MAIN PROJECT

ROUTER 1x3 IN SYSTEM VERILOG

SUBMITTED BY

MELVIN RIJOHN T

04/02/2025

Table of Contents

TESTBENCH FILES

TRANSACTION

```
/**********************************************************************/
/*      AUTHOR: METECH                                                */
/*      FILE_NAME: transaction.sv                                     */
/*      DESCRIPTION: Formatting data packet                           */
/*      DATE: 03/02/2025                                              */
/**********************************************************************/

// Packet type enumeration for defining packet states
typedef enum logic[1:0]{RESET = 0, HEADER = 1, PAYLOAD = 2, PARITY = 3}
pkt_type_t;

// Packet class definition
class Packet;
    // Randomizable fields representing packet attributes
    rand bit[7:0] header;
    rand bit[7:0] data;
    rand bit pkt_valid;
    randc bit rd_en_0;
    randc bit rd_en_1;
    randc bit rd_en_2;

    // Status signals
    bit vld_out_0;
    bit vld_out_1;
    bit vld_out_2;
    bit err;
    bit busy;
    bit rst;

    // Data output signals
    bit [7:0] dout_0;
    bit [7:0] dout_1;
    bit [7:0] dout_2;

    // Parity check logic
    logic[7:0] parity;

    // Packet type enumeration variable
    pkt_type_t pkt_type;

    // Constraints to ensure valid header values
    constraint con1 {
        header[1:0] != 2'b11;
        header[7:2] != 0;
        header[7:2] <= 20;
    }

    // Function to copy data from another Packet instance
    function void copy(Packet tmp);
        data = tmp.data;
        pkt_valid = tmp.pkt_valid;
        rd_en_0 = tmp.rd_en_0;
        rd_en_1 = tmp.rd_en_1;
        rd_en_2 = tmp.rd_en_2;
        vld_out_0 = tmp.vld_out_0;
        vld_out_1 = tmp.vld_out_1;
        vld_out_2 = tmp.vld_out_2;
        err = tmp.err;
        busy = tmp.busy;
        dout_0 = tmp.dout_0;
        dout_1 = tmp.dout_1;
        dout_2 = tmp.dout_2;
        parity = tmp.parity;
    endfunction

endclass
```

GENERATOR

```
/************************************************************************/
/*      AUTHOR: METECH                                                  */
/*      FILE_NAME: generator.sv                                         */
/*      DESCRIPTION: Generates input streams                            */
/*      DATE: 03/02/2025                                                */
/************************************************************************/

class Generator;
    mailbox #(Packet) mbx; // Mailbox for communication with driver
    event drv_done; // Event to synchronize with driver
    Packet pkt; // Packet instance
    bit[7:0] header; // Header field storage

    function new(mailbox #(Packet) mbx, event drv_done);
        this.mbx = mbx; // Initialize mailbox
        this.drv_done = drv_done; // Initialize event
    endfunction

    task run(int loopCount = 1);
        repeat(loopCount) begin
            pkt = new(); // Create a new packet instance
            pkt.parity = 0; // Initialize parity bit
            $display("[%0tps] Generator: Starting....", $time);

            pkt.pkt_type = RESET; // Send reset packet
            mbx.put(pkt);
            @(drv_done); // Wait for driver to complete

            if(!pkt.randomize()) $error("Randomization failed"); //
Randomize packet fields
            pkt.pkt_type = HEADER; // Set packet type to HEADER
            header = pkt.header; // Store header value
            pkt.parity = pkt.parity ^ pkt.header; // Compute parity
            mbx.put(pkt); // Send header packet
            @(drv_done);

            for (int i = 0; i < header[7:2]; i++) begin // Loop through
payload data
                if(!pkt.randomize()) $error("Randomization failed"); //
Randomize payload data
                pkt.parity = pkt.parity ^ pkt.data; // Update parity
                pkt.pkt_type = PAYLOAD; // Set packet type to PAYLOAD
                mbx.put(pkt); // Send payload packet
                @(drv_done);
            end
            pkt.pkt_type = PARITY; // Set packet type to PARITY
            pkt.data = pkt.parity; // Store computed parity in data field
            mbx.put(pkt); // Send parity packet
        end
    endtask
endclass
```

