reset
      for (int i = 0; i < 16; i = i + 1)
        mem[i] <= 0;
    end else if (wr_en && !full) begin
      // Write data into memory if write enabled and not full
      if (lfd_state_t) begin
        mem[wr_ptr[3:0]][8] <= 1'b1;  // Mark as header word
        mem[wr_ptr[3:0]][7:0] <= din; // Store input data
      end else begin
        mem[wr_ptr[3:0]][8] <= 1'b0;  // Mark as regular data word
        mem[wr_ptr[3:0]][7:0] <= din; // Store input data
      end
    end
  end

  // Write pointer update logic
  always @(posedge clk) begin
    if (!rst)
      wr_ptr <= 0;             // Reset write pointer
    else if (wr_en && !full)
      wr_ptr <= wr_ptr + 1;    // Increment write pointer on write enable
  end

  // Read pointer update logic
```

```
  always @(posedge clk) begin
    if (!rst)
      rd_ptr <= 0;              // Reset read pointer
    else if (rd_en && !empty)
      rd_ptr <= rd_ptr + 1;     // Increment read pointer on read enable
  end

  // Internal counter management for tracking data packets
  always @(posedge clk) begin
    if (rd_en && !empty) begin
      // If header word, initialize intCount with data size + 1
      if (mem[rd_ptr[3:0]][8] == 1'b1)
        intCount <= mem[rd_ptr[3:0]][7:2] + 1'b1;
      // Otherwise, decrement intCount if it's not zero
      else if (intCount != 0)
        intCount <= intCount - 1'b1;
    end
  end

  // Full flag: Set when write and read pointers overlap with different MSBs
  assign full = (wr_ptr == {~rd_ptr[4], rd_ptr[3:0]});

  // Empty flag: Set when write and read pointers are identical
  assign empty = (rd_ptr == wr_ptr);

endmodule
```

SYNCHRONIZER

```
/************************************************************************/
/*      AUTHOR: METECH                                                  */
/*      FILE_NAME: synchronizer.sv                                      */
/*      DESCRIPTION:  Synchronizer Module                               */
/*      DATE: 21/12/2024                                                */
/************************************************************************/

module synchronizer (
    input logic clk,              // Clock input signal
    input logic rst,              // Active-low reset signal
    input logic[1:0] din,        // 2-bit input data to determine FIFO selection
    input logic detect_addr,      // Signal indicating if address detection is
active
```

```
    input logic full_0,           // Signal indicating if FIFO 0 is full
    input logic full_1,           // Signal indicating if FIFO 1 is full
    input logic full_2,           // Signal indicating if FIFO 2 is full
    input logic empty_0,          // Signal indicating if FIFO 0 is empty
    input logic empty_1,          // Signal indicating if FIFO 1 is empty
    input logic empty_2,          // Signal indicating if FIFO 2 is empty
    input logic wr_en_reg,        // Write enable register input
    input logic rd_en_0,          // Read enable signal for FIFO 0
    input logic rd_en_1,          // Read enable signal for FIFO 1
    input logic rd_en_2,          // Read enable signal for FIFO 2
    output logic [2:0] wr_en, // 3-bit output to control write enable for each
FIFO
    output logic fifo_full,   // Output indicating if the selected FIFO is full
    output logic vld_out_0,       // Valid output signal for FIFO 0 (not empty)
    output logic vld_out_1,       // Valid output signal for FIFO 1 (not empty)
    output logic vld_out_2,       // Valid output signal for FIFO 2 (not empty)
    output logic soft_reset_0,// Soft reset signal for FIFO 0
    output logic soft_reset_1,// Soft reset signal for FIFO 1
    output logic soft_reset_2 // Soft reset signal for FIFO 2
);

  // Registers to count cycles for each FIFO's inactivity
  logic [5:0] count0, count1, count2;
  logic [1:0] tmp_din; // Temporary register to hold the address input

  // Capture 'din' value on the detection of address
  always @(posedge clk) begin
    if (!rst)
      tmp_din <= 0; // Reset 'tmp_din' to 0 when reset is active
    else if (detect_addr)
      tmp_din <= din; // Store 'din' if address detection is active
  end

