**MAIN PROJECT**

**ROUTER 1x3 IN SYSTEM VERILOG**

**SUBMITTED BY**

**MELVIN RIJOHN T**

**03/02/2025**

TESTBENCH FILES 2

# 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

