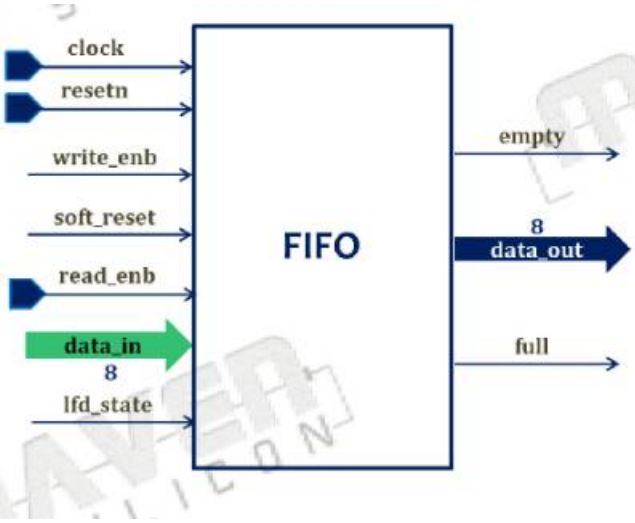
Router 1X3 Design

Table of Contents

Router-FIFO:

Block diagram:



Functionality:

There are 3 FIFOs used in the router design. Each FIFO is of 9 bits width and of 16 locations as depth.The FIFO works on the system clock and is reset with a synchronous active low reset. The FIFO is also internally reset by an internal reset signal **soft_reset** .**soft_reset** is an active high signal which is generated by the SYNCHRONIZER block during the time out state of the ROUTER.

If **resetn** is low then full =0, empty = 1 and data_out = 0.

The FIFO memory size is 16X 9. The extra bit in the data width is appended in order to detect the header byte. The **lfd_state** detects the header byte of a packet.The 9th bit is 1 for header byte and 0 for remaining bytes.

Write Operation:

- Signal **data_in** is sampled at the rising edge of the clock when **write_enb** is high.
- Write operation only takes place when FIFO is not full in order to avoid over_run condition.

Read operation:

- The data is read from **data_out** at rising edge of the clock, when **read_enb** is high.
- Read operation only takes place when FIFO is not empty in order to avoid under_run condition.
- During the read operation when a header byte is read, an internal counter is loaded with the payload length of the packet plus "1" (Parity byte) and starts decrementing every clock cycle till it reaches 0. The counter holds 0 till it is reloaded back with a new packet payload length.

- During the time out state, full = 0, empty = 1.
- **data_out** is driven to HIGH impedance state under 2 scenarios :
  - When the FIFO memory is read completely (Header+Payload+Parity).
  - Under the time out state of the ROUTER.

full – FIFO status which indicates that all the locations inside FIFO have been written.

empty – FIFO status which indicates that all the locations of FIFO have been read and made empty.

Read and Write operation can be done simultaneously.

RTL:

```
module router_fifo(clk,resetn,soft_rst,datain,dout,we,re,empty,full,lfd_state);

input clk,resetn,soft_rst,we,re,lfd_state;
input [7:0] datain;
output reg [7:0] dout;
output reg full,empty;
reg [3:0] rd_pter,wr_pter;
reg[4:0] counter;
reg [5:0] payload_len;
reg temp;
integer i;

reg [8:0] mem [15:0];

//lfd_state
always@(posedge clk)
begin
    if(!resetn)
        temp<=1'b0;
    else
        temp<=lfd_state;
    end

//full and empty logic
always @(counter)
begin

    empty = (counter==0);
    full = (counter==16);

end
 //counter logic
 always @(posedge clk)
begin
     if(!resetn)
         counter <=0;
     else if((!full && we) && (!empty && re))
         counter <= counter;
     else if(!full && we)
         counter <= counter+1;
     else if(!empty && re)
```

```
        counter <= counter-1;
    else
        counter <= counter;
end

//FIFO READ logic
always @(posedge clk)
begin
    if(!resetn)
        dout<=0;
    else if (soft_rst)
        dout <= 'bz;
    else
    begin
        if(re && !empty)
            dout <= mem[rd_pter];
        else if (payload_len==0)
            dout <= 'bz;
        else
            dout <= dout;
    end
end

//Fifo write logic
always @(posedge clk)
begin
    if (!resetn || soft_rst)
    begin
        for(i=0;i<16;i=i+1)
        mem[i]<=0;
    end
    else if(we && !full)
    begin
        mem[wr_pter] <= {lfd_state,datain};
    end
    else
        mem[wr_pter] <= mem[wr_pter];
end

//payload_len logic
always@(posedge clk)
begin
    if(!resetn || soft_rst)
        payload_len <= 0;
    if(re && !empty)
    begin
        if(mem[rd_pter[3:0]][8])
            payload_len<=mem[rd_pter[3:0]][7:2]+1'b1;
        else if(payload_len!=0)
            payload_len<=payload_len-1'b1;

    end

end
```

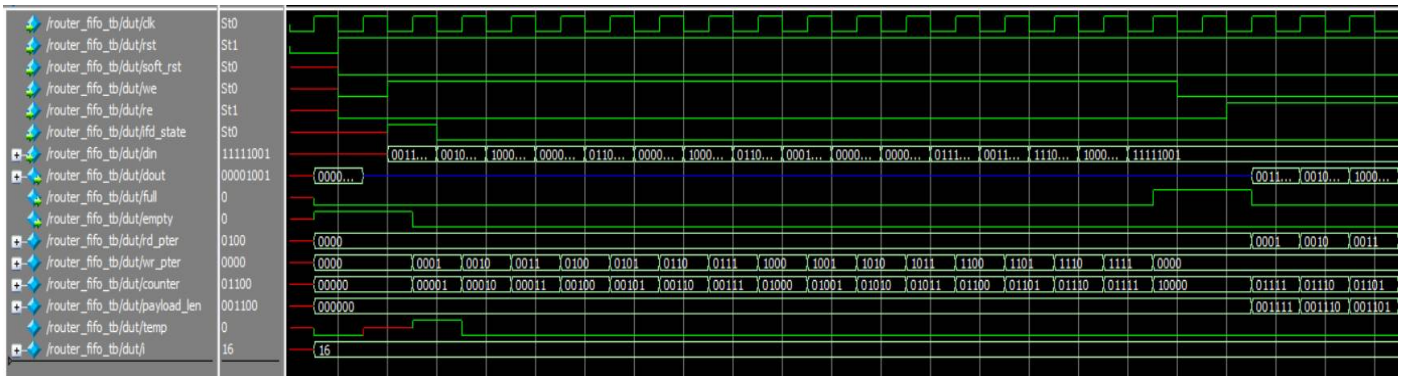
```
//pointer logic
always @(posedge clk)
begin
    if (!resetn)
    begin
        wr_pter <= 0;
        rd_pter <= 0;
    end
    else
    begin
    if (!full && we)
        wr_pter <= wr_pter +1;
    else
        wr_pter <= wr_pter;
    if (!empty && re)
        rd_pter <= rd_pter +1;
    else
        rd_pter <= rd_pter;
    end
end
endmodule
```

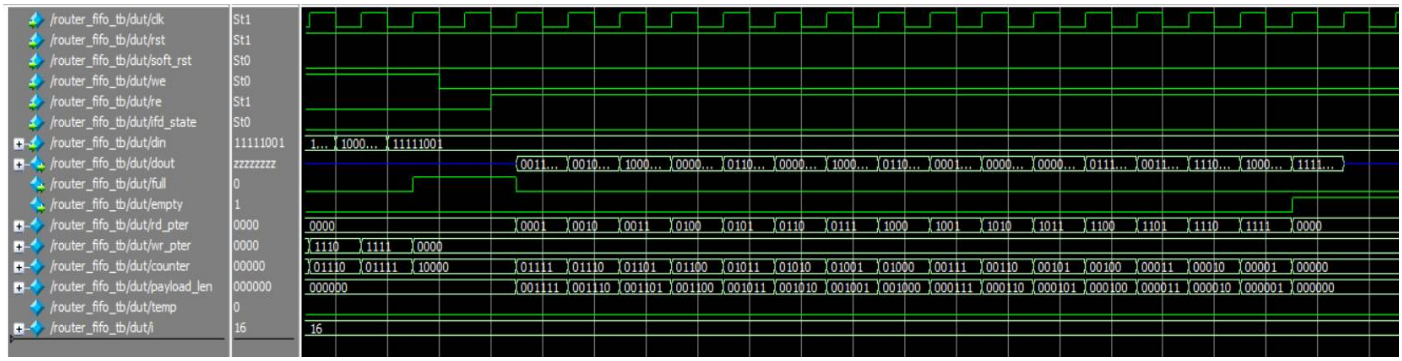
Test Bench:

```
module router_fifo_tb();

reg clk,rst,soft_rst,we,re,ifd_state;
reg [7:0] din;
wire [7:0] dout;
wire full,empty;
integer l;

router_fifo dut(clk,rst,soft_rst,din,dout,we,re,empty,full,ifd_state);

initial
begin
    clk = 1'b0;
    forever
    #5 clk = ~clk;
end

task initialize();
begin
    we=1'b0;
    re=1'b0;
    soft_rst=0;

end
endtask

task reset();
begin
    rst = 1'b0;
    @(negedge clk)
    rst = 1'b1;
end
endtask
```

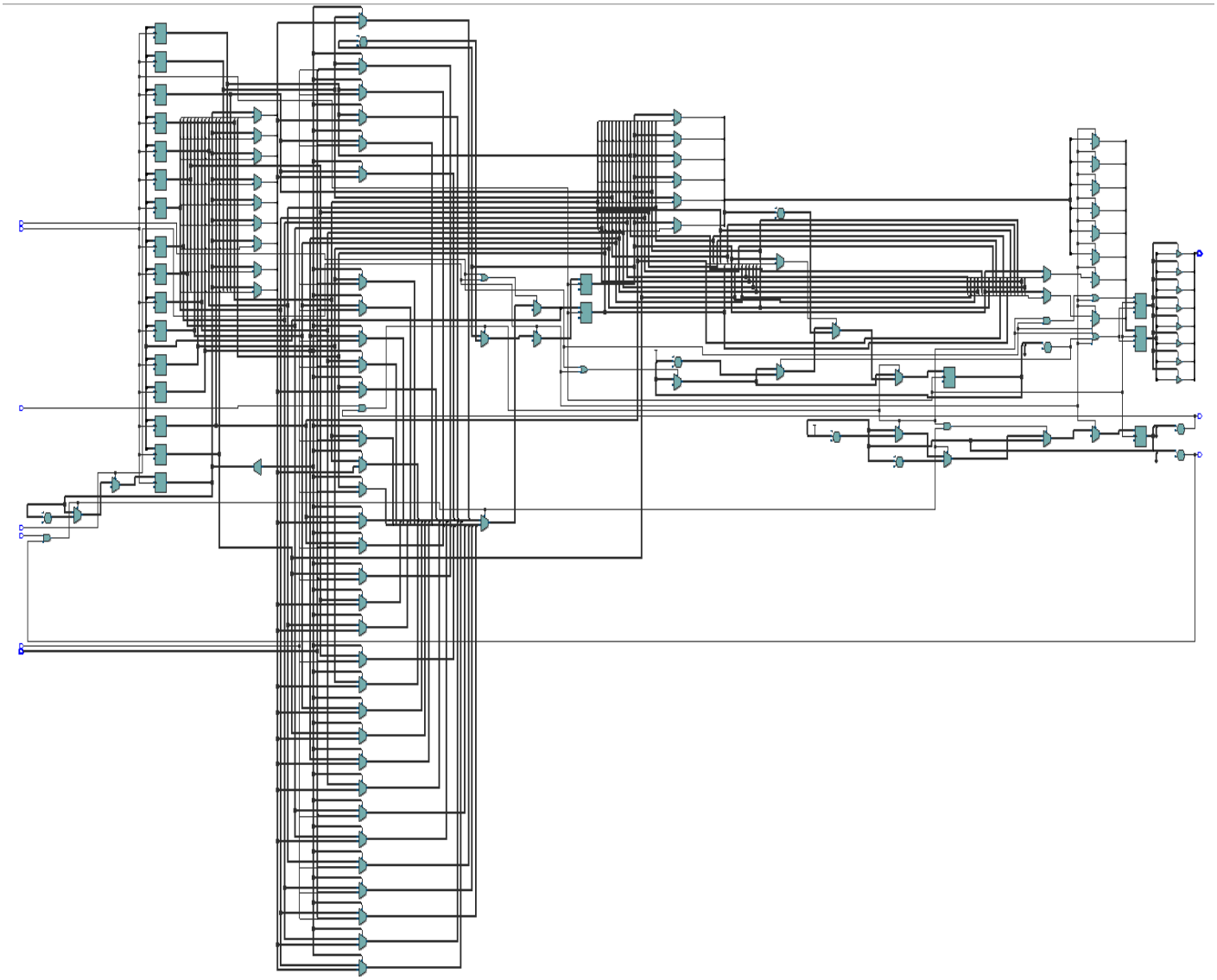
```
task delay;
begin
    #10;
end
endtask

task write;
reg[7:0] payload_data, parity,header;
reg[5:0] payload_len;
reg[1:0] addr;
integer k;
begin
    @(negedge clk);
    payload_len =6'd14;
    addr=2'b01;
    header= {payload_len,addr};
    din= header;
    ifd_state=1'b1;
    we=1;
    for(k=0;k<payload_len;k=k+1)
        begin
        @(negedge clk);
        ifd_state =0;
        payload_data =  {$random} %256;
        din = payload_data;
        end
    //@(negedge clk);
    //soft_rst =1;
    //@(negedge clk);
    //soft_rst =0;
    @(negedge clk);
    parity={$random}%256;
    din=parity;
    end
endtask

initial
begin
    reset;
    initialize;
    delay;
    write;
    delay;
    initialize;
    delay;
    re=1'b1;
    #200;
    $finish;
end

initial
    $monitor("din =%b,we =%b,re =%b,dout =%b",din,we,re,dout);

endmodule
```

Wave form:



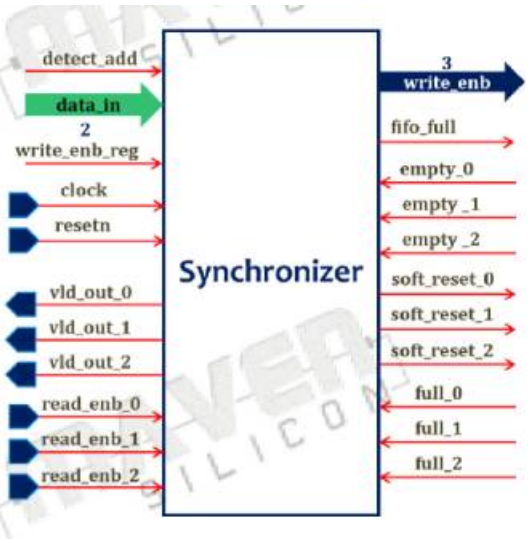


RTL View:



Router-Synchronizer:

Block diagram:



Functionality:

This module provides synchronization between router FSM and router FIFO modules. It provides faithful communication between the single input port and three output ports.