DRIVER

```
/**********************************************************************/
/*      AUTHOR: METECH                                                */
/*      FILE_NAME: driver.sv                                          */
/*      DESCRIPTION: Drives the input streams to the dut              */
/*      DATE: 03/02/2025                                              */
/**********************************************************************/

class Driver;
    local bit[7:0] header; // Stores packet header data
    mailbox #(Packet) mbx; // Mailbox for communication with other
components
    event drv_done; // Event to signal when driving is done
    virtual router_if vif; // Virtual interface for DUT interaction

    function new(mailbox #(Packet) mbx, event drv_done, virtual router_if
vif);
        this.mbx = mbx; // Initialize mailbox
        this.drv_done = drv_done; // Initialize event
        this.vif = vif; // Initialize virtual interface
    endfunction

    task run();
        $display("[%0tps] Driver: Starting...", $time);

        forever begin
            Packet pkt = new(); // Create new packet instance
            mbx.get(pkt); // Retrieve packet from mailbox
            drive(pkt); // Drive packet data to DUT
            vif.rd_en_0 = pkt.rd_en_0; // Set read enable signals
            vif.rd_en_1 = pkt.rd_en_1;
            vif.rd_en_2 = pkt.rd_en_2;
            ->drv_done; // Signal driver completion
        end
    endtask

    task drive(Packet pkt);
        case (pkt.pkt_type)
            RESET: reset_dut(); // Handle reset packet
            HEADER: drive_header(pkt); // Handle header packet
            PAYLOAD: drive_payload(pkt); // Handle payload packet
            PARITY: drive_parity(pkt); // Handle parity packet
            default: $display("Invalid packet type"); // Handle invalid
packet types
        endcase
    endtask

    task reset_dut();
        vif.rst = 0; // Deassert reset
        @(posedge vif.clk);
        vif.rst = 1; // Assert reset
        vif.pkt_valid = 0; // Deassert packet valid
        @(posedge vif.clk);
    endtask

    task drive_header(Packet pkt);
        wait(vif.busy == 0); // Wait until DUT is not busy
        @(negedge vif.clk);
        vif.pkt_valid = 1; // Assert packet valid
        vif.data = pkt.header; // Drive header data
        @(posedge vif.clk);
        @(posedge vif.clk);
    endtask

    task drive_payload(Packet pkt);
        wait(vif.busy == 0); // Wait until DUT is not busy
        @(negedge vif.clk);
        vif.pkt_valid <= 1; // Assert packet valid
        vif.data <= pkt.data; // Drive payload data
    endtask

    task drive_parity(Packet pkt);
        wait(vif.busy == 0); // Wait until DUT is not busy
        @(negedge vif.clk);
        vif.pkt_valid <= 0; // Deassert packet valid
        vif.data <= pkt.parity; // Drive parity data
    endtask
endclass
```

INTERFACE

```
/**********************************************************************/
/*      AUTHOR: METECH                                               */
/*      FILE_NAME: interface.sv                                      */
/*      DESCRIPTION: Actual interface definition                     */
/*      DATE: 03/02/2025                                             */
/**********************************************************************/

interface router_if();
    logic clk;          // Clock signal
    logic rst;          // Reset signal
    logic [7:0] data;   // Data input
    logic pkt_valid;    // Packet valid signal
    logic rd_en_0;      // Read enable signal for output 0
    logic rd_en_1;      // Read enable signal for output 1
    logic rd_en_2;      // Read enable signal for output 2
    logic vld_out_0;    // Valid output signal for output 0
    logic vld_out_1;    // Valid output signal for output 1
    logic vld_out_2;    // Valid output signal for output 2
    logic err;          // Error signal
    logic busy;         // Busy signal indicating router activity
    logic [7:0] dout_0; // Data output for output 0
    logic [7:0] dout_1; // Data output for output 1
    logic [7:0] dout_2; // Data output for output 2

    initial clk = 0; // Initialize clock to 0
    always #5 clk = ~clk; // Clock toggles every 5 time units

endinterface
```