  // Control logic for write enable and FIFO full status based on 'tmp_din'
  always @(*) begin
    case (tmp_din)
      2'b00: begin // Case for FIFO 0
        fifo_full <= full_0; // Set full status for FIFO 0
        wr_en <= (wr_en_reg) ? 3'b001 : 0; // Enable write if 'wr_en_reg' is set
      end
      2'b01: begin // Case for FIFO 1
        fifo_full <= full_1; // Set full status for FIFO 1
```

```
      wr_en <= (wr_en_reg) ? 3'b010 : 0; // Enable write if 'wr_en_reg' is set
    end
    2'b10: begin // Case for FIFO 2
      fifo_full <= full_2; // Set full status for FIFO 2
      wr_en <= (wr_en_reg) ? 3'b100 : 0; // Enable write if 'wr_en_reg' is set
    end
    default: begin // Default case: no FIFO selected
      fifo_full <= 0;
      wr_en <= 0;
    end
  endcase
end

// Assign valid outputs based on FIFO emptiness
assign vld_out_0 = (~empty_0); // Valid if FIFO 0 is not empty
assign vld_out_1 = (~empty_1); // Valid if FIFO 1 is not empty
assign vld_out_2 = (~empty_2); // Valid if FIFO 2 is not empty

// Monitor FIFO 0 for inactivity and trigger soft reset if needed
always @(posedge clk) begin
  if (!rst) begin
    count0 <= 0;
    soft_reset_0 <= 0; // Reset state for FIFO 0
  end else if (vld_out_0) begin // If FIFO 0 has valid data
    if (!rd_en_0) begin // If not being read
      if (count0 == 29) begin // After 30 cycles
        soft_reset_0 <= 1; // Trigger soft reset
        count0 <= 0; // Reset counter
      end else begin
        soft_reset_0 <= 0;
        count0 <= count0 + 1; // Increment counter
      end
    end else
      count0 <= 0; // Reset counter if being read
  end
end

// Monitor FIFO 1 for inactivity and trigger soft reset if needed
always @(posedge clk) begin
  if (!rst) begin
    count1 <= 0;
    soft_reset_1 <= 0; // Reset state for FIFO 1
```

```
    end else if (vld_out_1) begin // If FIFO 1 has valid data
      if (!rd_en_1) begin // If not being read
        if (count1 == 29) begin // After 30 cycles
          soft_reset_1 <= 1; // Trigger soft reset
          count1 <= 0; // Reset counter
        end else begin
          soft_reset_1 <= 0;
          count1 <= count1 + 1; // Increment counter
        end
      end else
        count1 <= 0; // Reset counter if being read
    end
  end

  // Monitor FIFO 2 for inactivity and trigger soft reset if needed
  always @(posedge clk) begin
    if (!rst) begin
      count2 <= 0;
      soft_reset_2 <= 0; // Reset state for FIFO 2
    end else if (vld_out_2) begin // If FIFO 2 has valid data
      if (!rd_en_2) begin // If not being read
        if (count2 == 29) begin // After 30 cycles
          soft_reset_2 <= 1; // Trigger soft reset
          count2 <= 0; // Reset counter
        end else begin
          soft_reset_2 <= 0;
          count2 <= count2 + 1; // Increment counter
        end
      end else
        count2 <= 0; // Reset counter if being read
    end
  end
endmodule
```

OUTPUT

```
UVM_INFO testbench.sv(352) @ 180: uvm_test_top.env.sbd [SCOREBOARD] /*********************************************************************************/
UVM_INFO testbench.sv(353) @ 180: uvm_test_top.env.sbd [SCOREBOARD] /* Successfully verified Router 1x3
UVM_INFO testbench.sv(354) @ 180: uvm_test_top.env.sbd [SCOREBOARD] /* Input:  '{53, 32, 63, 95, 10, 193, 1, 215, 88, 218, 132, 5, 135, 172, 64}
UVM_INFO testbench.sv(355) @ 180: uvm_test_top.env.sbd [SCOREBOARD] /* Output: '{53, 32, 63, 95, 10, 193, 1, 215, 88, 218, 132, 5, 135, 172, 64}
UVM_INFO testbench.sv(356) @ 180: uvm_test_top.env.sbd [SCOREBOARD] /*********************************************************************************/
```