- **detect_add** and **data_in** signals are used to select a FIFO till a packet routing is over for the selected FIFO.
- Signal **fifo_full** signal is asserted based on full status of FIFO_0 or FIFO_1 or FIFO_2.
- If **data_in** = 2'b00 then **fifo_full = full_0**
- If **data_in** = 2'b01 then **fifo_full = full_1**
- If **data_in** = 2'b10 then **fifo_full = full_2 else fifo_full = 0**
- The signal **vld_out_x** signal is generated based on empty status of the FIFO as shown below
  - **vld_out_0 = ~empty_0**
  - **vld_out_1 = ~empty_1**
  - **vld_out_2 = ~empty_2**
- The **write_enb_reg** signal is used to generate **write_enb** signal for the write operation of the selected FIFO.
- There are 3 internal reset signals **(soft_reset_0,soft_reset_1,soft_reset_2)** for each of the FIFO respectively. The respective internal reset signals goes high if read_enb_X (read_enb_0, read_enb_1 or read_enb_2) is not asserted within 30 clock cycles of the vld_out_X (vld_out_0, vld_out_1 or vld_out_2) being asserted respectively.

RTL:

```
module router_sync(clk,resetn,detect_add,we_reg,re_0,re_1,
                   re_2,empty_0,empty_1,empty_2,full_0,
                   full_1,full_2,datain,
                   v_out_0,v_out_1,v_out_2,we,fifo_full,
                   soft_rst_0,soft_rst_1,soft_rst_2);

input clk,resetn,detect_add,we_reg,re_0,re_1,re_2;
input empty_0,empty_1,empty_2,full_0,full_1,full_2;
input [1:0]datain;
output wire v_out_0,v_out_1,v_out_2;
output reg [2:0]we;
output reg fifo_full, soft_rst_0,soft_rst_1,soft_rst_2;

reg [1:0]temp;
reg [4:0]count0,count1,count2;

//sendataing datain to temp once address ditected
always@(posedge clk)
begin
    if(!resetn)
        temp <= 2'd0;
    else if(detect_add)
        temp<=datain;
    end

//write enable
always@(*)
begin
    if(we_reg)
    begin
        case(temp)
            2'b00: we=3'b001;
            2'b01: we=3'b010;
            2'b10: we=3'b100;
            default: we=3'b000;
        endcase
    end
    else
        we = 3'b000;
end

//for fifo full
always@(*)
begin
    case(temp)
        2'b00: fifo_full=full_0;
        2'b01: fifo_full=full_1;
        2'b10: fifo_full=full_2;
        default fifo_full=0;
    endcase
end
```

```
//valid out
assign v_out_0 = !empty_0;
assign v_out_1 = !empty_1;
assign v_out_2 = !empty_2;

//soft reset counter
always@(posedge clk)
begin
    if(!resetn)
        count0<=5'b0;
    else if(!empty_0)
    begin
        if(!re_0)
        begin
            if(count0==5'b11110)
            begin
                soft_rst_0<=1'b1;
                count0<=5'b0;
            end
            else
            begin
                count0<=count0+1'b1;
                soft_rst_0<=1'b0;
            end
        end
        else
            count0<=5'd0;
    end
    else
        count0<=5'd0;
end
always@(posedge clk)
begin
    if(!resetn)
        count1<=5'b0;
    else if(!empty_1)
    begin
        if(!re_1)
        begin
            if(count1==5'b11110)
            begin
                soft_rst_1<=1'b1;
                count1<=5'b0;
            end
            else
            begin
                count1<=count1+1'b1;
                soft_rst_1<=1'b0;
            end
        end
        else
            count1<=5'd0;
    end
    else
        count1<=5'd0;
end
```

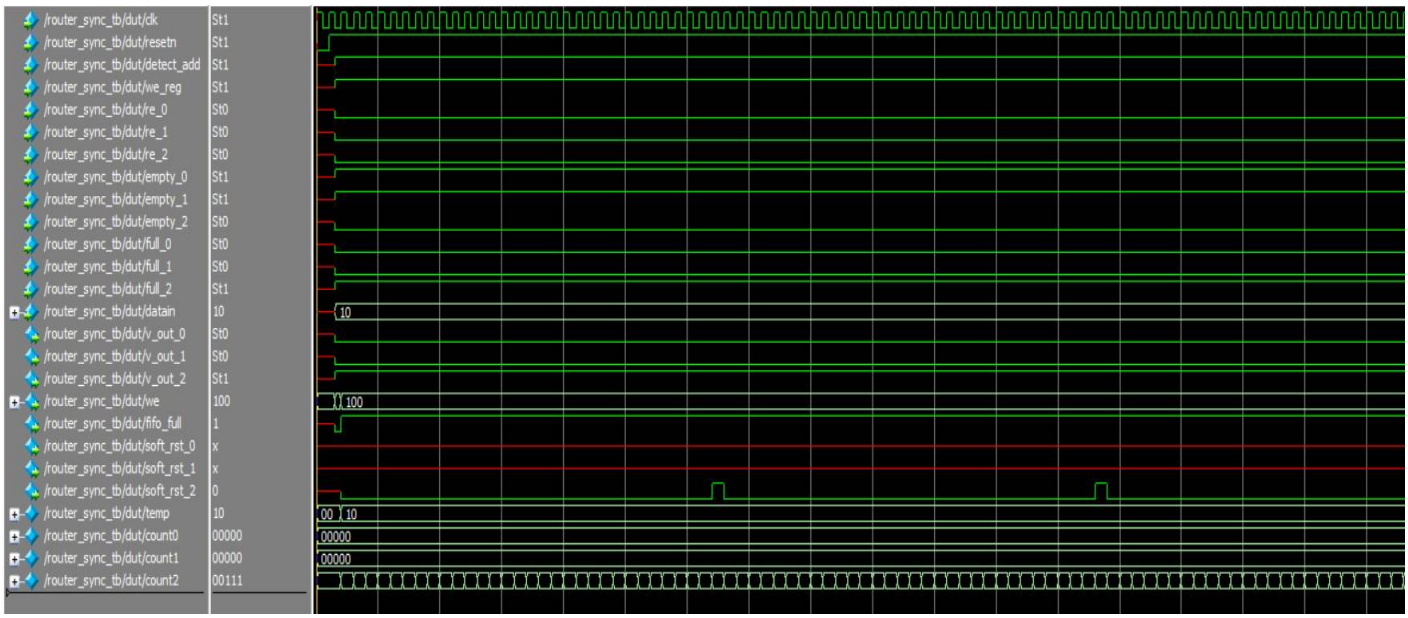
```
always@(posedge clk)
begin
    if(!resetn)
        count2<=5'b0;
    else if(!empty_2)
    begin
        if(!re_2)
        begin
            if(count2==5'b11110)
            begin
                soft_rst_2<=1'b1;
                count2<=5'b0;
            end
            else
            begin
                count2<=count2+1'b1;
                soft_rst_2<=1'b0;
            end
        end
        else
            count2<=5'd0;
    end
    else
        count2<=5'd0;
end

endmodule
```

Test Bench:

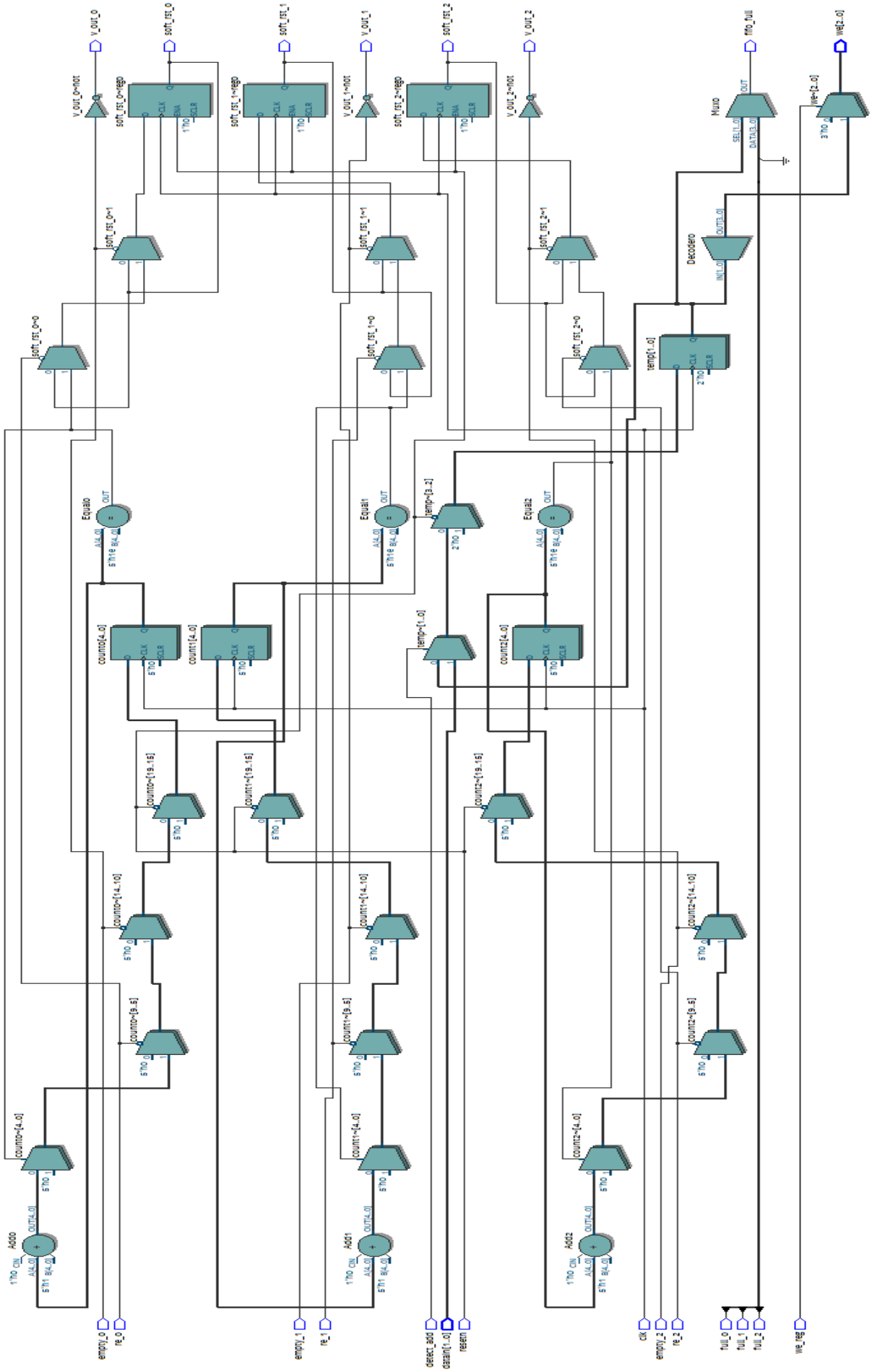
```
module router_sync_tb();

reg clk, rst, d_addr,we_reg,re_0, re_1, re_2, empty_0;
reg empty_1, empty_2,full_0, full_1, full_2;
reg [1:0]din;
wire [2:0]write_enb;
wire v_out_0, v_out_1, v_out_2,fifo_full, soft_rst_0;
wire soft_rst_1, soft_rst_2;

router_sync dut(clk,rst,d_addr,we_reg,re_0,re_1,re_2,empty_0,empty_1,
                empty_2,full_0,full_1,full_2,din, v_out_0,v_out_1,v_out_2,
                we,fifo_full, soft_rst_0,soft_rst_1,soft_rst_2);

initial
begin
clk = 1;
forever
#5 clk=~clk;
end

task reset;
begin
    rst=1'b0;
    #10;
    rst=1'b1;
end
endtask
```

```
task stimulus();
begin
    d_addr=1'b1;
    din=2'b10;
    re_0=1'b0;
    re_1=1'b0;
    re_2=1'b0;
    we_reg=1'b1;
    full_0=1'b0;
    full_1=1'b0;
    full_2=1'b1;
    empty_0=1'b1;
    empty_1=1'b1;
    empty_2=1'b0;
end
endtask

initial
begin
    reset;
    #5;
    stimulus;
    #1000;
    $finish;
end
endmodule
```

Wave form:

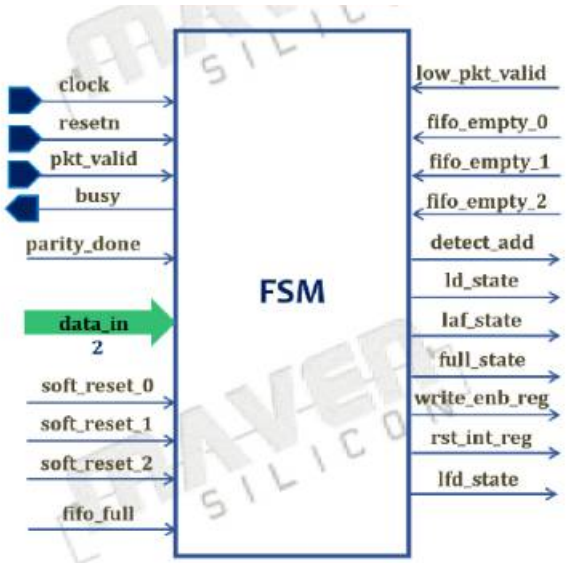


RTL View:



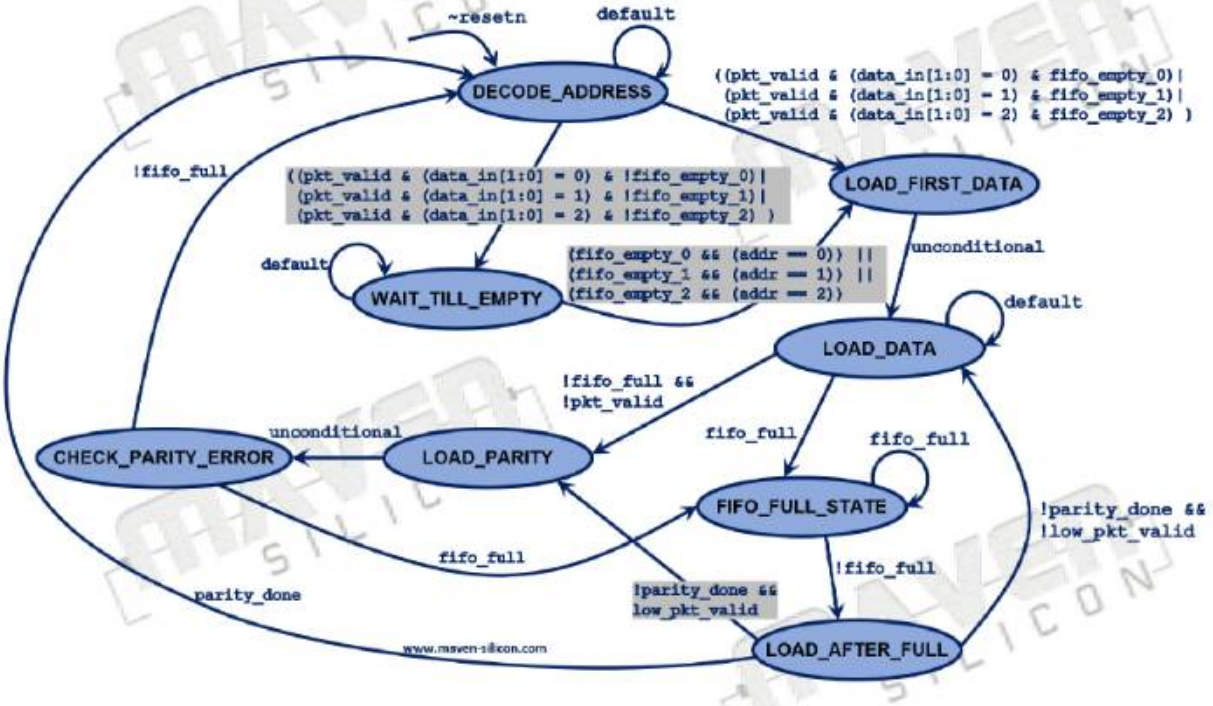
Router-FSM:

Block diagram:



Functionality:

The FSM module is the controller circuit for the ROUTER. This module generates all the control signals when a new packet is received by the ROUTER. These control signals are used by other design components in order to transfer the packet to the output port.



STATE – DECODE_ADDRESS

- This is the initial reset state.
- Signal **detect_add** is asserted in this state which is used to detect an incoming packet. It is also used to latch the first byte as a header byte.

STATE - LOAD_FIRST_DATA

- Signal **lfd_state** is asserted in this state which is used to load the first data byte to the FIFO.
- Signal **busy** is also asserted in this state so that header byte that is already latched doesn't update to a new value for the current packet.
- This state is changed to **LOAD_DATA** state unconditionally in the next clock cycle.

STATE - LOAD_DATA

- In this state the signal **ld_state** is asserted which is used to load the payload data to the FIFO.
- Signal **busy** is deasserted in this state, so that ROUTER can receive new data from input source every clock cycle.
- Signal **write_enb_reg** is asserted in this state in order to write the the Packet information(Header+Payload+Parity)to the selected FIFO.
- This state transits to LOAD_PARITY state when pkt_valid goes low and to FIFO_FULL_STATE when FIFO is full.

STATE – LOAD_PARITY

- In this state the last byte is latched which is the parity byte.
- It goes unconditionally to the state CHECK_PARITY_ERROR..
- Signal **busy** is asserted so that ROUTER doesn't accepts any new data.
- **write_enb_reg** is made high for latching the parity byte to FIFO.

STATE – FIFO_FULL_STATE

- **busy** signal is made high and **write_enb_reg** signal is made low.
- Signal **full_state** is asserted which detects the FIFO full state.

STATE – LOAD_AFTER_FULL

- In this state **laf_state** signal is asserted which is used to latch the data after FIFO_FULL_STATE.
- Signal **busy** & **write_enb_reg** is asserted.
- It checks for **parity_done** signal and if it is high ,shows that LOAD_PARITY state has been detected and it goes to the state DECODE_ADDRESS.
- If **low_pkt_valid** is high it goes to LOAD_PARITY state otherwise it goes back to the LOAD_DATA state.

STATE – WAIT_TILL_EMPTY

- **busy** signal is made high and **write_enb_reg** signal is made low.

STATE – CHECK_PARITY_ERROR

- In this state **rst_int_reg** signal is generated, which is used to reset **low_pkt_valid** signal.
- This state changes to **DECODE_ADDRESS** when **FIFO** is not full and to **FIFO_FULL_STATE** when **FIFO** is full.
- **busy** is asserted in this state.

P.S: The Soft-reset signals should be used in the FSM in such a way that the current state should change back to **"DECODE_ADDRESS"** state only for the timeout situation of the current transmitted packet.

RTL:

```
module router_fsm(clk,resetn,pkt_valid,parity_done,datain,soft_rst_0,
                  soft_rst_1,soft_rst_2,fifo_full,low_pkt_valid,
                  fifo_empty_0,fifo_empty_1,fifo_empty_2,
                  detect_add,ld_state,laf_state,full_state,
                  we_reg,rst_int_reg,lfd_state,busy);

parameter
decode_address      =3'b000,
load_firesetn_data  =3'b001,
load_data           =3'b010,
fifo_full_state     =3'b011,
load_after_full     =3'b100,
load_parity         =3'b101,
check_parity_error  =3'b110,
wait_till_empty     =3'b111;


input clk,resetn,pkt_valid,parity_done,soft_rst_0,soft_rst_1,soft_rst_2;
input fifo_full,low_pkt_valid,fifo_empty_0,fifo_empty_1,fifo_empty_2;
input [1:0] datain;
output detect_add,ld_state,laf_state,full_state,we_reg,rst_int_reg,lfd_state,busy;

reg[3:0] present_state,next_state;
reg [1:0]addr;

always @(posedge clk)
begin
    if(!resetn)
        addr<=1'b0;
    else
        addr<=datain;
end
always@(posedge clk)
begin
    if(!resetn)
        present_state<=decode_address;
    else if (((soft_rst_0) && (addr==2'b00)) ||
             ((soft_rst_1) && (addr==2'b01)) ||
             ((soft_rst_2) && (addr==2'b10)))
```

```
        present_state<=decode_address;
    else
        present_state<=next_state;

end
always@(*)
begin
    case(present_state)
        decode_address :
        begin
            if((pkt_valid & (datain == 2'b00) & fifo_empty_0) |
                (pkt_valid & (datain == 2'b01) & fifo_empty_1) |
                (pkt_valid & (datain == 2'b10) & fifo_empty_2) )
                next_state =load_firesetn_data;
            else if ((pkt_valid & (datain == 2'b00) & !fifo_empty_0) |
                     (pkt_valid & (datain == 2'b01) & !fifo_empty_0) |
                     (pkt_valid & (datain == 2'b10) & !fifo_empty_0) )
                next_state =wait_till_empty;
            else
                next_state = decode_address;
        end

        load_firesetn_data:
        begin
            next_state =load_data;
        end

        load_data:
        begin
            if(!fifo_full && !pkt_valid)
                next_state = load_parity;
            else if (fifo_full)
                next_state =fifo_full_state;
            else
                next_state = load_data;
        end

        load_parity:
        begin
            next_state = check_parity_error;
        end
        fifo_full_state:
        begin
            if (!fifo_full)
                next_state = load_after_full;
            else
                next_state = fifo_full_state;
        end

        load_after_full:
        begin
            if (!parity_done && !low_pkt_valid)
                next_state = load_data;
            else if (!parity_done && low_pkt_valid)
                next_state = load_parity;
            else
            begin
```

```
                if(parity_done==1'b1)
                    next_state=decode_address;
                else
                    next_state=load_after_full;
            end
        end

        wait_till_empty:
        begin
            if ((fifo_empty_0 && (addr == 2'b00)) ||
                (fifo_empty_0 && (addr == 2'b01)) ||
                (fifo_empty_0 && (addr == 2'b10)))
                next_state = load_firesetn_data;
            else
                next_state = wait_till_empty;
        end
        check_parity_error:
        begin
            if (!fifo_full)
                next_state = decode_address;
            else if(fifo_full)
                next_state = fifo_full_state;
            else
                next_state = check_parity_error;
        end
        default:
            next_state =decode_address;
    endcase
end
assign busy=((present_state==load_firesetn_data)||(present_state==load_parity)||
            (present_state==fifo_full_state)||(present_state==load_after_full)||
            (present_state==wait_till_empty)||(present_state==check_parity_error))?1:0;
assign detect_add=((present_state==decode_address))?1:0;
assign lfd_state=((present_state==load_firesetn_data))?1:0;
assign ld_state=((present_state==load_data))?1:0;
assign we_reg=((present_state==load_data)||(present_state==load_after_full)||
                (present_state==load_parity))?1:0;
assign full_state=((present_state==fifo_full_state))?1:0;
assign laf_state=((present_state==load_after_full))?1:0;
assign rst_int_reg=((present_state==check_parity_error))?1:0;

endmodule
```

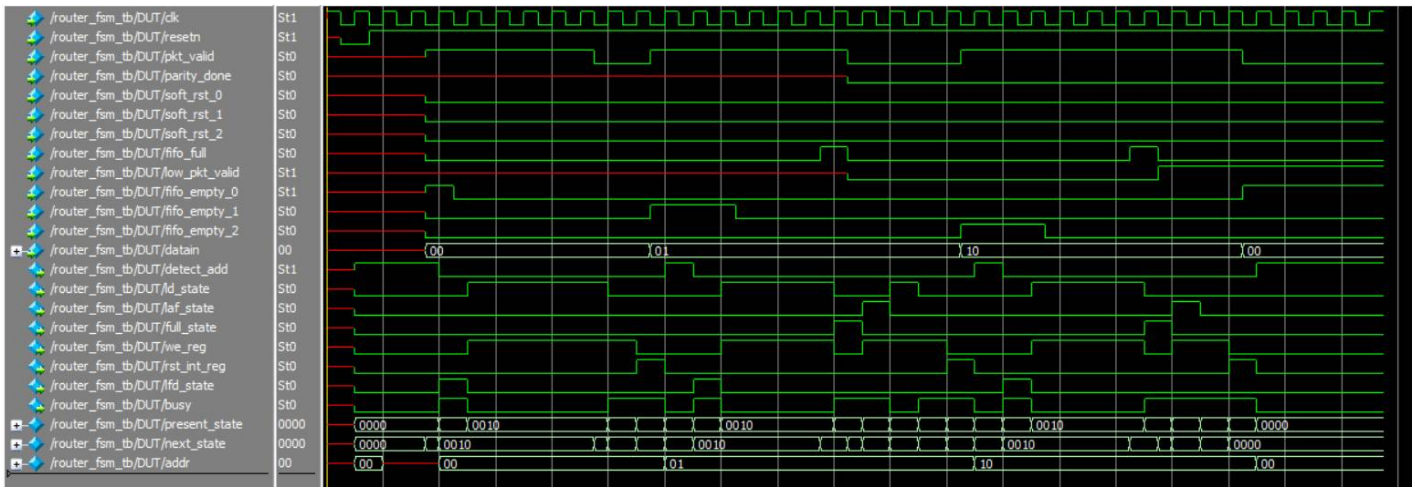
Test Bench:

```
module router_fsm_tb();

reg clk,rst,pkt_valid,parity_done,soft_rst_0,soft_rst_1;
reg soft_rst_2,fifo_full,low_pkt_valid,fifo_empty_0;
reg fifo_empty_1,fifo_empty_2;
reg [1:0] din;

wire busy,d_addr,ld_state,laf_state,full_state;
wire we_reg,rst_int_reg,lfd_state;

router_fsm DUT(clk,rst,pkt_valid,parity_done,din,soft_rst_0,
               soft_rst_1,soft_rst_2,fifo_full,low_pkt_valid,
               fifo_empty_0,fifo_empty_1,fifo_empty_2,
               d_addr,ld_state,laf_state,full_state,we_reg,
               rst_int_reg,lfd_state,busy);

initial
begin
clk=1'b1;
forever
#5 clk=~clk;
end

task reset;
begin
@(negedge clk)
rst=1'b0;
@(negedge clk)
rst=1'b1;
end
endtask
task task1;
begin
    pkt_valid=1'b1;
    din=2'b00;
    fifo_empty_0=1'b1;
    fifo_empty_1=1'b0;
    fifo_empty_2=1'b0;
    fifo_full=1'b0;
    soft_rst_0=1'b0;
    soft_rst_1=1'b0;
    soft_rst_2=1'b0;
    #10;
    fifo_empty_0=1'b0;
    #50;
    pkt_valid=1'b0;
end
endtask
```