MONITOR

```
/**********************************************************************/
/*      AUTHOR: METECH                                               */
/*      FILE_NAME: monitor.sv                                        */
/*      DESCRIPTION: Mointors the output from the dut                */
/*      DATE: 03/02/2025                                             */
/**********************************************************************/

class Monitor;
    local bit[7:0] header = 0; // Stores the header value
    mailbox #(Packet) mbx_in; // Mailbox for storing incoming packets
    mailbox #(Packet) mbx_out; // Mailbox for storing outgoing packets
    event drv_done; // Event to synchronize with the driver
    virtual router_if vif; // Virtual interface for communication with
DUT
    int count = 0; // Counter for incoming packets
    int prev_val = 0; // Stores previous output value to detect changes

    function new(mailbox #(Packet) mbx_in, mailbox #(Packet) mbx_out,
event drv_done, virtual router_if vif);
        this.mbx_in = mbx_in; // Initialize input mailbox
        this.mbx_out = mbx_out; // Initialize output mailbox
        this.drv_done = drv_done; // Initialize event
        this.vif = vif; // Initialize virtual interface
    endfunction

    task run();
        $display("[%0tps] Monitor: Starting...", $time);

        forever begin
            checkPacket_in(header); // Check incoming packets
        end
    endtask

    task run1();
        checkPacket_out(); // Check outgoing packets
    endtask

    task checkPacket_in(ref bit[7:0] header);
        Packet item = new(); // Create a new packet instance
        @(drv_done); // Wait for driver completion
        @(posedge vif.clk);
        #1;
        item = parsePacket(); // Parse packet data from interface

        if(!header && item.pkt_valid) begin
            item.pkt_type = HEADER; // Identify header packet
            header = item.data;
        end
        else if(header && item.pkt_valid) begin
            item.pkt_type = PAYLOAD; // Identify payload packet
        end
        else if(header && !item.pkt_valid) begin
            item.pkt_type = PARITY; // Identify parity packet
        end
        else begin
            item.pkt_type = RESET; // Identify reset packet
        end

        mbx_in.put(item); // Store packet in input mailbox
        count = count + 1; // Increment packet count
    endtask

    task checkPacket_out();
        int count_1 = 0;
        Packet item = new(); // Create a new packet instance

        while(count_1 < header[7:2] + 1) begin // Process expected number
of packets
            @(posedge vif.clk);
            #1;
            wait(vif.rd_en_0 || vif.rd_en_1 || vif.rd_en_2); // Wait for
read enable signals

            item = parsePacket(); // Parse packet data from interface

            if(item.rd_en_0 && (item.dout_0 != 0) && (prev_val !=
item.dout_0)) begin
                mbx_out.put(item); // Store packet in output mailbox
                count_1 = count_1 + 1; // Increment processed packet
count
                prev_val = item.dout_0; // Update previous value
            end
            else if(item.rd_en_1 && (item.dout_1 != 0) && (prev_val !=
item.dout_1)) begin
                mbx_out.put(item);
                count_1 = count_1 + 1;
                prev_val = item.dout_1;
            end
            else if (item.rd_en_2 && (item.dout_2 != 0) && (prev_val !=
item.dout_2)) begin
                mbx_out.put(item);
                count_1 = count_1 + 1;
                prev_val = item.dout_2;
            end else begin
                count_1 = count_1; // Maintain count if no new data
            end
        end
    endtask

    function Packet parsePacket();
        Packet pkt = new(); // Create a new packet instance
        pkt.rst = vif.rst;
        pkt.pkt_valid = vif.pkt_valid;
        pkt.data = vif.data;
        pkt.rd_en_0 = vif.rd_en_0;
        pkt.rd_en_1 = vif.rd_en_1;
        pkt.rd_en_2 = vif.rd_en_2;
        pkt.vld_out_0 = vif.vld_out_0;
        pkt.vld_out_1 = vif.vld_out_1;
        pkt.vld_out_2 = vif.vld_out_2;
        pkt.err = vif.err;
        pkt.busy = vif.busy;
        pkt.dout_0 = vif.dout_0;
        pkt.dout_1 = vif.dout_1;
        pkt.dout_2 = vif.dout_2;
        return pkt; // Return parsed packet
    endfunction

endclass
```

