```
`timescale 1ns/1ps
`default_nettype none
module router # ( parameter NumPorts = 4, PassThrough=4'b0011 )
   ( input wire CLK, RST,

     input wire [NumPorts-1:0][63:0]  D,
     input wire [NumPorts-1:0]        D_VALID,
     output wire [NumPorts-1:0]       D_BP,

     output wire [NumPorts-1:0][63:0] Q,
     output wire [NumPorts-1:0]       Q_VALID,
     input wire [NumPorts-1:0]        Q_BP,
     output wire [NumPorts-1:0]       Q_SOF   // used to generate PE_RST
     );

   // RX buffer to router core
   wire [NumPorts-1:0][63:0]          Di;
   wire [NumPorts-1:0]                D_VALIDi, D_BPi, COLi;

   // Passthrough signals
   wire [NumPorts-1:0][63:0]          PT;
   wire [NumPorts-1:0]                PT_VALID;

   generate
      genvar                        i;

      for (i=0; i<NumPorts; i=i+1) begin : rxbuf_gen
         wire [63:0] BUF_D  = PT_VALID[i] ? PT[i] : D[i];
         wire        BUF_DV = D_VALID[i] | PT_VALID[i];

         router_rx_buf rxbuf
             ( .CLK(CLK), .RST(RST),

               .D_VALID(BUF_DV),                    // I
               .D_BP    (D_BP   [i]),               // O
               .D       (BUF_D),    // I

               .Q_VALID(D_VALIDi[i]),               // O
               .Q_BP    (D_BPi   [i]),              // I
               .Q       (Di      [i]),   // O
               .COLLISION(COLi  [i])                // I
               );
      end
   endgenerate

   router_core # (.NumPorts(NumPorts), .PassThrough(PassThrough) ) rc
     ( .CLK(CLK), .RST(RST),

       .D       (Di),          // I
       .D_VALID(D_VALIDi),     // I
       .D_BP    (D_BPi),       // O
       .COLLISION(COLi),       // O

       .PT       (PT), // O
       .PT_VALID (PT_VALID), // O

       .Q       (Q),           // O
       .Q_VALID(Q_VALID),      // O
       .Q_BP    (Q_BP),        // I
       .Q_SOF   (Q_SOF)        // O
      );
```



```
endmodule // router

module router_rx_buf
  ( input wire CLK, RST,

    input wire          D_VALID,
    input wire [63:0]   D,
    output wire         D_BP,

    output wire         Q_VALID,
    output wire [63:0]  Q,
    input wire          Q_BP,
    input wire          COLLISION
   );

   wire [63:0]          FIFO_Q;
   wire                 RD_EN;
   wire                 FIFO_VALID;

   fwft_64x512_afull fwft
     ( .clk(CLK), .srst(RST),
       .din (D), .wr_en(D_VALID), .prog_full(D_BP),
       .dout(FIFO_Q), .valid(FIFO_VALID), .rd_en(RD_EN)
        );


   // EOF detector
   reg [31:0]           Q_TOGO, Q_TOGO_R;
   wire                 EOF = (Q_VALID & Q_TOGO==1);

   always @ (posedge CLK) begin
      if (RST | COLLISION) begin
         Q_TOGO <= 0;
      end else begin
         if (Q_VALID) begin
            if (Q_TOGO==0) begin
               Q_TOGO   <=  (Q[63:56]!=8'h01) ? Q[31:0] : 0;
            end else begin
               Q_TOGO <= Q_TOGO - 1;
            end
         end
      end
   end

   // Write to Backup

   reg                  COLLISION_R;
   reg [63:0]           BUP [0:15];
   reg [3:0]            BUP_WA, BUP_WA_R; // address for 16 words
   reg                  RE_TX;

   always @ (posedge CLK) begin
      COLLISION_R <= COLLISION;

      if (RST) begin
         BUP_WA <= 0;
         BUP_WA_R <= 8'hff;
         RE_TX  <= 0;
      end else begin
         if (EOF) begin
            BUP_WA <= 0;
```

```
            BUP_WA_R <= 8'hff;
            RE_TX  <= 0;
         end else begin
            if (COLLISION)  RE_TX <= 1;

//          if (~COLLISION & Q_VALID & ~RE_TX) begin
            if (~COLLISION_R & COLLISION) begin
               BUP_WA_R <= BUP_WA;
               BUP_WA <= 0;
            end else         if (~COLLISION & Q_VALID & ~&BUP_WA) begin
               BUP_WA <= BUP_WA+1;
               BUP[BUP_WA] <= Q;
            end
         end
      end
   end

   // Read from Backup

   reg [3:0] BUP_RA;

   always @ (posedge CLK) begin
      if (RST) begin
         BUP_RA <= 0;
      end else begin
         if (COLLISION | EOF)
           BUP_RA <= 0;
         else begin
            if (RE_TX) begin
               if(~Q_BP & (BUP_RA != BUP_WA_R)) BUP_RA <= BUP_RA+1;
            end
         end
      end
   end

   wire RE_TX_DONE = RE_TX & (BUP_WA_R==BUP_RA);

   assign RD_EN = ~Q_BP & ~COLLISION & (RE_TX ? RE_TX_DONE : 1);

   assign Q = (RE_TX & ~RE_TX_DONE) ? BUP[BUP_RA]  : FIFO_Q;
   assign Q_VALID = RD_EN & FIFO_VALID | (RE_TX & ~COLLISION & ~Q_BP & BUP_WA_R!