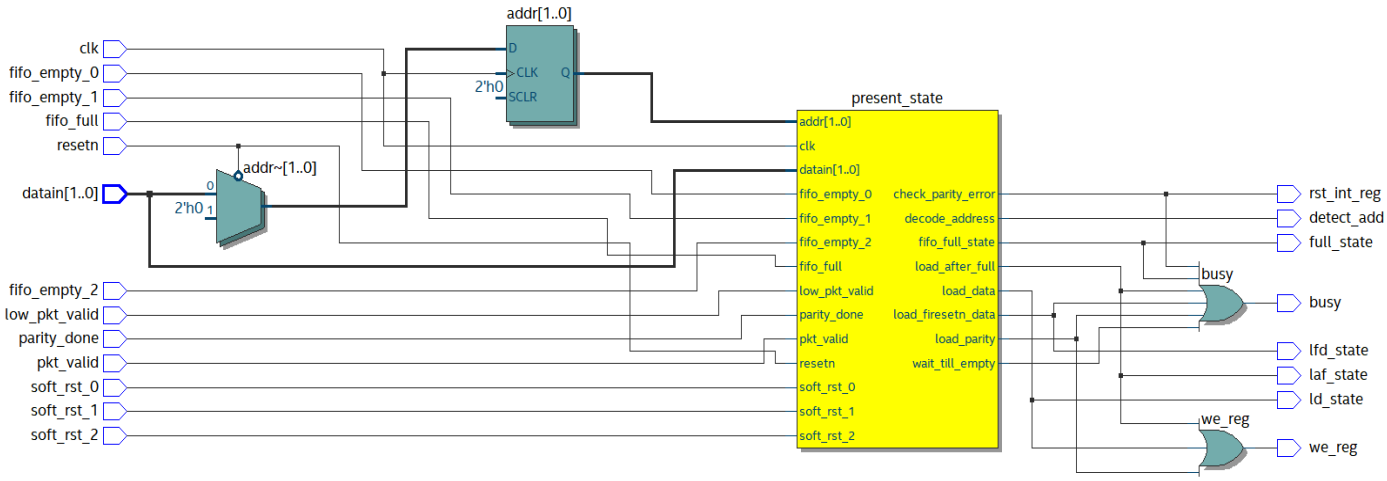
```
task task2;
begin
    pkt_valid=1'b1;
    din=2'b01;
    fifo_full=1'b0;
    fifo_empty_0=1'b0;
    fifo_empty_1=1'b1;
    fifo_empty_2=1'b0;
    soft_rst_0=1'b0;
    soft_rst_1=1'b0;
    soft_rst_2=1'b0;
    #30;
    fifo_empty_1=1'b0;
    #30;
    fifo_full=1'b1;
    #10;
    fifo_full=1'b0;
    parity_done =1'b0;
    low_pkt_valid=1'b0;
    pkt_valid=1'b0;
end
endtask


task task3;
begin
    pkt_valid=1'b1;
    din=2'b10;
    fifo_full=1'b0;
    fifo_empty_0=1'b0;
    fifo_empty_1=1'b0;
    fifo_empty_2=1'b1;
    soft_rst_0=1'b0;
    soft_rst_1=1'b0;
    soft_rst_2=1'b0;
    #30;
    fifo_empty_2=1'b0;
    #30;
    fifo_full=1'b1;
    #10;
    fifo_full=1'b0;
    parity_done =1'b0;
    low_pkt_valid=1'b1;
end
endtask
```

```
task task4;
begin
    pkt_valid=1'b0;
    din=2'b00;
    fifo_full=1'b0;
    fifo_empty_0=1'b1;
    fifo_empty_1=1'b0;
    fifo_empty_2=1'b0;
    soft_rst_0=1'b0;
    soft_rst_1=1'b0;
    soft_rst_2=1'b0;

end
endtask

initial
begin
    reset;
    #20;
    task1;
    #20;
    task2;
    #40;
    task3;
    #30;
    task4;
    #50;
    $finish;
end
endmodule
```

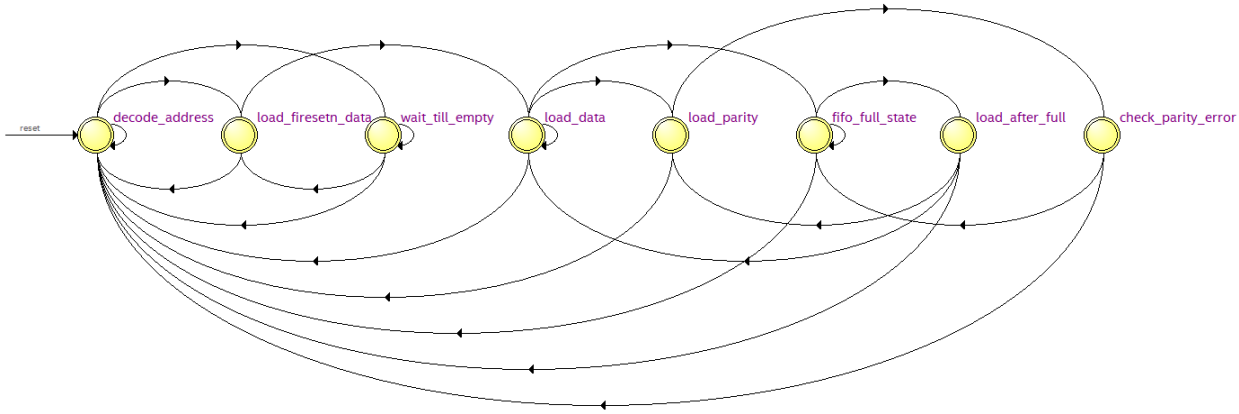
Wave form:



RTL View:



State Machine viewer:

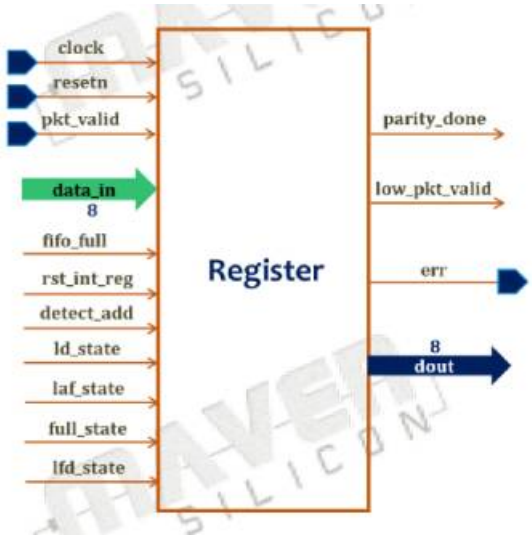


State table:

	Source State	Destination State	
1	check_parity_error	fifo_full_state	(fifo_full),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (fifo_full),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (fifo_full),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (fifo_full),(addr[0]),(addr[1]),(resetn)
2	check_parity_error	decode_address	(!fifo_full) + (fifo_full),(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (fifo_full),(!addr[0]),(!addr[1]),(soft_rst_0) + (fifo_full),(!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (fifo_full),(!addr[0]),(addr[1]),(soft_rst_2) + (fifo_full),(addr[0]),(!addr[1]),(!soft
3	decode_address	wait_till_empty	(!addr[0]),(!addr[1]),(!fifo_empty_0),(!datain[0]),(!datain[1]),(pkt_valid),(!soft_rst_0),(resetn) + (!addr[0]),(!addr[1]),(!fifo_empty_0),(!datain[0]),(datain[1]),(pkt_valid),(!fifo_empty_2),(!soft_rst_0),(resetn) + (!addr[0]),(!addr[1]),(!fifo_empty_0),(!
4	decode_address	load_firesetn_data	(!addr[0]),(!addr[1]),(!fifo_empty_0),(!datain[0]),(datain[1]),(pkt_valid),(fifo_empty_2),(!soft_rst_0),(resetn) + (!addr[0]),(!addr[1]),(!fifo_empty_0),(datain[0]),(!datain[1]),(pkt_valid),(fifo_empty_1),(!soft_rst_0),(resetn) + (!addr[0]),(!addr[1]),(fif
5	decode_address	decode_address	(!datain[0]),(!datain[1]),(!pkt_valid) + (!datain[0]),(!datain[1]),(pkt_valid),(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!datain[0]),(!datain[1]),(pkt_valid),(!addr[0]),(!addr[1]),(soft_rst_0) + (!datain[0]),(!datain[1]),(pkt_valid),(!addr[0]),(addr[1]),(!
6	fifo_full_state	load_after_full	(!fifo_full),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!fifo_full),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (!fifo_full),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (!fifo_full),(addr[0]),(addr[1]),(resetn)
7	fifo_full_state	fifo_full_state	(fifo_full),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (fifo_full),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (fifo_full),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (fifo_full),(addr[0]),(addr[1]),(resetn)
8	fifo_full_state	decode_address	(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!addr[0]),(!addr[1]),(soft_rst_0) + (!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (!addr[0]),(addr[1]),(soft_rst_2) + (addr[0]),(!addr[1]),(!soft_rst_1),(!resetn) + (addr[0]),(!addr[1]),(soft_rst_1) + (addr[0]
9	load_after_full	load_parity	(!parity_done),(low_pkt_valid),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!parity_done),(low_pkt_valid),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (!parity_done),(low_pkt_valid),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (!parity_done),(low
10	load_after_full	load_data	(!parity_done),(!low_pkt_valid),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!parity_done),(!low_pkt_valid),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (!parity_done),(!low_pkt_valid),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (!parity_done),(!l
11	load_after_full	decode_address	(!parity_done),(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!parity_done),(!addr[0]),(!addr[1]),(soft_rst_0) + (!parity_done),(!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (!parity_done),(!addr[0]),(addr[1]),(soft_rst_2) + (!parity_done),(addr[0]),(!
12	load_data	load_parity	(!pkt_valid),(!fifo_full),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!pkt_valid),(!fifo_full),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (!pkt_valid),(!fifo_full),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (!pkt_valid),(!fifo_full),(addr[0]),(addr[1]),(rese
13	load_data	load_data	(!fifo_full),(pkt_valid),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!fifo_full),(pkt_valid),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (!fifo_full),(pkt_valid),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (!fifo_full),(pkt_valid),(addr[0]),(addr[1]),(resetn
14	load_data	fifo_full_state	(fifo_full),(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (fifo_full),(!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (fifo_full),(addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (fifo_full),(addr[0]),(addr[1]),(resetn)
15	load_data	decode_address	(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!addr[0]),(!addr[1]),(soft_rst_0) + (!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (!addr[0]),(addr[1]),(soft_rst_2) + (addr[0]),(!addr[1]),(!soft_rst_1),(!resetn) + (addr[0]),(!addr[1]),(soft_rst_1) + (addr[0]
16	load_firesetn_data	load_data	(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (addr[0]),(addr[1]),(resetn)
17	load_firesetn_data	decode_address	(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!addr[0]),(!addr[1]),(soft_rst_0) + (!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (!addr[0]),(addr[1]),(soft_rst_2) + (addr[0]),(!addr[1]),(!soft_rst_1),(!resetn) + (addr[0]),(!addr[1]),(soft_rst_1) + (addr[0]
18	load_parity	decode_address	(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!addr[0]),(!addr[1]),(soft_rst_0) + (!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (!addr[0]),(addr[1]),(soft_rst_2) + (addr[0]),(!addr[1]),(!soft_rst_1),(!resetn) + (addr[0]),(!addr[1]),(soft_rst_1) + (addr[0]
19	load_parity	check_parity_err...	(!addr[0]),(!addr[1]),(!soft_rst_0),(resetn) + (!addr[0]),(addr[1]),(!soft_rst_2),(resetn) + (addr[0]),(!addr[1]),(!soft_rst_1),(resetn) + (addr[0]),(addr[1]),(resetn)
20	wait_till_empty	wait_till_empty	(!addr[0]),(!addr[1]),(!fifo_empty_0),(!soft_rst_0),(resetn) + (!addr[0]),(addr[1]),(!fifo_empty_0),(!soft_rst_2),(resetn) + (addr[0]),(!addr[1]),(!fifo_empty_0),(!soft_rst_1),(resetn) + (addr[0]),(addr[1]),(resetn)
21	wait_till_empty	load_firesetn_data	(!addr[0]),(!addr[1]),(fifo_empty_0),(!soft_rst_0),(resetn) + (!addr[0]),(addr[1]),(fifo_empty_0),(!soft_rst_2),(resetn) + (addr[0]),(!addr[1]),(fifo_empty_0),(!soft_rst_1),(resetn)
22	wait_till_empty	decode_address	(!addr[0]),(!addr[1]),(!soft_rst_0),(!resetn) + (!addr[0]),(!addr[1]),(soft_rst_0) + (!addr[0]),(addr[1]),(!soft_rst_2),(!resetn) + (!addr[0]),(addr[1]),(soft_rst_2) + (addr[0]),(!addr[1]),(!soft_rst_1),(!resetn) + (addr[0]),(!addr[1]),(soft_rst_1) + (addr[0]

Router-Register:

Block diagram:



Functionality:

This module implements 4 internal registers in order to hold header byte, FIFO full state byte ,internal parity and packet parity byte.