SCOREBOARD

```
/*********************************************************************/
/*   AUTHOR: METECH                                                  */
/*   FILE_NAME: scoreboard.sv                                        */
/*   DESCRIPTION: Verifies design using the received output and golden reference */
/*   DATE: 03/02/2025                                                */
/*********************************************************************/

class Scoreboard;
    bit[7:0] header = 0; // Stores the packet header
    mailbox #(Packet) mbx_in; // Mailbox for input packets
    mailbox #(Packet) mbx_out; // Mailbox for output packets
    virtual router_if vif; // Virtual interface for router
    bit[7:0] in_stream[$], out_stream[$]; // Queues to store input and
output data streams
    logic TX_done = 0; // Transmission done flag
    logic RX_done = 0; // Reception done flag
    int prev_val = 0; // Stores previous value to prevent duplicates
```

```
// Constructor: Initializes mailboxes
function new(mailbox #(Packet) mbx_in, mailbox #(Packet) mbx_out);
    this.mbx_in = mbx_in;
    this.mbx_out = mbx_out;
endfunction

// Task to process incoming packets
task in_run();
    $display("[%0tps] Scoreboard: Starting...", $time);
    forever begin
        Packet pkt;
        if(mbx_in.num() > 0) begin
            mbx_in.get(pkt);
            checkPacket(pkt, this.header);
        end else begin
            #10; // Wait to avoid busy-waiting
        end
    end
endtask

// Task to process outgoing packets
task out_run();
    int count = 1;
    forever begin
        Packet pkt;
        if (mbx_out.num() > 0) begin
            mbx_out.get(pkt);
            checkPacket_1(pkt);
            checkall();
            if (!(count > header[7:2] + 1)) begin
               count = count + 1;
            end
        end else begin
            #10; // Prevent busy-waiting
        end
    end
endtask

// Task to check incoming packets and store them
task checkPacket(Packet item, ref bit[7:0] header);
    bit[8:0] cnt;
    if(item.pkt_valid && item.rst) begin
        if(item.pkt_type == HEADER) begin
```

```
                header = item.data;
                cnt = header[7:2] + 2;
                in_stream.push_back(header);
            end
            if(item.pkt_type == PAYLOAD || item.pkt_type == PARITY) begin
                in_stream.push_back(item.data);
            end
        end
    endtask

    // Task to check outgoing packets and store them
    task checkPacket_1(Packet item);
        if(item.rd_en_0 && (item.dout_0 != 0) && (prev_val !=
item.dout_0)) begin
            out_stream.push_back(item.dout_0);
            prev_val = item.dout_0;
        end
        else if(item.rd_en_1 && (item.dout_1 != 0) && (prev_val !=
item.dout_1)) begin
            out_stream.push_back(item.dout_1);
            prev_val = item.dout_1;
        end
        else if(item.rd_en_2 && (item.dout_2 != 0) && (prev_val !=
item.dout_2)) begin
            out_stream.push_back(item.dout_2);
            prev_val = item.dout_2;
        end
    endtask

    // Task to compare input and output streams for verification
    task checkall();
        int in_parity = 0;
        int out_parity = 0;
        if((in_stream.size() == header[7:2] + 1) && (out_stream.size() ==
header[7:2] + 1)) begin

            foreach(in_stream[i]) begin
                in_parity = in_parity ^ in_stream[i];
            end
            in_stream.push_back(in_parity);

            foreach(out_stream[i]) begin
                out_parity = out_parity ^ out_stream[i];
            end
            out_stream.push_back(out_parity);

            if(in_parity == out_parity) begin

$display("/**********************************************************************************************/");
                $display("/* Successfully verified Router 1x3");
                $display("/* Input: %0p", in_stream);
                $display("/* Output: %0p", out_stream);

$display("/**********************************************************************************************/");
            end
            else begin
                $display("Verification unsuccessful: in_parity = %0h,
out_parity = %0h", in_parity, out_parity);
            end
        end
    endtask
endclass
```

ENVIRONMENT

```
/**********************************************************************/
/*      AUTHOR: METECH                                                */
/*      FILE_NAME: environment.sv                                     */
/*      DESCRIPTION: Connects driver, generator, monitor &scoreboard   */
/*      DATE: 03/02/2025                                              */
/**********************************************************************/

`include "transaction.sv"
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "scoreboard.sv"

// Environment class to instantiate and connect all verification
components
class Environment;
    Generator gen; // Generates test packets
    Driver drv; // Drives packets to DUT
    Monitor mon; // Monitors DUT output
    Scoreboard sbd; // Compares expected vs actual results

    mailbox #(Packet) drv_mbx; // Mailbox for driver communication
    mailbox #(Packet) sbd_mbx_in; // Mailbox for input packets to
scoreboard
    mailbox #(Packet) sbd_mbx_out; // Mailbox for output packets from
scoreboard
    event drv_done; // Event to synchronize driver completion
    virtual router_if vif; // Virtual interface for DUT interaction

    function new(virtual router_if vif);
        drv_mbx = new(); // Initialize driver mailbox
        sbd_mbx_in = new(); // Initialize input mailbox for scoreboard
        sbd_mbx_out = new(); // Initialize output mailbox for scoreboard
        gen = new(drv_mbx, drv_done); // Instantiate generator
        drv = new(drv_mbx, drv_done, vif); // Instantiate driver
        mon = new(sbd_mbx_in, sbd_mbx_out, drv_done, vif); // Instantiate
monitor
        sbd = new(sbd_mbx_in, sbd_mbx_out); // Instantiate scoreboard
    endfunction

    task run();
        fork
            gen.run(); // Run generator
            drv.run(); // Run driver
            mon.run(); // Run monitor
            mon.run1(); // Additional monitor function
            sbd.in_run(); // Run input side of scoreboard
            sbd.out_run(); // Run output side of scoreboard
        join
    endtask
endclass
```