=BUP_RA) ;

endmodule // router_rx_buf

module router_core # ( parameter NumPorts = 4, PassThrough=4'b0011 )
   ( input wire CLK, RST,

     input wire [NumPorts-1:0][63:0]  D,
     input wire [NumPorts-1:0]        D_VALID,
     output wire [NumPorts-1:0]       D_BP,
     output wire [NumPorts-1:0]       COLLISION,

     output wire [NumPorts-1:0][63:0] PT,
     output wire [NumPorts-1:0]       PT_VALID,

     output wire [NumPorts-1:0][63:0] Q,
     output wire [NumPorts-1:0]       Q_VALID,
     input wire [NumPorts-1:0]        Q_BP,
     output wire [NumPorts-1:0]       Q_SOF
     );
```



```
  // - - - - - - - - - - - - - - - - - - - -
  // Crossbar outputs

  wire [NumPorts-1:0][63:0]          CB_Q;
  wire [NumPorts-1:0]                CB_Q_HDR_VALID, CB_Q_PLD_VALID;

  // - - - - - - - - - - - - - - - - - - - -
  // Router sink ports x4

  wire [NumPorts-1:0][7:0]     DEST;
  wire [NumPorts-1:0]          DEST_VALID, SOF, EOF, HDR_VALID, PLD_VALID;
  wire [NumPorts-1:0][63:0]    FRAME;

  reg [NumPorts-1:0][63:0]     D_R;
  reg [NumPorts-1:0]           D_VR;

  always @ (posedge CLK) begin
     D_R <= D;
     D_VR <= D_VALID;
  end

  genvar i;
  generate
     for (i=0; i<NumPorts; i=i+1) begin : router_sink_gen
        wire [63:0] SINK_D;
        wire        SINK_D_VALID;

        // per lane connection signals
        wire [63:0] Dc    = D_R [i];
        wire [63:0] CB_Qc = CB_Q[i];
        wire [63:0] PTc;
        assign PT[i] = PTc;

        // Header pass-through logic
        if (PassThrough[i]) begin
           // pass-through logic before router_sink for PEs
           assign SINK_D       = Dc;
           assign SINK_D_VALID = D_VR[i];
           assign Q_VALID[i]   = CB_Q_PLD_VALID[i];

           assign PTc          = CB_Q_HDR_VALID[i] ? CB_Qc : 0;
           assign PT_VALID[i]  = CB_Q_HDR_VALID[i];
        end else begin // No pass-through ports (PCIe/Aurora)
           assign SINK_D       = Dc;
           assign SINK_D_VALID = D_VR[i];
           assign Q_VALID[i]   = CB_Q_PLD_VALID[i] | CB_Q_HDR_VALID[i];
           assign PT_VALID[i]  = 0;
        end

        router_sink uut_rs
          ( .CLK(CLK), .RST(RST | COLLISION[i]),

            // from source
            .D       (SINK_D),
            .D_VALID(SINK_D_VALID),

            // to matrix
            .DEST(DEST[i]),
            .DEST_VALID(DEST_VALID[i]),

            .SOF(SOF[i]), .EOF(EOF[i]),
            .FRAME(FRAME[i]),
```

```
            .HEADER_VALID (HDR_VALID[i]),
            .PAYLOAD_VALID(PLD_VALID[i])
            );
       end // block: router_sink_gen
   endgenerate

   router_cb # ( .NumPorts(NumPorts) ) cb
     ( .CLK(CLK), .RST(RST),
       .D    ( FRAME ),
       .DEST( DEST ),
       .DEST_VALID(DEST_VALID),
       .D_HDR_VALID(HDR_VALID),
       .D_PLD_VALID(PLD_VALID),
       .D_SOF(SOF),
       .D_EOF(EOF),
       .D_BP (D_BP),
       .COLLISION(COLLISION),

       .Q          (CB_Q),
       .Q_HDR_VALID(CB_Q_HDR_VALID),
       .Q_PLD_VALID(CB_Q_PLD_VALID),
       .Q_SOF      (Q_SOF),
       .Q_EOF      (),
       .Q_BP       (Q_BP)
       );

   assign Q = CB_Q;
   // Q_VALID is under control of PassThrough[x]

endmodule // router_core

module router_cb # ( parameter NumPorts = 4 )
   ( input wire CLK, RST,

     input wire [NumPorts-1:0][63:0]  D,
     input wire [NumPorts-1:0][7:0]   DEST,
     input wire [NumPorts-1:0]        DEST_VALID, D_HDR_VALID, D_PLD_VALID,
     input wire [NumPorts-1:0]        D_SOF, D_EOF,
     output wire [NumPorts-1:0]       D_BP, COLLISION,

     output wire [NumPorts-1:0][63:0] Q,
     output wire [NumPorts-1:0]       Q_HDR_VALID, Q_PLD_VALID, Q_SOF, Q_EOF,