All the registers in this module are latched on the rising edge of the **clock**.

- If **resetn** is low then the signals (**dout, err, parity_done and low_pkt_valid**) are made low.
- The signal **parity_done** is high under the following conditions
  - When signal **ld_state** is high and signals (**fifo_full and pkt_valid**) are low.
  - When signals **laf_state** and **low_pkt_valid** both are high and the previous value of **parity_done** is low.
- **rst_int_reg** signal is used to reset **low_pkt_valid** signal.
- **detect_add** signal is used to reset **parity_done** signal.
- **Signal low_pkt_valid** is high when **ld_state** is high and **pkt_valid** is low. **low_pkt_valid** shows that pkt_valid for current packet has been deasserted.
- First data byte i.e., header is latched inside an internal register when **detect_add** and **pkt_valid** signals are high. This data is latched to the output **dout** when **lfd_state** signal goes high.
- Then signal **data_in** i.e payload is latched to **dout** if **ld_state** signal is high and **fifo_full** is low.
- Signal **data_in** is latched to an internal register when **ld_state** and **fifo_full** are high.This data is latched to output **dout** when **laf_state** goes high.

- **full_state** is used to calculate internal parity.
- Another internal register is used to store internal parity for parity matching .Internal parity is calculated using the bit-wise xor operation between header byte,payload byte and previous parity values as shown below :

  parity_reg = parity_reg_previous ^ header_byte ---- t1 clock cycle

  parity_reg = parity_reg_previous ^ payload_byte1 --- t2 clock cycle

  parity_reg = parity_reg_previous ^ payload_byte 2--- t3 clock cycle

  .

  .

  .

  parity_reg = parity_reg_previous ^ payload_byte n---tn clock cycle

  last payload byte

- The **err** is calculated only after packet parity is loaded and goes high if the packet parity doesn't match with the internal parity.

RTL:

```
module router_reg(clk,resetn,packet_valid,datain,fifo_full,detect_add,
                  ld_state,laf_state,full_state,lfd_state,rst_int_reg,
                  err,parity_done,low_packet_valid,dout);

input clk,resetn,packet_valid;
input [7:0] datain;
input fifo_full,detect_add,ld_state,laf_state,full_state,lfd_state,rst_int_reg;
output reg err,parity_done,low_packet_valid;
output reg [7:0] dout;


reg [7:0] hold_header_byte,fifo_full_state_byte,internal_parity,packet_parity_byte;

//parity done
always@(posedge clk)
begin
    if(!resetn)
    begin
        parity_done<=1'b0;
    end
    else
    begin
        if(ld_state && !fifo_full && !packet_valid)
            parity_done<=1'b1;
        else if(laf_state && low_packet_valid && !parity_done)
            parity_done<=1'b1;
        else
        begin
            if(detect_add)
                parity_done<=1'b0;
        end
    end
end
```

```
//low_packet valid
always@(posedge clk)
begin
    if(!resetn)
        low_packet_valid<=1'b0;
    else
    begin
        if(rst_int_reg)
            low_packet_valid<=1'b0;
        if(ld_state==1'b1 && packet_valid==1'b0)
            low_packet_valid<=1'b1;
    end
end

//dout
always@(posedge clk)
begin
    if(!resetn)
        dout<=8'b0;
    else
    begin
        if(detect_add && packet_valid)
            hold_header_byte<=datain;
        else if(lfd_state)
            dout<=hold_header_byte;
        else if(ld_state && !fifo_full)
            dout<=datain;
        else if(ld_state && fifo_full)
            fifo_full_state_byte<=datain;
        else
        begin
            if(laf_state)
                dout<=fifo_full_state_byte;
        end
    end
end
// internal parity
always@(posedge clk)
begin
    if(!resetn)
        internal_parity<=8'b0;
    else if(lfd_state)
        internal_parity<=internal_parity ^ hold_header_byte;
    else if(ld_state && packet_valid && !full_state)
        internal_parity<=internal_parity ^ datain;
    else
    begin
        if (detect_add)
            internal_parity<=8'b0;
    end
end
```

```
//error and packet_
always@(posedge clk)
begin
    if(!resetn)
        packet_parity_byte<=8'b0;
    else
    begin
        if(!packet_valid && ld_state)
            packet_parity_byte<=datain;
    end
end

//error
always@(posedge clk)
begin
    if(!resetn)
        err<=1'b0;
    else
    begin
        if(parity_done)
        begin
            if(internal_parity!=packet_parity_byte)
                err<=1'b1;
            else
                err<=1'b0;
        end
    end
end

endmodule
```

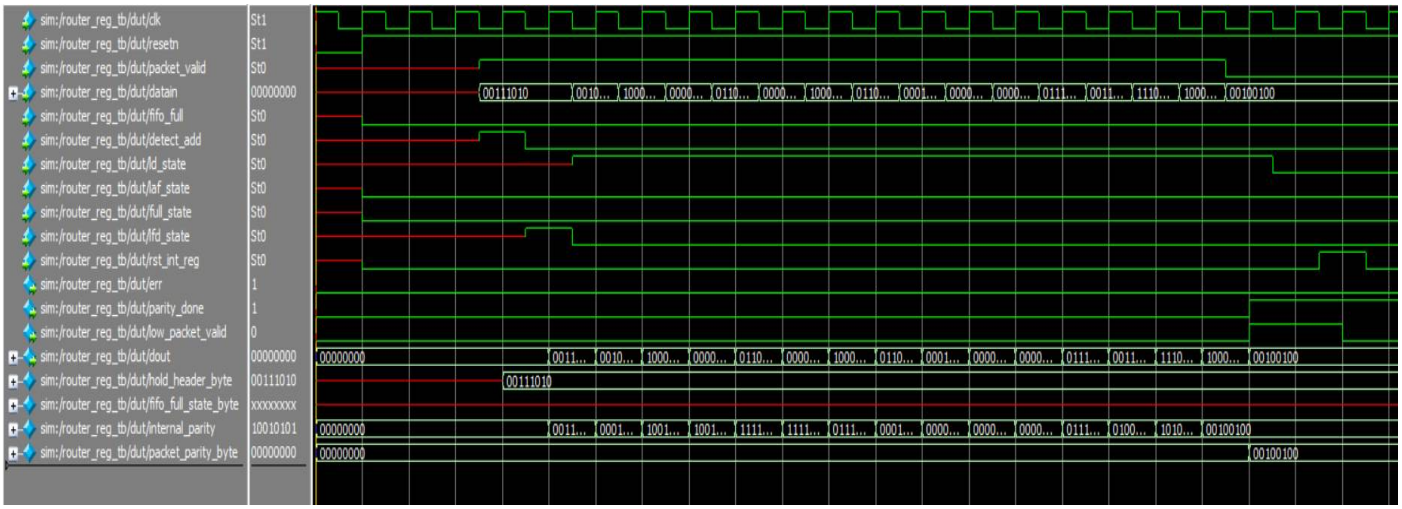
Test Bench:

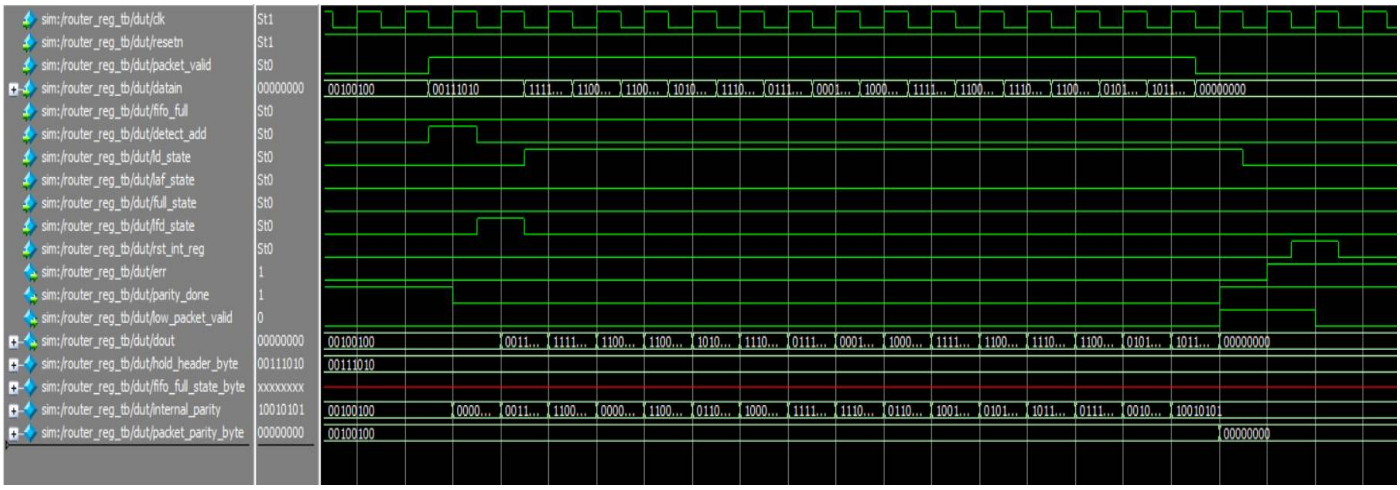
```
module router_reg_tb();

reg clk, resetn, packet_valid,fifo_full, detect_add, ld_state;
reg laf_state, full_state, lfd_state, rst_int_reg;
reg [7:0] datain;
wire err, parity_done, low_packet_valid;
wire [7:0]dout;
integer i;
router_reg dut(clk,resetn,packet_valid,datain,fifo_full,detect_add,
                ld_state,laf_state,full_state,lfd_state,rst_int_reg,
                    err,parity_done,low_packet_valid,dout);

//clock generation
initial
begin
    clk = 1;
forever
    #5 clk=~clk;
end

task reset;
begin
    resetn=1'b0;
    #10;
    resetn=1'b1;
end
endtask
```

```
task packet1();
reg [7:0]header, payload_data, parity;
reg [5:0]payloadlen;
begin
    @(negedge clk);
    payloadlen=14;
    parity=0;
    detect_add=1'b1;
    packet_valid=1'b1;
    header={payloadlen,2'b10};
    datain=header;
    parity=parity^datain;

    @(negedge clk);
    detect_add=1'b0;
    lfd_state=1'b1;

    for(i=0;i<payloadlen;i=i+1)
    begin
        @(negedge clk);
        lfd_state=0;
        ld_state=1;
        payload_data={$random}%256;
        datain=payload_data;
        parity=parity^datain;
    end

    @(negedge clk);
    packet_valid=0;
    datain=parity;

    @(negedge clk);
    ld_state=0;
end
endtask
task packet2();

reg [7:0]header, payload_data, parity;
reg [5:0]payloadlen;
begin
    @(negedge clk);
    payloadlen=14;
    parity=0;
    detect_add=1'b1;
    packet_valid=1'b1;
    header={payloadlen,2'b10};
    datain=header;
    parity=parity^datain;

    @(negedge clk);
    detect_add=1'b0;
    lfd_state=1'b1;

    for(i=0;i<payloadlen;i=i+1)
    begin
```

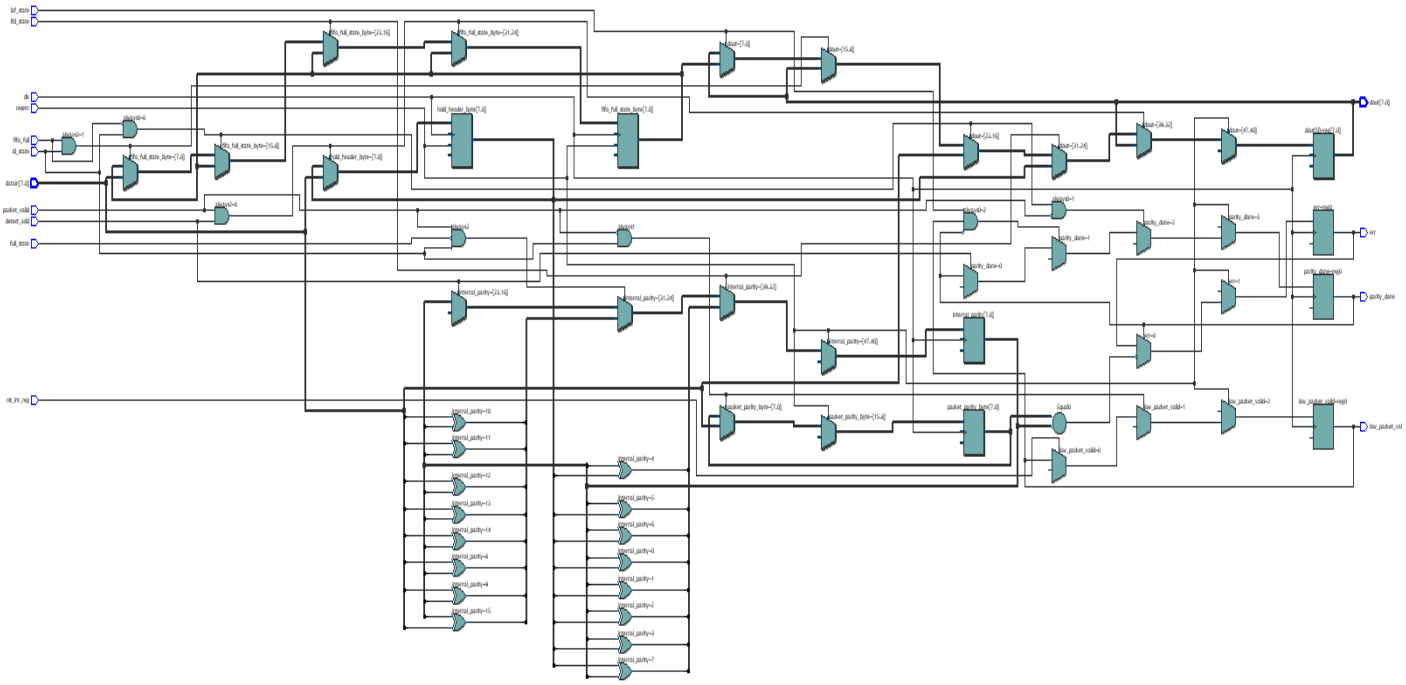
```
        @(negedge clk);
        lfd_state=0;
        ld_state=1;
        payload_data={$random}%256;
        datain=payload_data;
        parity=parity^datain;
    end

    @(negedge clk);
    packet_valid=0;
    datain=!parity;

    @(negedge clk);
    ld_state=0;
end
endtask

initial
    begin
        reset;
        fifo_full=1'b0;
        laf_state=1'b0;
        full_state=1'b0;
        rst_int_reg=1'b0;
        #20;
        packet1();
        #10;
        rst_int_reg=1'b1;
        #10;
        rst_int_reg=1'b0;
        #50;
        packet2();
        #10;
        rst_int_reg=1'b1;
        #10;
        rst_int_reg=1'b0;
        #200;
        $finish;
    end

endmodule
```