TESTBENCH TOP

```
/**********************************************************************/
/*      AUTHOR: METECH                                                */
/*      FILE_NAME: testbench.sv                                       */
/*      DESCRIPTION: Testbench top module                             */
/*      DATE: 03/02/2025                                              */
/**********************************************************************/
```

```
`include "interface.sv"
`include "environment.sv"

module tb();
    router_if intf(); // Instantiate the router interface
    Environment env = new(intf); // Create environment instance with the
interface

    router dut(intf.clk, intf.rst, intf.data, intf.pkt_valid,
intf.rd_en_0, intf.rd_en_1, intf.rd_en_2, intf.vld_out_0, intf.vld_out_1,
intf.vld_out_2, intf.err, intf.busy, intf.dout_0, intf.dout_1,
intf.dout_2); // Instantiate the router DUT
    initial begin
        env.run(); // Execute testbench environment
    end
    initial begin
        $dumpfile("out.vcd"); // Specify the VCD file for waveform
dumping
        $dumpvars(1); // Dump all variables for debugging
        #2000 $finish; // Terminate simulation after 2000 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
);
```

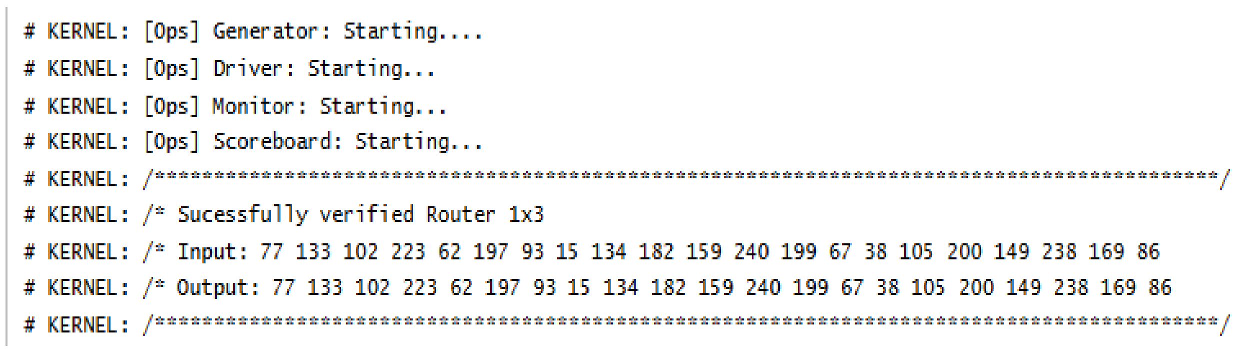
```
  logic [4:0] rd_ptr, wr_ptr;    // Read and write pointers (5 bits to
track overflow)
  logic [6:0] intCount;          // Counter for tracking multi-byte
packet size
  logic [8:0] mem[15:0];         // Memory array with 9-bit width for
data + header bit                   //
  logic lfd_state_t;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



```
# KERNEL: [Ops] Generator: Starting....
# KERNEL: [Ops] Driver: Starting...
# KERNEL: [Ops] Monitor: Starting...
# KERNEL: [Ops] Scoreboard: Starting...
# KERNEL: /******************************************************************************************/
# KERNEL: /* Sucessfully verified Router 1x3
# KERNEL: /* Input: 77 133 102 223 62 197 93 15 134 182 159 240 199 67 38 105 200 149 238 169 86
# KERNEL: /* Output: 77 133 102 223 62 197 93 15 134 182 159 240 199 67 38 105 200 149 238 169 86
# KERNEL: /******************************************************************************************/
```