     input wire [NumPorts-1:0]        Q_BP
     );

   wire [NumPorts-1:0][NumPorts-1:0]  D_BPi, COLi;

   genvar                             i;
   generate
      for (i=0; i<NumPorts; i=i+1) begin : mux_gen

         router_mux4 # ( .NumPorts(NumPorts), .PortNo(i+1) ) rm
             ( .CLK(CLK), .RST(RST),
               .D(D),
               .DEST       (DEST),        .DEST_VALID (DEST_VALID),
               .D_HDR_VALID(D_HDR_VALID), .D_PLD_VALID(D_PLD_VALID),
               .D_SOF      (D_SOF),       .D_EOF      (D_EOF),
               .D_BP       (D_BPi[i]),
               .COLLISION  (COLi [i]),

               .Q          (Q          [i]),
               .Q_HDR_VALID(Q_HDR_VALID[i]),
```



```
               .Q_PLD_VALID(Q_PLD_VALID[i]),
               .Q_SOF      (Q_SOF      [i]),
               .Q_EOF      (Q_EOF      [i]),
               .Q_BP       (Q_BP       [i])
               );

      end
   endgenerate

   // OR'ing all D_BPi[]
   //    assign D_BP = D_BPi[0] | D_BPi[1] | D_BPi[2] | D_BPi[3];

   channel_or # ( .Width(NumPorts), .NumCh(NumPorts) ) d_bp_or
     ( .D(D_BPi), .Q(D_BP) );


   channel_or # ( .Width(NumPorts), .NumCh(NumPorts) ) col_or
     ( .D(COLi), .Q(COLLISION) );

endmodule // router_cb


module channel_or # ( parameter Width=64, NumCh=4 )
   ( input wire [Width-1:0][NumCh-1:0] D,
     output wire [Width-1:0] Q );

   wire [Width-1:0][NumCh-1:0] TEMP;

   assign TEMP[0] = D[0];

   generate
      genvar                    i;
      for (i=1; i<NumCh; i=i+1) begin : or_gen
         assign TEMP[i] = D[i] | TEMP[i-1];
      end
   endgenerate

   assign Q = TEMP[NumCh-1];

endmodule // channel_or

module router_mux4 # (parameter NumPorts = 4, PortNo = 1) // 4 to 1 switch
   ( input wire CLK, RST,

     input wire [NumPorts-1:0][63:0] D,
     input wire [NumPorts-1:0][7:0]  DEST,
     input wire [NumPorts-1:0]       DEST_VALID, D_HDR_VALID, D_PLD_VALID,
     input wire [NumPorts-1:0]       D_SOF, D_EOF,
     output wire [NumPorts-1:0]      D_BP, COLLISION,

     output wire [63:0]              Q,
     output wire                     Q_HDR_VALID, Q_PLD_VALID, Q_SOF, Q_EOF,
     input wire                      Q_BP
     );

   // SOF detection

   wire [NumPorts-1:0]               SOF_DETECT;
   reg [NumPorts-1:0]                SOF_DETECTr;

   assign SOF_DETECT[3] = D_SOF[3] & (DEST[3] == PortNo);
   assign SOF_DETECT[2] = D_SOF[2] & (DEST[2] == PortNo);
```

```
   assign SOF_DETECT[1] = D_SOF[1] & (DEST[1] == PortNo);
   assign SOF_DETECT[0] = D_SOF[0] & (DEST[0] == PortNo);

   // Source switching on SOF_DETECT and corresponding EOF

   reg [NumPorts-1:0]                SRC_PORT;
   always @ (posedge CLK) begin
      if (RST) begin
         SRC_PORT <= 0;
         SOF_DETECTr <= 0;
      end else begin
         SOF_DETECTr <= SOF_DETECT;
         if (SRC_PORT==0) begin
            casex (SOF_DETECT)
              4'bxxx1: SRC_PORT <= 4'b0001;
              4'bxx10: SRC_PORT <= 4'b0010;
              4'bx100: SRC_PORT <= 4'b0100;
              4'b1000: SRC_PORT <= 4'b1000;
              default: SRC_PORT <= 0;
            endcase // casex (SOF_DETECT)
         end else begin
            if (SRC_PORT & D_EOF) SRC_PORT <= 0;
         end
      end
   end // always @ (posedge CLK)

   // - - - - - - - - - - - - - - - - - - - -
   // Collision detection

   reg [NumPorts-1:0] CD;
   wire               EOF_DETECT = |(SRC_PORT & D_EOF);

   /*
   always @ (posedge CLK) begin
      if (RST) begin
         CD <= 0;
      end else begin
         CD <= (SRC_PORT==0) ? (SOF_DETECT & ~SRC_PORTi) :  // may be slow...
               (EOF_DETECT ? 0 :  // end of session
                CD | SOF_DETECT); // secondary collision
      end
   end
    */

   reg [2:0]          EOF_CNT;
   wire               EOF_CNT_FULL = &EOF_CNT;

   always @ (posedge CLK) begin
      if (RST) begin
         CD <= 0;
         EOF_CNT <= 0;
      end else begin
         if (EOF_DETECT) EOF_CNT <= 1;
         else EOF_CNT <= (EOF_CNT != 0) ? EOF_CNT+1 : 0;

         if (CD==0) begin
            CD <= (SOF_DETECTr & ~SRC_PORT);
         end else begin
            CD <= EOF_CNT_FULL ? 0 : // end of session
                  CD | SOF_DETECT; // secondary collision
         end
      end
```



```
   end


   assign COLLISION = CD;

   // - - - - - - - - - - - - - - - - - - - -
   // registered output

   reg [63:0] Qi;
   reg        Q_HDR_VALIDi, Q_PLD_VALIDi, Q_SOFi, Q_EOFi;
   reg [3:0]  D_BPi;

   always @ (posedge CLK) begin
      Q_SOFi <= (SRC_PORT==0) & (|SOF_DETECT);
      Q_EOFi <= |(D_EOF & SRC_PORT);
      Q_HDR_VALIDi <= |(D_HDR_VALID & SRC_PORT);
      Q_PLD_VALIDi <= |(D_PLD_VALID & SRC_PORT);

      Qi <= SRC_PORT[3] ? D[3] :
            SRC_PORT[2] ? D[2] :
            SRC_PORT[1] ? D[1] :
            SRC_PORT[0] ? D[0] : 64'h0;

      D_BPi <= Q_BP ? SRC_PORT : 0;
   end

   assign Q     = Qi;
   assign Q_HDR_VALID = Q_HDR_VALIDi;
   assign Q_PLD_VALID = Q_PLD_VALIDi;
   assign Q_SOF = Q_SOFi;
   assign Q_EOF = Q_EOFi;
   assign D_BP  = D_BPi;

endmodule // router_mux4

module router_sink
  ( input wire CLK, RST,

    // Router ports
    input wire [63:0] D,
    input wire        D_VALID,

    output reg [7:0]  DEST,
    output wire       DEST_VALID,

    output reg        SOF, EOF,
    output reg [63:0] FRAME,
    output reg        HEADER_VALID,
    output reg        PAYLOAD_VALID
    );

   // Receiver state machine
   reg [2:0]          STAT, STAT_R; // idle (dest), header, payload
   reg [31:0]         D_TOGO;

   always @ (posedge CLK) begin
      STAT_R <= STAT;

      if (RST) begin
         STAT <= 3'b001;
         SOF <= 0;
```

```
         EOF <= 0;
         D_TOGO <= 0;
      end else begin
         case (STAT)
           3'b001: begin
              EOF <= 0;
              PAYLOAD_VALID <= 0;

              if (D_VALID) begin
                 STAT <= 3'b010;
                 DEST <= D[7:0];
                 SOF <= 1;
              end end

           3'b010: begin
              SOF <= 0;

              if (D_VALID & (D[63:56] != 8'h01)) begin
                 D_TOGO <= D[31:0];
                 STAT <= 3'b100;
              end

              HEADER_VALID <= (D_VALID & (D[63:56] == 8'h01));
              PAYLOAD_VALID <= (D_VALID & (D[63:56] == 8'h00));
              FRAME <= D;
           end

           3'b100: begin
              if (D_VALID) begin
                 D_TOGO <= D_TOGO -1;
                 if (D_TOGO == 1) begin
                    STAT <= 3'b001;
                    EOF <= 1;
                 end
              end

              PAYLOAD_VALID <= D_VALID;
              FRAME <= D;
           end

           default: begin
              STAT <= 3'b001; end
         endcase
      end
   end

   assign DEST_VALID = |STAT[2:1] | STAT_R[2];

endmodule

`default_nettype wire
```