Wave form:



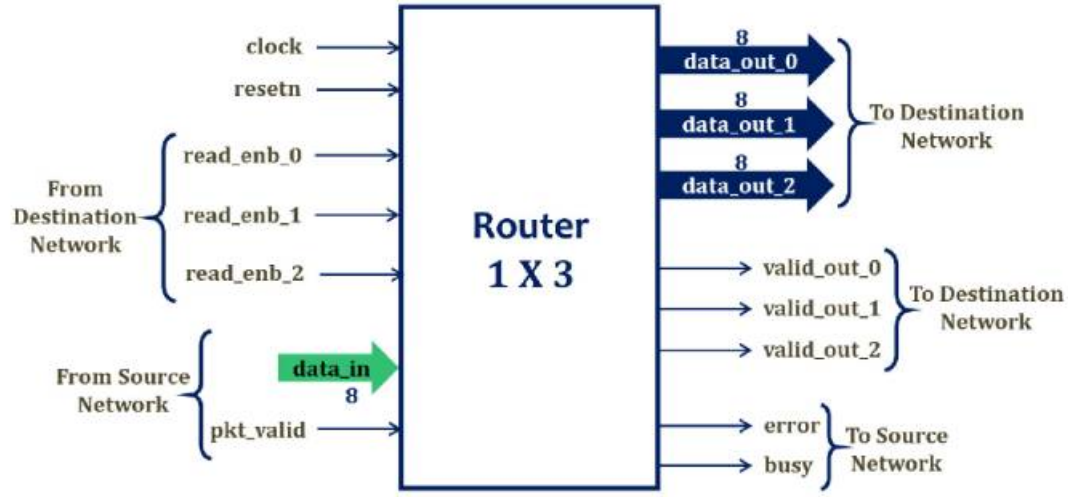


RTL View:

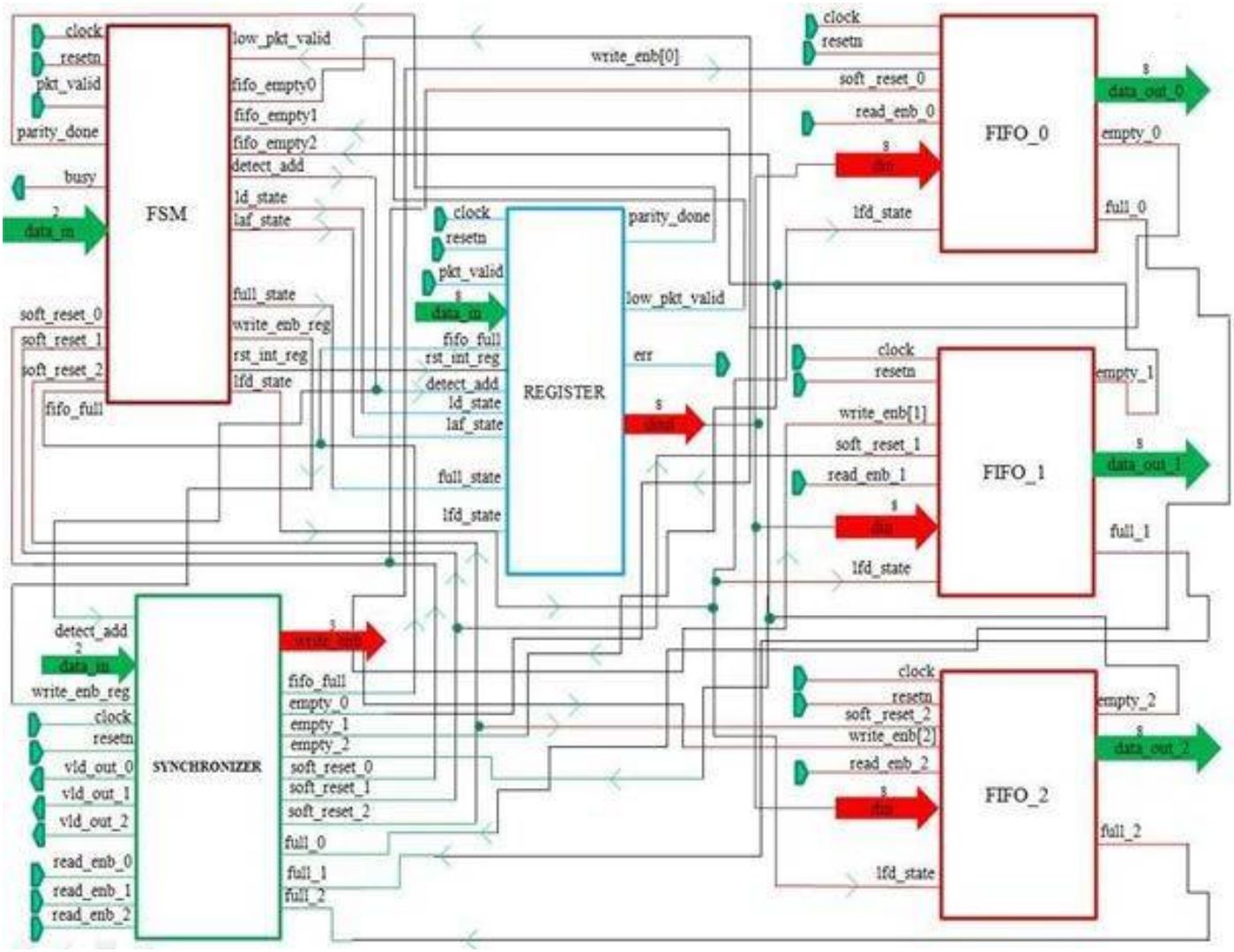


Router-Top:

Block diagram:



Internal block diagram:



Functionality:

clock	Active high clocking event
pkt_valid	pkt_valid is an active high input signal that detects an arrival of a new packet from a source network
resetn	Active low synchronous reset
data_in	8 bit input data bus that transmits the packet from source network to router
read_enb_0	Active high input signal for reading the packet through output data bus data_out_0
read_enb_1	Active high input signal for reading the packet through output data bus data_out_1
read_enb_2	Active high input signal for reading the packet through output data bus data_out_2
data_out_0	8 bit output data bus that transmits the packet from the router to destination client network 1
data_out_1	8 bit output data bus that transmits the packet from the router to destination client network 2
data_out_2	8 bit output data bus that transmits the packet from the router to destination client network 3
vld_out_0	Active high signal that detects that a valid byte is available for destination client network 1
vld_out_1	Active high signal that detects that a valid byte is available for destination client network 2
vld_out_2	Active high signal that detects that a valid byte is available for destination client network 3
busy	Active high signal that detects a busy state for the router that stops accepting any new byte
error	Active high signal that detects the mismatch between packet parity and internal parity

Router- Packet

- **Packet Format :** The Packet consists of 3 parts i.e Header, payload and parity such that each id of 8 bits width and the length of the payload can be extended between 1 byte to 63 bytes.

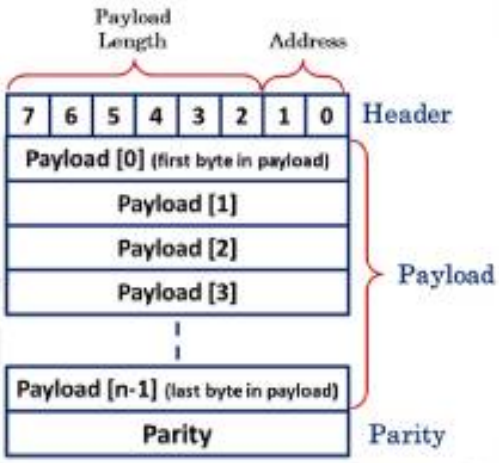
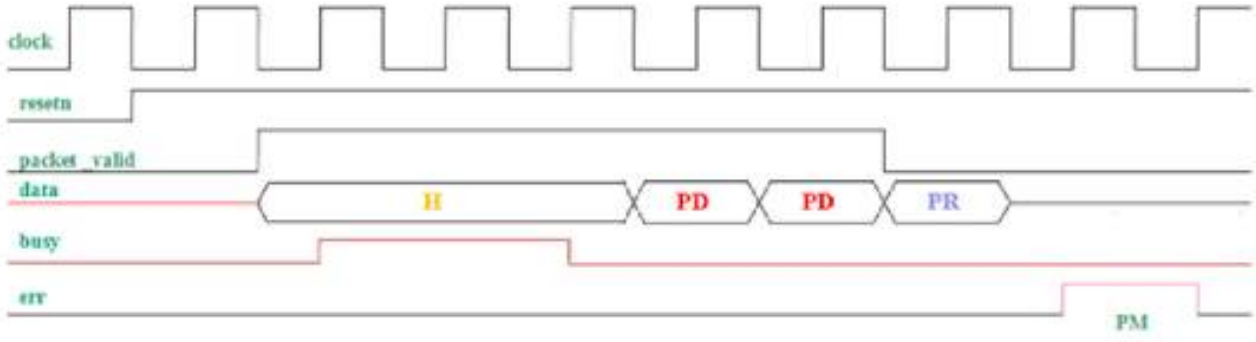


Figure - Packet Format

- **Header :** Packet header contains two fields DA and length.
  - DA: Destination address of the packet is of 2 bits. The router drives the packet to respective ports based on this destination address of the packets. Each output port has 2-bit unique port address. If the destination address of the packet matches the port address, then router drives the packet to the output port. The address "3" is invalid.
  - Length: Length of the data is of 6 bits. It specifies the number of data bytes. A packet can have a minimum data size of 1 byte and a maximum size of 63 bytes.
    If Length = 1, it means data length is 1 byte
    If Length = 63, it means data length is 63 bytes
- **Payload :** Payload is the data information. Data should be in terms of bytes.
- **Parity :** This field contains the security check of the packet. It is calculated as bitwise parity over the header and payload bytes of the packet as mentioned below.

Router- Input Protocol

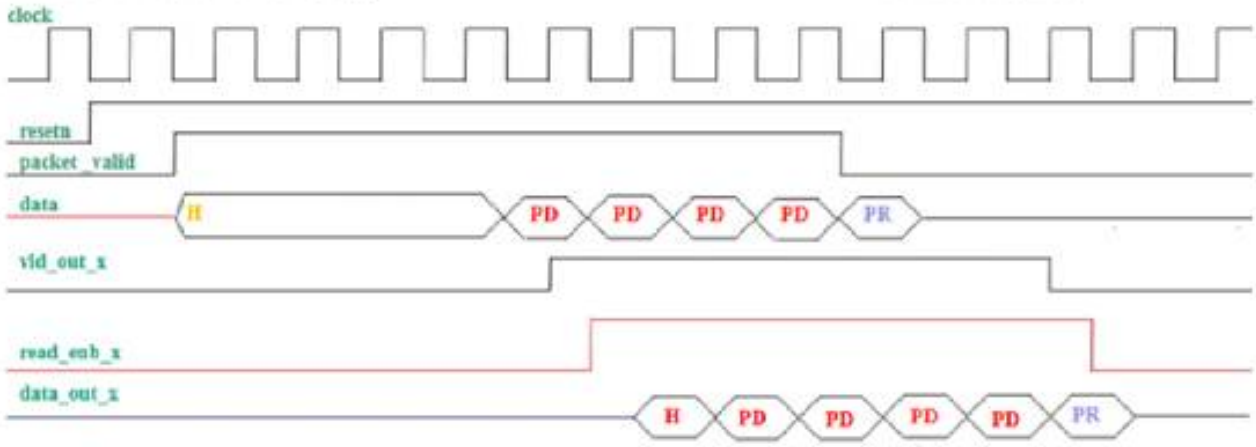


H : Header Byte
PD : Payload Data
PR : Packet Parity byte
PM : Parity Mismatch

The characteristics of the DUT input protocol are as follows:

- **TestBench Note :** All input signals are active high except active low reset and are synchronized to the falling edge of the clock. This is because the DUT router is sensitive to the rising edge of the clock. Therefore, in the testbench, driving input signals on the falling edge ensures adequate setup and hold time. But in the SystemVerilog/UVM based testbench, clocking block can be used to drive the signals on the positive edge of the clock itself and thus avoids metastability.
- The packet_valid signal is asserted on the same clock edge when the header byte is driven onto the input data bus.
- Since the header byte contains the address, this tells the router to which output channel the packet should be routed to (data_out_0, data_out_1, or data_out_2).
- Each subsequent byte of payload after header byte should be driven on the input data bus for every new falling edge of clock.
- After the last payload byte has been driven, on the next falling clock, the packet_valid signal must be deasserted, and the packet parity byte should be driven. This signals completion of the packet.
- The testbench shouldn't drive any bytes when busy signal is detected instead it should hold the last driven value.
- The "busy" signal when asserted drops any incoming byte of data.
- The "err" signal is asserted when a packet parity mismatch is detected.

Router- Output Protocol



H : Header Byte
PD : Payload Data
PR : Packet Parity byte
x : 0,1,2

The characteristics of the output protocol are as follows:

- **TestBench Note :** All output signals are active high and are synchronized to the rising edge of the clock.
- Each output port data_out_X (data_out_0, data_out_1, data_out_2) is internally buffered by a FIFO of size 16X9.
- The router asserts the vld_out_X (vld_out_0, vld_out_1 or vld_out_2) signal when valid data appears on the vld_out_X (data_out_0, data_out_1 or data_out_2) output bus. This is a signal to the receiver's client which indicates that data is available on a particular output data bus.
- The packet receiver will then wait until it has enough space to hold the bytes of the packet and then respond with the assertion of the read_enb_X (read_enb_0, read_enb1 or read_enb_2) signal.
- The read_enb_X (read_enb_0, read_enb_1 or read_enb_2) input signal can be asserted on the falling clock edge in which data are read from the data_out_X (data_out_0, data_out_1 or data_out_2) bus.
- The read_enb_X (read_enb_0, read_enb_1 or read_enb_2) must be asserted within 30 clock cycles of vld_out_X (vld_out_0, vld_out_1 or vld_out_2) being asserted else time-out occurs, which resets the FIFO.
- The data_out_X bus will be tri-stated during a scenario when a packet's byte is lost due to time-out condition.

RTL:

```
module router_top(clk, resetn, packet_valid, read_enb_0, read_enb_1,
                  read_enb_2,datain,vld_out_0, vld_out_1, vld_out_2,
                  err, busy,data_out_0, data_out_1, data_out_2);

input clk, resetn, packet_valid, read_enb_0, read_enb_1, read_enb_2;
input [7:0]datain;
output vld_out_0, vld_out_1, vld_out_2, err, busy;
output [7:0]data_out_0, data_out_1, data_out_2;
wire [2:0]w_enb;
wire [7:0]dout;

router_fifo f0(.clk(clk), .resetn(resetn), .soft_rst(soft_rst_0),
              .lfd_state(lfd_state_w), .we(w_enb[0]), .datain(dout),
              .re(read_enb_0),.full(full_0),.empty(empty_0),.dout(data_out_0));

router_fifo f1(.clk(clk), .resetn(resetn), .soft_rst(soft_reset_1),
               .lfd_state(lfd_state_w), .we(w_enb[1]), .datain(dout),
               .re(read_enb_1),.full(full_1),.empty(empty_1),.dout(data_out_1));

router_fifo f2(.clk(clk), .resetn(resetn), .soft_rst(soft_reset_2),
               .lfd_state(lfd_state_w), .we(w_enb[2]), .datain(dout),
               .re(read_enb_2),.full(full_2),.empty(empty_2),.dout(data_out_2));


router_reg r1(.clk(clk), .resetn(resetn), .packet_valid(packet_valid),.datain(datain),
              .dout(dout), .fifo_full(fifo_full), .detect_add(detect_add),
              .ld_state(ld_state),  .laf_state(laf_state), .full_state(full_state),
              .lfd_state(lfd_state_w), .rst_int_reg(rst_int_reg),  .err(err),
              .parity_done(parity_done),.low_packet_valid(low_packet_valid));
router_fsm fsm(.clk(clk), .resetn(resetn), .pkt_valid(packet_valid),
              .datain(datain[1:0]), .soft_rst_0(soft_rst_0), .soft_rst_1(soft_rst_1),
              .soft_rst_2(soft_reset_2),.fifo_full(fifo_full), .fifo_empty_0(empty_0),
              .fifo_empty_1(empty_1), .fifo_empty_2(empty_2),.parity_done(parity_done),
              .low_pkt_valid(low_packet_valid), .busy(busy), .rst_int_reg(rst_int_reg),
              .full_state(full_state), .lfd_state(lfd_state_w), .laf_state(laf_state),
              .ld_state(ld_state),detect_add(detect_add), .we_reg(write_enb_reg));

router_sync s(.clk(clk), .resetn(resetn), .datain(datain[1:0]), .detect_add(detect_add),
             .full_0(full_0), .full_1(full_1), .full_2(full_2), .re_0(read_enb_0),
             .re_1(read_enb_1), .re_2(read_enb_2), .we_reg(write_enb_reg),
             .empty_0(empty_0), .empty_1(empty_1), .empty_2(empty_2), .v_out_0(vld_out_0),
             .v_out_1(vld_out_1),.v_out_2(vld_out_2), .soft_rst_0(soft_rst_0),
             .soft_rst_1(soft_rst_1), .soft_rst_2(soft_reset_2),
             .we(w_enb), .fifo_full(fifo_full));

endmodule
```

Test Bench:

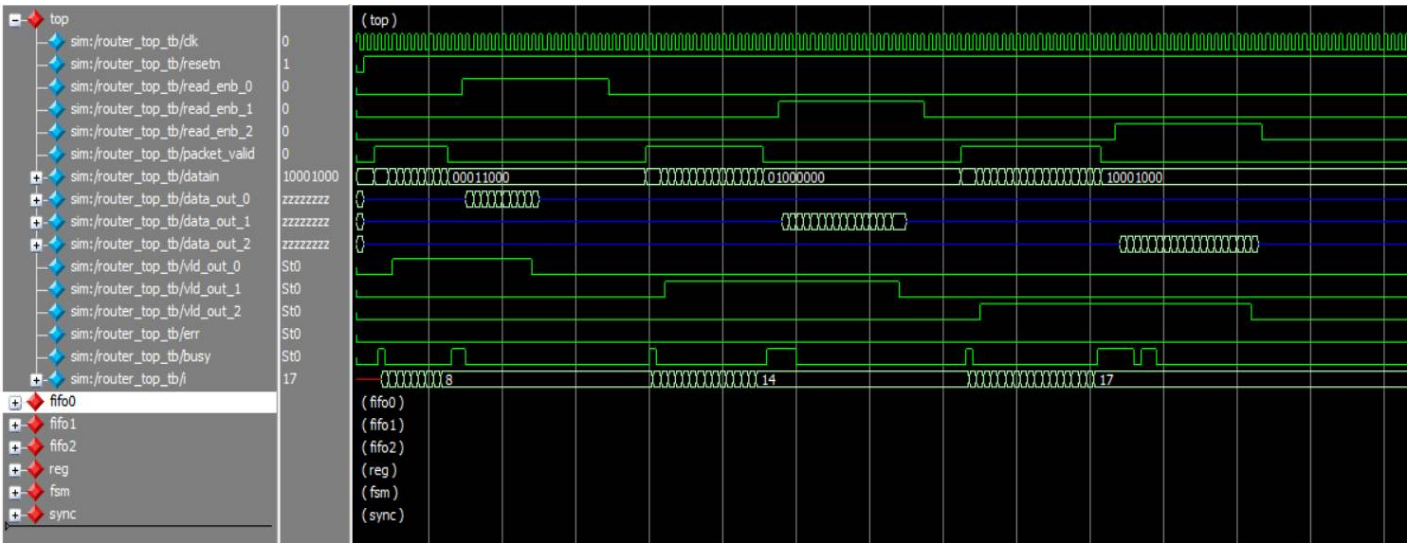
```
module router_top_tb();

reg clk, resetn, read_enb_0, read_enb_1, read_enb_2, packet_valid;
reg [7:0]datain;
wire [7:0]data_out_0, data_out_1, data_out_2;
wire vld_out_0, vld_out_1, vld_out_2, err, busy;
integer i;

router_top dut(clk, resetn, packet_valid, read_enb_0, read_enb_1,
               read_enb_2,datain,vld_out_0, vld_out_1, vld_out_2,
               err, busy,data_out_0, data_out_1, data_out_2);

//clock generation
initial
begin
    clk = 1;
    forever
    #5 clk=~clk;
end

task initialize;
begin
    read_enb_0 = 1'b0;
    read_enb_1 = 1'b0;
    read_enb_2 = 1'b0;
end
endtask

task reset;
begin
    resetn=1'b0;
    {read_enb_0, read_enb_1, read_enb_2, packet_valid, datain}=0;
    #10;
    resetn=1'b1;
end
endtask
task pktm_gen_8;    // packet generation payload 8
reg [7:0]header, payload_data, parity;
reg [8:0]payloadlen;
begin
    parity=0;
    wait(!busy)
    begin
        @(negedge clk);
        payloadlen=8;
        packet_valid=1'b1;
        header={payloadlen,2'b00};
        datain=header;
        parity=parity^datain;
    end
    @(negedge clk);

    for(i=0;i<payloadlen;i=i+1)
    begin
```

```
        wait(!busy)
        @(negedge clk);
        payload_data={$random}%256;
        datain=payload_data;
        parity=parity^datain;
    end

    wait(!busy)
    @(negedge clk);
    packet_valid=0;
    datain=parity;
    #20
    @(negedge clk);
    read_enb_0=1'b1;
end
endtask

task pktm_gen_14;   // packet generation payload 14
reg [7:0]header, payload_data, parity;
reg [4:0]payloadlen;
begin
    parity=0;
    wait(!busy)
    begin
        @(negedge clk);
        payloadlen=14;
        packet_valid=1'b1;
        header={payloadlen,2'b01};
        datain=header;
        parity=parity^datain;
    end
    @(negedge clk);

    for(i=0;i<payloadlen;i=i+1)
    begin
        wait(!busy)
        @(negedge clk);
        payload_data={$random}%256;
        datain=payload_data;
        parity=parity^datain;
    end

    wait(!busy)
    @(negedge clk);
    packet_valid=0;
    datain=parity;
    #20;
    @(negedge clk);
    read_enb_1=1'b1;
    end
endtask
```

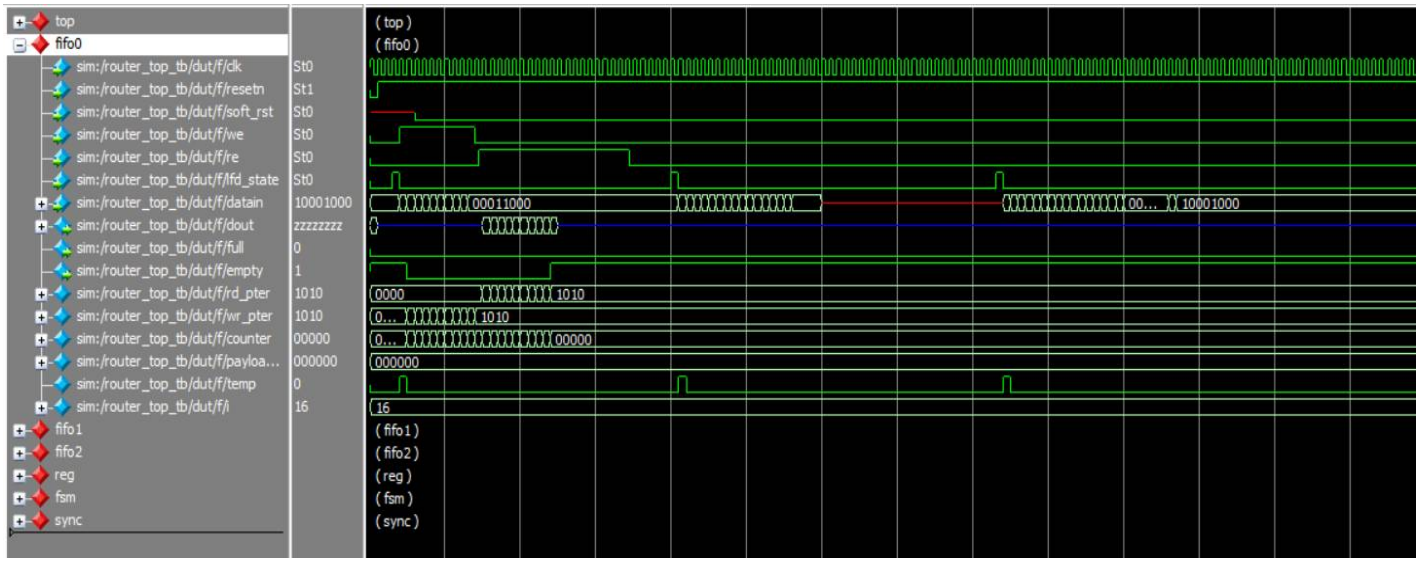
```
task pktm_gen_17;   // packet generation payload 17
reg [7:0]header, payload_data, parity;
reg [4:0]payloadlen;
begin
    parity=0;
    wait(!busy)
    begin
        @(negedge clk);
        payloadlen=17;
        packet_valid=1'b1;
        header={payloadlen,2'b10};
        datain=header;
        parity=parity^datain;
    end
    @(negedge clk);

    for(i=0;i<payloadlen;i=i+1)
    begin
        wait(!busy)
        @(negedge clk);
        payload_data={$random}%256;
        datain=payload_data;
        parity=parity^datain;
    end

    wait(!busy)
    @(negedge clk);
    packet_valid=0;
    datain=parity;
    #20;
    @(negedge clk);
    read_enb_2=1'b1;
    end
endtask

initial
begin
    reset;
    #10;
    pktm_gen_8;
    #200;
    initialize;
    #50;
    pktm_gen_14;
    #200;
    initialize;
    #50;
    pktm_gen_17;
    #200;
    initialize;
    #1000;
    $finish;
end
endmodule
```

Wave form:

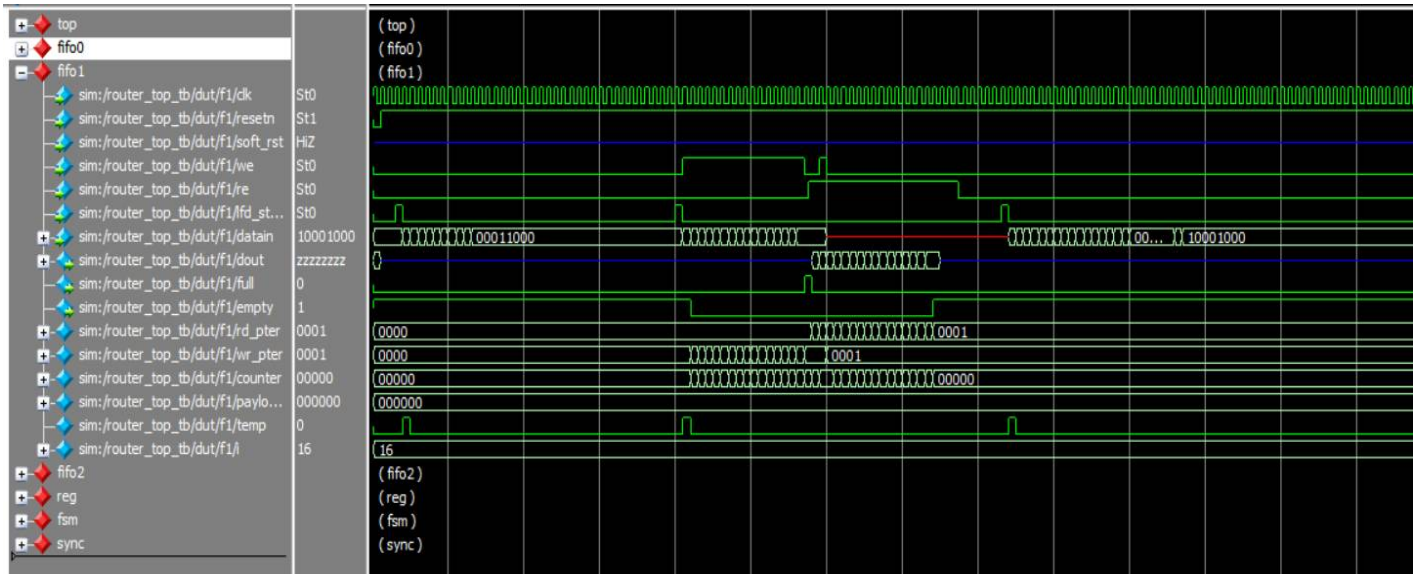
TOP Wave form:



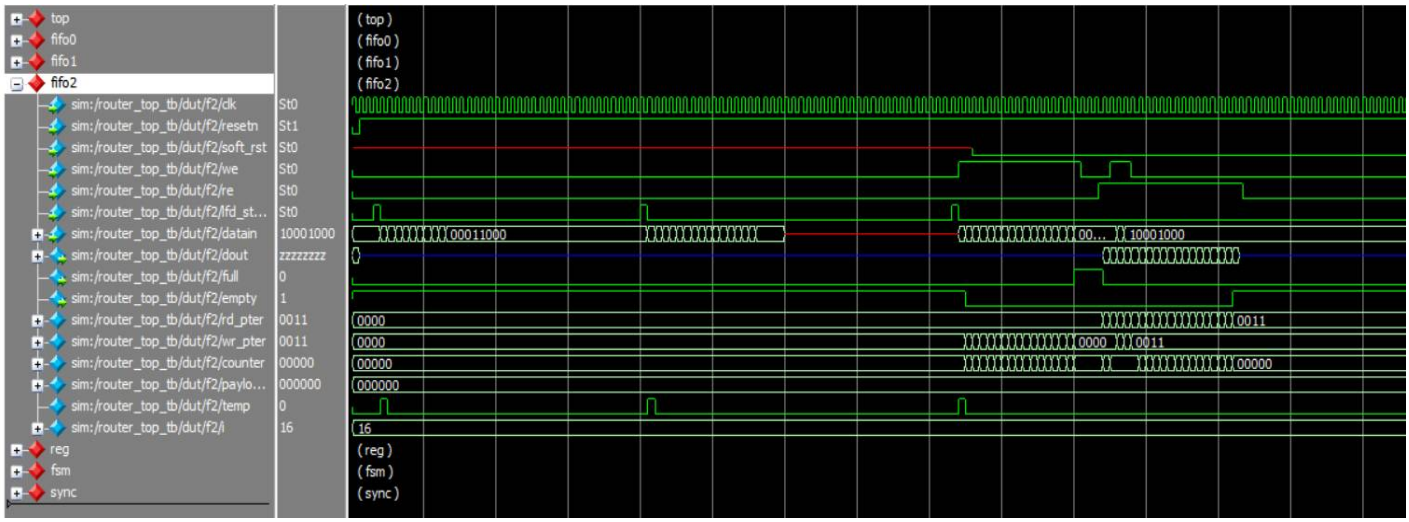
FIFO0 Wave form:



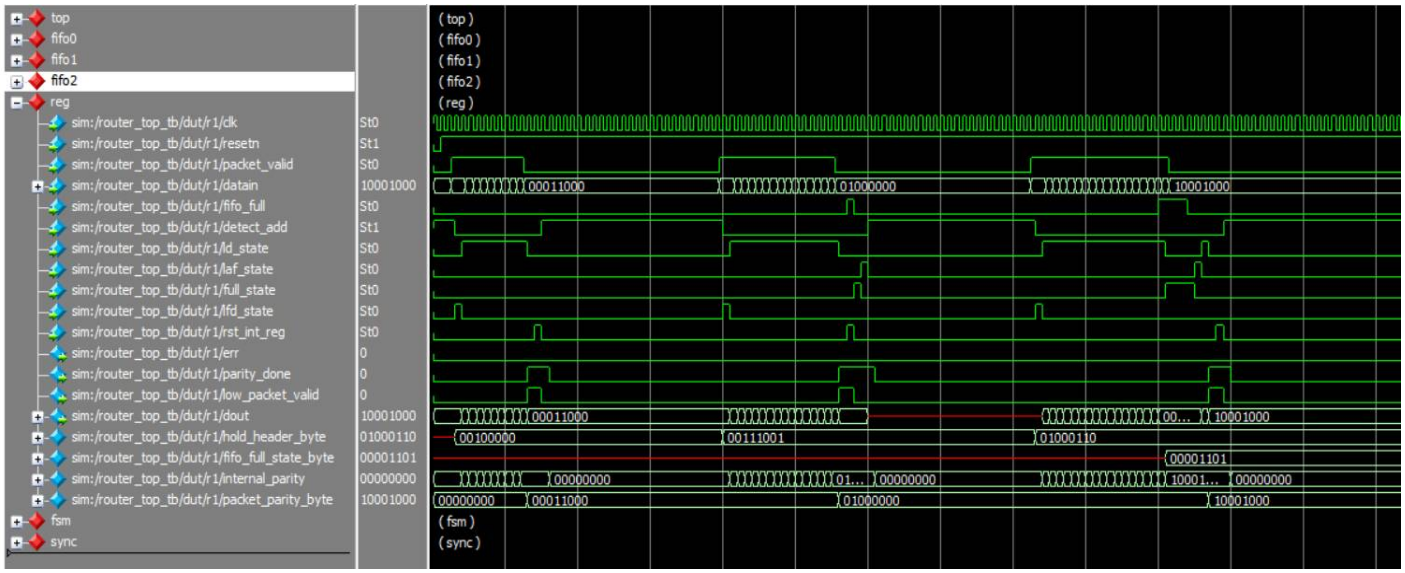
FIFO1 Wave form:



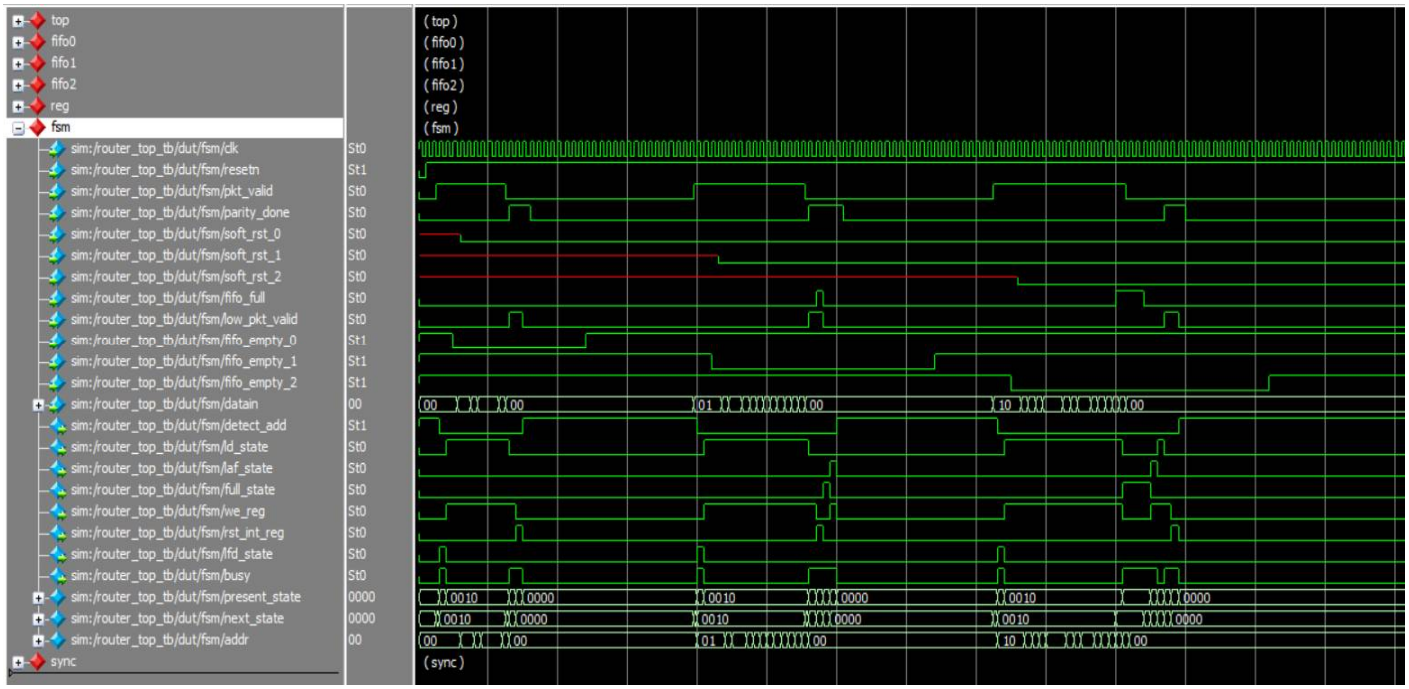
FIFO2 Wave form:



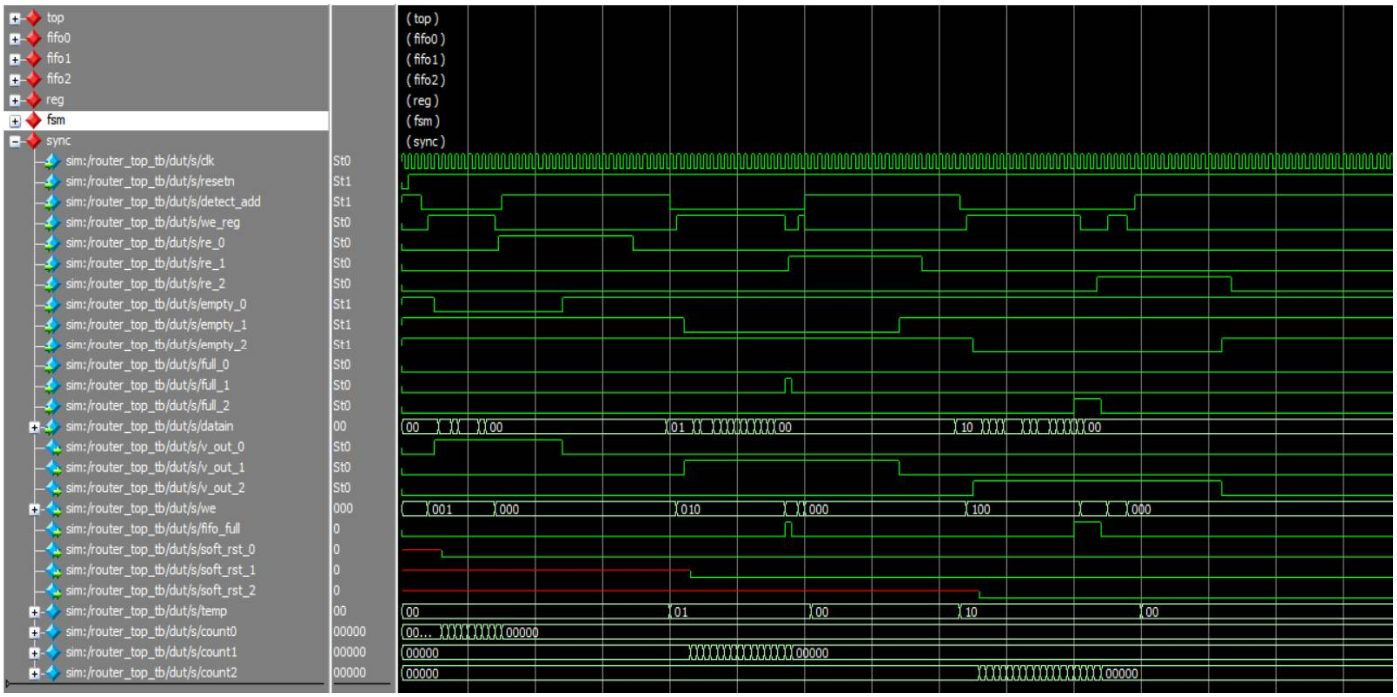
REG Wave form:



FSM Wave form:



SYNC Wave form:



RTL View: