Certainly! Let's break down the code and explain its functionality step by step:

Module Declaration

```verilog
module router_fsm(
  clock, resetn, pkt_valid, data_in, fifo_full, fifo_empty_0, fifo_empty_1, fifo_empty_2,
  soft_reset_0, soft_reset_1, soft_reset_2, parity_done, low_packet_valid,
  write_enb_reg, detect_add, ld_state, laf_state, lfd_state, full_state, rst_int_reg, busy
);
```

This line declares the module `router_fsm` with its input and output ports.

Input and Output Ports

```verilog
input clock, resetn, pkt_valid, fifo_full, fifo_empty_0, fifo_empty_1, fifo_empty_2;
input soft_reset_0, soft_reset_1, soft_reset_2, parity_done, low_packet_valid;
input [1:0] data_in;
output write_enb_reg, detect_add, ld_state, laf_state, lfd_state, full_state, rst_int_reg, busy;
```

- **Inputs:**
  - `clock`: Clock signal.
  - `resetn`: Active-low reset signal.
  - `pkt_valid`: Packet valid signal.
  - `fifo_full`: FIFO full signal.
  - `fifo_empty_0`, `fifo_empty_1`, `fifo_empty_2`: FIFO empty signals for three FIFOs.
  - `soft_reset_0`, `soft_reset_1`, `soft_reset_2`: Soft reset signals for three FIFOs.
  - `parity_done`: Parity check completion signal.

  - `low_packet_valid`: Low packet valid signal.
  - `data_in`: 2-bit input data.

- **Outputs:**
  - `write_enb_reg`: Write enable register.
  - `detect_add`: Detect address signal.
  - `ld_state`: Load state signal.
  - `laf_state`: Load after full state signal.
  - `lfd_state`: Load first data state signal.
  - `full_state`: Full state signal.
  - `rst_int_reg`: Reset internal register signal.
  - `busy`: Busy signal.

Parameters

```verilog
parameter DECODE_ADDRESS     = 3'b000,
          LOAD_FIRST_DATA    = 3'b001,
          LOAD_DATA          = 3'b010,
          WAIT_TILL_EMPTY    = 3'b011,
          CHECK_PARITY_ERROR = 3'b100,
          LOAD_PARITY        = 3'b101,
          FIFO_FULL_STATE    = 3'b110,
          LOAD_AFTER_FULL    = 3'b111;
```

These parameters define the states of the FSM (Finite State Machine).

State Registers

```verilog
```

reg [2:0] PS, NS;

```

- `PS` (Present State): Stores the current state.
- `NS` (Next State): Stores the next state.

### State Transition Logic

#### Present State Update

```verilog
always @(posedge clock)
begin
  if (!resetn)
    PS <= DECODE_ADDRESS;
  else if (soft_reset_0 || soft_reset_1 || soft_reset_2)
    PS <= DECODE_ADDRESS;
  else
    PS <= NS;
end
```

This block updates the present state (`PS`) based on the clock signal, reset signals, and soft reset signals.

Next State Logic

```verilog
always @(*)
begin
  NS = DECODE_ADDRESS;
```
```

```
case (PS)

  DECODE_ADDRESS :
    begin
      if ((pkt_valid && (data_in[1:0] == 0) && fifo_empty_0) ||
          (pkt_valid && (data_in[1:0] == 1) && fifo_empty_1) ||
          (pkt_valid && (data_in[1:0] == 2) && fifo_empty_2))
        NS = LOAD_FIRST_DATA;

      else if ((pkt_valid && (data_in[1:0] == 0) && (~fifo_empty_0)) ||
               (pkt_valid && (data_in[1:0] == 1) && (~fifo_empty_1)) ||
               (pkt_valid && (data_in[1:0] == 2) && (~fifo_empty_2)))
        NS = WAIT_TILL_EMPTY;

      else
        NS = DECODE_ADDRESS;
    end

  LOAD_FIRST_DATA :  NS = LOAD_DATA;

  LOAD_DATA :
    begin
      if (fifo_full)
        NS = FIFO_FULL_STATE;
      else if (!fifo_full && !pkt_valid)
        NS = LOAD_PARITY;
      else
        NS = LOAD_DATA;
    end

  WAIT_TILL_EMPTY :
```

```
  begin
    if ((!fifo_empty_0) || (!fifo_empty_1) || (!fifo_empty_2))
      NS = WAIT_TILL_EMPTY;
    else if (fifo_empty_0 || fifo_empty_1 || fifo_empty_2)
      NS = LOAD_FIRST_DATA;
    else
      NS = WAIT_TILL_EMPTY;
  end

CHECK_PARITY_ERROR :
  begin
    if (fifo_full)
      NS = FIFO_FULL_STATE;
    else
      NS = DECODE_ADDRESS;
  end

LOAD_PARITY : NS = CHECK_PARITY_ERROR;

FIFO_FULL_STATE :
  begin
    if (!fifo_full)
      NS = LOAD_AFTER_FULL;
    else if (fifo_full)
      NS = FIFO_FULL_STATE;
  end

LOAD_AFTER_FULL :
  begin
    if ((!parity_done) && (!low_packet_valid))
      NS = LOAD_DATA;
```

```
      else if ((!parity_done) && (low_packet_valid))
        NS = LOAD_PARITY;
      else if (parity_done)
        NS = DECODE_ADDRESS;
    end

  endcase
end
```

This block determines the next state (`NS`) based on the current state (`PS`) and input signals.

Output Logic

```verilog
assign detect_add = (PS == DECODE_ADDRESS) ? 1 : 0;
assign write_enb_reg = (PS == LOAD_DATA || PS == LOAD_PARITY || PS == LOAD_AFTER_FULL) ? 1 : 0;
assign full_state = (PS == FIFO_FULL_STATE) ? 1 : 0;
assign lfd_state = (PS == LOAD_FIRST_DATA) ? 1 : 0;
assign busy = (PS == FIFO_FULL_STATE || PS == LOAD_AFTER_FULL || PS == WAIT_TILL_EMPTY ||
          PS == LOAD_FIRST_DATA || PS == LOAD_PARITY || PS == CHECK_PARITY_ERROR) ? 1 : 0;
assign ld_state = (PS == LOAD_DATA) ? 1 : 0;
assign laf_state = (PS == LOAD_AFTER_FULL) ? 1 : 0;
assign rst_int_reg = (PS == CHECK_PARITY_ERROR) ? 1 : 0;
```

These lines generate the output signals based on the current state (`PS`).

State Descriptions

- **DECODE_ADDRESS:** Decodes the address to determine which FIFO to use based on `data_in` and emptiness status.

- **LOAD_FIRST_DATA:** Loads the first data byte.

- **LOAD_DATA:** Loads the data bytes into the FIFO.

- **WAIT_TILL_EMPTY:** Waits until the FIFO becomes empty.

- **CHECK_PARITY_ERROR:** Checks for parity errors.

- **LOAD_PARITY:** Loads the parity byte.

- **FIFO_FULL_STATE:** Handles the FIFO full condition.

- **LOAD_AFTER_FULL:** Loads data after the FIFO is full.

This FSM manages the flow of data into and out of the FIFOs, ensuring proper loading, checking for errors, and handling full and empty conditions.

Fifo

Certainly! Let's break down the code and explain its functionality:

Module Declaration

```verilog
module router_fifo(clock, resetn, soft_reset, write_enb, read_enb, lfd_state, data_in, full, empty, data_out);
```

This line declares the module `router_fifo` with its input and output ports.

Input and Output Ports

```verilog
input clock, resetn, soft_reset;
input write_enb, read_enb, lfd_state;
```

input [7:0] data_in;

output reg [7:0] data_out;

output full, empty;

```

- **Inputs:**
  - `clock`: Clock signal.
  - `resetn`: Active-low reset signal.
  - `soft_reset`: Soft reset signal.
  - `write_enb`: Write enable signal.
  - `read_enb`: Read enable signal.
  - `lfd_state`: State indicating the presence of the last frame delimiter.
  - `data_in`: 8-bit input data.

- **Outputs:**
  - `data_out`: 8-bit output data.
  - `full`: Signal indicating the FIFO is full.
  - `empty`: Signal indicating the FIFO is empty.

### Internal Registers and Memory

```verilog
reg [4:0] rd_pointer, wr_pointer;

reg [6:0] count;

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

integer i;

reg lfd_state_t;

```
```

- `rd_pointer`, `wr_pointer`: 5-bit read and write pointers.

- `count`: 7-bit counter to track the number of data elements.

- `mem`: 16x9 memory to store data. The extra bit is used to store the last frame delimiter (LFD) state.

- `i`: Integer variable for loops.

- `lfd_state_t`: Temporary register to store the LFD state.

LFD State Capture

```verilog
always @(posedge clock)
begin
  if (!resetn)
    lfd_state_t <= 0;
  else
    lfd_state_t <= lfd_state;
end
```

This block captures the LFD state on every positive edge of the clock. The state is reset when `resetn` is low.

Read Operation

```verilog
always @(posedge clock)
begin
  if (!resetn)
    data_out <= 8'b0;
  else if (soft_reset)
    data_out <= 8'bz;
  else if (read_enb && !empty)
```

```
    data_out <= mem[rd_pointer[3:0]][7:0];
  else if (count == 0)
    data_out <= 8'bz;
end
```

This block handles the read operation. It sets `data_out` to zero on reset, high-impedance (`8'bz`) on soft reset, and outputs data from memory if `read_enb` is asserted and the FIFO is not empty.

Write Operation

```verilog
always @(posedge clock)
begin
  if (!resetn || soft_reset)
  begin
    for (i = 0; i < 16; i = i + 1)
      mem[i] <= 0;
  end
  else if (write_enb && !full)
  begin
    if (lfd_state_t)
    begin
      mem[wr_pointer[3:0]][8] <= 1'b1;
      mem[wr_pointer[3:0]][7:0] <= data_in;
    end
    else
    begin
      mem[wr_pointer[3:0]][8] <= 1'b0;
      mem[wr_pointer[3:0]][7:0] <= data_in;
    end
```

```
  end
end
```

This block handles the write operation. It resets the memory on reset or soft reset. If `write_enb` is asserted and the FIFO is not full, it writes `data_in` to the memory. The MSB of the memory element stores the LFD state.

Pointer Generation

Write Pointer

```verilog
always @(posedge clock)
begin
  if (!resetn)
    wr_pointer <= 0;
  else if (write_enb && !full)
    wr_pointer <= wr_pointer + 1;
end
```

This block increments the write pointer on every write operation if the FIFO is not full.

Read Pointer

```verilog
always @(posedge clock)
begin
  if (!resetn)
    rd_pointer <= 0;
  else if (read_enb && !empty)
```

```
    rd_pointer <= rd_pointer + 1;
end
```

This block increments the read pointer on every read operation if the FIFO is not empty.

Counter Block

```verilog
always @(posedge clock)
begin
  if (read_enb && !empty)
  begin
    if ((mem[rd_pointer[3:0]][8]) == 1'b1)
      count <= mem[rd_pointer[3:0]][7:2] + 1'b1;
    else if (count != 0)
      count <= count - 1'b1;
  end
end
```

This block manages the `count` register. It updates the count based on the memory element being read.

Full and Empty Conditions

```verilog
assign full  = (wr_pointer == {~rd_pointer[4], rd_pointer[3:0]});
assign empty = (rd_pointer == wr_pointer);
```

These lines determine the full and empty conditions of the FIFO:

- The FIFO is full when the write pointer equals the inverted read pointer.

- The FIFO is empty when the read pointer equals the write pointer.

This FIFO design handles read and write operations with the additional functionality of storing the last frame delimiter (LFD) state in the MSB of each memory element.

Register

Certainly! Let's break down the `router_sync` module and explain its functionality step by step:

Module Declaration

```verilog
module router_sync(
  clock, resetn, data_in, detect_add, full_0, full_1, full_2, empty_0, empty_1, empty_2,
  write_enb_reg, read_enb_0, read_enb_1, read_enb_2, write_enb, fifo_full,
  vld_out_0, vld_out_1, vld_out_2, soft_reset_0, soft_reset_1, soft_reset_2
);
```

This line declares the module `router_sync` with its input and output ports.

Input and Output Ports

```verilog
input clock, resetn, detect_add, full_0, full_1, full_2, empty_0, empty_1, empty_2;
input write_enb_reg, read_enb_0, read_enb_1, read_enb_2;
input [1:0] data_in;
output reg [2:0] write_enb;
```

output reg fifo_full, soft_reset_0, soft_reset_1, soft_reset_2;

output vld_out_0, vld_out_1, vld_out_2;

```

- **Inputs:**
  - `clock`: Clock signal.
  - `resetn`: Active-low reset signal.
  - `detect_add`: Signal to detect the address.
  - `full_0`, `full_1`, `full_2`: Signals indicating if the three FIFOs are full.
  - `empty_0`, `empty_1`, `empty_2`: Signals indicating if the three FIFOs are empty.
  - `write_enb_reg`: Write enable register signal.
  - `read_enb_0`, `read_enb_1`, `read_enb_2`: Read enable signals for the three FIFOs.
  - `data_in`: 2-bit input data.

- **Outputs:**
  - `write_enb`: 3-bit write enable signal for the three FIFOs.
  - `fifo_full`: Signal indicating if the selected FIFO is full.
  - `soft_reset_0`, `soft_reset_1`, `soft_reset_2`: Soft reset signals for the three FIFOs.
  - `vld_out_0`, `vld_out_1`, `vld_out_2`: Valid output signals for the three FIFOs.

### Internal Registers

```verilog
reg [1:0] data_in_tmp;
reg [4:0] count0, count1, count2;
```

- `data_in_tmp`: 2-bit temporary register to store the input data.
- `count0`, `count1`, `count2`: 5-bit counters for the three FIFOs.

Data Input Storage
```

```verilog
always @(posedge clock)
begin
  if (~resetn)
    data_in_tmp <= 0;
  else if (detect_add)
    data_in_tmp <= data_in;
end
```

This block stores the input data into `data_in_tmp` on the rising edge of the clock when `detect_add` is asserted. It resets `data_in_tmp` when `resetn` is low.

Address Decoding and FIFO Full

```verilog
always @(*)
begin
  case (data_in_tmp)
    2'b00: begin
      fifo_full <= full_0;
      if (write_enb_reg)
        write_enb <= 3'b001;
      else
        write_enb <= 0;
    end
    2'b01: begin
      fifo_full <= full_1;
      if (write_enb_reg)
        write_enb <= 3'b010;
```

```
      else
        write_enb <= 0;
    end
    2'b10: begin
      fifo_full <= full_2;
      if (write_enb_reg)
        write_enb <= 3'b100;
      else
        write_enb <= 0;
    end
    default: begin
      fifo_full <= 0;
      write_enb <= 0;
    end
  endcase
end
```

This block decodes the `data_in_tmp` to determine which FIFO to access. It sets the `fifo_full` signal and `write_enb` signal based on the input data and `write_enb_reg`.

Valid Byte Block

```verilog
assign vld_out_0 = (~empty_0);
assign vld_out_1 = (~empty_1);
assign vld_out_2 = (~empty_2);
```

These lines generate the valid output signals for the three FIFOs. If a FIFO is not empty, the corresponding valid output signal is asserted.

Soft Reset Block

There are three soft reset blocks, one for each FIFO.

FIFO 0 Soft Reset

```verilog
always @(posedge clock)
begin
 if (~resetn)
 begin
  count0 <= 0;
  soft_reset_0 <= 0;
 end
 else if (vld_out_0)
 begin
  if (~read_enb_0)
  begin
   if (count0 == 29)
   begin
    soft_reset_0 <= 1'b1;
    count0 <= 0;
   end
   else
   begin
    soft_reset_0 <= 1'b0;
    count0 <= count0 + 1'b1;
   end
  end
  else
   count0 <= 0;
```

```
  end
end
```

FIFO 1 Soft Reset

```verilog
always @(posedge clock)
begin
  if (~resetn)
  begin
    count1 <= 0;
    soft_reset_1 <= 0;
  end
  else if (vld_out_1)
  begin
    if (~read_enb_1)
    begin
      if (count1 == 29)
      begin
        soft_reset_1 <= 1'b1;
        count1 <= 0;
      end
      else
      begin
        soft_reset_1 <= 1'b0;
        count1 <= count1 + 1'b1;
      end
    end
    else
      count1 <= 0;
```

```
  end
end
```

FIFO 2 Soft Reset

```verilog
always @(posedge clock)
begin
  if (~resetn)
  begin
    count2 <= 0;
    soft_reset_2 <= 0;
  end
  else if (vld_out_2)
  begin
    if (~read_enb_2)
    begin
      if (count2 == 29)
      begin
        soft_reset_2 <= 1'b1;
        count2 <= 0;
      end
      else
      begin
        soft_reset_2 <= 1'b0;
        count2 <= count2 + 1'b1;
      end
    end
    else
      count2 <= 0;
```

```
  end
end
```

These blocks generate the soft reset signals for the three FIFOs. If a FIFO is valid and not being read, the corresponding counter increments. If the counter reaches 29, the soft reset signal is asserted, and the counter is reset.

This module manages the synchronization of data and control signals between multiple FIFOs, ensuring proper read/write operations, valid signal generation, and soft reset handling.

Synchornizer

Sure! Let's go through the `router_sync` module line by line to understand its functionality:

Module Declaration

```verilog
module router_sync(
  clock, resetn, data_in, detect_add, full_0, full_1, full_2, empty_0, empty_1, empty_2,
  write_enb_reg, read_enb_0, read_enb_1, read_enb_2, write_enb, fifo_full,
  vld_out_0, vld_out_1, vld_out_2, soft_reset_0, soft_reset_1, soft_reset_2
);
```

This line declares the module `router_sync` with its input and output ports.

Input and Output Ports

```verilog
input clock, resetn, detect_add, full_0, full_1, full_2, empty_0, empty_1, empty_2;
input write_enb_reg, read_enb_0, read_enb_1, read_enb_2;
input [1:0] data_in;
```

output reg [2:0] write_enb;

output reg fifo_full, soft_reset_0, soft_reset_1, soft_reset_2;

output vld_out_0, vld_out_1, vld_out_2;

```

- **Inputs:**
  - `clock`: Clock signal.
  - `resetn`: Active-low reset signal.
  - `detect_add`: Signal to detect the address.
  - `full_0`, `full_1`, `full_2`: Signals indicating if the three FIFOs are full.
  - `empty_0`, `empty_1`, `empty_2`: Signals indicating if the three FIFOs are empty.
  - `write_enb_reg`: Write enable register signal.
  - `read_enb_0`, `read_enb_1`, `read_enb_2`: Read enable signals for the three FIFOs.
  - `data_in`: 2-bit input data.

- **Outputs:**
  - `write_enb`: 3-bit write enable signal for the three FIFOs.
  - `fifo_full`: Signal indicating if the selected FIFO is full.
  - `soft_reset_0`, `soft_reset_1`, `soft_reset_2`: Soft reset signals for the three FIFOs.
  - `vld_out_0`, `vld_out_1`, `vld_out_2`: Valid output signals for the three FIFOs.

### Internal Registers

```verilog
reg [1:0] data_in_tmp;
reg [4:0] count0, count1, count2;
```

- `data_in_tmp`: 2-bit temporary register to store the input data.
- `count0`, `count1`, `count2`: 5-bit counters for the three FIFOs.
```

Data Input Storage

```verilog
always @(posedge clock)
begin
  if (~resetn)
    data_in_tmp <= 0;
  else if (detect_add)
    data_in_tmp <= data_in;
end
```

This `always` block stores the input data into `data_in_tmp` on the rising edge of the clock when `detect_add` is asserted. It resets `data_in_tmp` when `resetn` is low.

Address Decoding and FIFO Full

```verilog
always @(*)
begin
  case (data_in_tmp)
    2'b00: begin
      fifo_full <= full_0;
      if (write_enb_reg)
        write_enb <= 3'b001;
      else
        write_enb <= 0;
    end
    2'b01: begin
      fifo_full <= full_1;
      if (write_enb_reg)
```

```
      write_enb <= 3'b010;
    else
      write_enb <= 0;
  end
  2'b10: begin
    fifo_full <= full_2;
    if (write_enb_reg)
      write_enb <= 3'b100;
    else
      write_enb <= 0;
  end
  default: begin
    fifo_full <= 0;
    write_enb <= 0;
  end
 endcase
end
```

This `always` block decodes the `data_in_tmp` to determine which FIFO to access. It sets the `fifo_full` signal and `write_enb` signal based on the input data and `write_enb_reg`.

Valid Byte Block

```verilog
assign vld_out_0 = (~empty_0);
assign vld_out_1 = (~empty_1);
assign vld_out_2 = (~empty_2);
```

These lines generate the valid output signals for the three FIFOs. If a FIFO is not empty, the corresponding valid output signal is asserted.

Soft Reset Block

There are three soft reset blocks, one for each FIFO.

FIFO 0 Soft Reset

```verilog
always @(posedge clock)
begin
  if (~resetn)
  begin
    count0 <= 0;
    soft_reset_0 <= 0;
  end
  else if (vld_out_0)
  begin
    if (~read_enb_0)
    begin
      if (count0 == 29)
      begin
        soft_reset_0 <= 1'b1;
        count0 <= 0;
      end
      else
      begin
        soft_reset_0 <= 1'b0;
        count0 <= count0 + 1'b1;
      end
    end
    else
```

```
    count0 <= 0;
  end
end
```

FIFO 1 Soft Reset

```verilog
always @(posedge clock)
begin
  if (~resetn)
  begin
    count1 <= 0;
    soft_reset_1 <= 0;
  end
  else if (vld_out_1)
  begin
    if (~read_enb_1)
    begin
      if (count1 == 29)
      begin
        soft_reset_1 <= 1'b1;
        count1 <= 0;
      end
      else
      begin
        soft_reset_1 <= 1'b0;
        count1 <= count1 + 1'b1;
      end
    end
    else
```

```
    count1 <= 0;
  end
end
```

FIFO 2 Soft Reset

```verilog
always @(posedge clock)
begin
  if (~resetn)
  begin
    count2 <= 0;
    soft_reset_2 <= 0;
  end
  else if (vld_out_2)
  begin
    if (~read_enb_2)
    begin
      if (count2 == 29)
      begin
        soft_reset_2 <= 1'b1;
        count2 <= 0;
      end
      else
      begin
        soft_reset_2 <= 1'b0;
        count2 <= count2 + 1'b1;
      end
    end
    else
```

```
    count2 <= 0;
  end
end
```

These blocks generate the soft reset signals for the three FIFOs. If a FIFO is valid and not being read, the corresponding counter increments. If the counter reaches 29, the soft reset signal is asserted, and the counter is reset.

Summary

- The `data_in_tmp` register stores the address from `data_in`.
- The address decoding logic sets the `fifo_full` and `write_enb` signals based on the selected FIFO.
- The valid byte logic determines if there is data in the FIFOs.
- The soft reset logic ensures that if data is not read from a FIFO for 30 cycles, the FIFO is reset.

This module manages the synchronization of data and control signals between multiple FIFOs, ensuring proper read/write operations, valid signal generation, and soft reset handling.

Register

Let's go through the `router_reg` module line by line to understand its functionality:

Module Declaration

```verilog
module router_reg(
  clock, resetn, pkt_valid, data_in, fifo_full, detect_add,
  ld_state, laf_state, full_state, lfd_state, rst_int_reg, err,
  parity_done, low_packet_valid, dout
);
```

This line declares the module `router_reg` with its input and output ports.

Input and Output Ports

```verilog
input clock, resetn, pkt_valid, fifo_full, detect_add, ld_state, laf_state, full_state, lfd_state, rst_int_reg;
input [7:0] data_in;
output reg err, parity_done, low_packet_valid;
output reg [7:0] dout;
```

- **Inputs:**
  - `clock`: Clock signal.
  - `resetn`: Active-low reset signal.
  - `pkt_valid`: Packet valid signal.
  - `fifo_full`: FIFO full signal.
  - `detect_add`: Detect address signal.
  - `ld_state`: Load state signal.
  - `laf_state`: Load after full state signal.
  - `full_state`: Full state signal.
  - `lfd_state`: Load first data state signal.
  - `rst_int_reg`: Reset internal register signal.
  - `data_in`: 8-bit input data.

- **Outputs:**
  - `err`: Error signal.
  - `parity_done`: Parity done signal.
  - `low_packet_valid`: Low packet valid signal.
  - `dout`: 8-bit data output.

Internal Registers

```verilog
reg [7:0] header, int_reg, int_parity, ext_parity;
```

- `header`: Register to store the packet header.
- `int_reg`: Internal register to store data temporarily.
- `int_parity`: Register to store the calculated internal parity.
- `ext_parity`: Register to store the external parity.

Data Out Logic

```verilog
always @(posedge clock)
begin
 if (!resetn)
 begin
  dout <= 0;
  header <= 0;
  int_reg <= 0;
 end
 else if (detect_add && pkt_valid && data_in[1:0] != 2'b11)
  header <= data_in;
 else if (lfd_state)
  dout <= header;
 else if (ld_state && !fifo_full)
  dout <= data_in;
 else if (ld_state && fifo_full)
  int_reg <= data_in;
 else if (laf_state)
```

```
    dout <= int_reg;
end
```

This `always` block updates the `dout` register based on various states and conditions. It also stores the header and intermediate data in the `header` and `int_reg` registers, respectively.

Low Packet Valid Logic

```verilog
always @(posedge clock)
begin
  if (!resetn)
    low_packet_valid <= 0;
  else if (rst_int_reg)
    low_packet_valid <= 0;
  else if (ld_state && !pkt_valid)
    low_packet_valid <= 1;
end
```

This `always` block updates the `low_packet_valid` signal based on the reset signal and `ld_state`.

Parity Done Logic

```verilog
always @(posedge clock)
begin
  if (!resetn)
    parity_done <= 0;
  else if (detect_add)
```

```
    parity_done <= 0;
  else if ((ld_state && !fifo_full && !pkt_valid) || (laf_state && low_packet_valid && !parity_done))
    parity_done <= 1;
end
```

This `always` block updates the `parity_done` signal based on various states and conditions.

Parity Calculate Logic

```verilog
always @(posedge clock)
begin
  if (!resetn)
    int_parity <= 0;
  else if (detect_add)
    int_parity <= 0;
  else if (lfd_state && pkt_valid)
    int_parity <= int_parity ^ header;
  else if (ld_state && pkt_valid && !full_state)
    int_parity <= int_parity ^ data_in;
  else
    int_parity <= int_parity;
end
```

This `always` block calculates the internal parity by XORing the `header` and `data_in` values.

Error Logic

```verilog
```

```
always @(posedge clock)
begin
 if (!resetn)
  err <= 0;
 else if (parity_done)
 begin
  if (int_parity == ext_parity)
   err <= 0;
  else
   err <= 1;
 end
 else
  err <= 0;
end
```

This `always` block updates the `err` signal based on the parity comparison between `int_parity` and `ext_parity`.

External Parity Logic

```verilog
always @(posedge clock)
begin
 if (!resetn)
  ext_parity <= 0;
 else if (detect_add)
  ext_parity <= 0;
 else if ((ld_state && !fifo_full && !pkt_valid) || (laf_state && !parity_done && low_packet_valid))
  ext_parity <= data_in;
end
```

```

This `always` block updates the `ext_parity` register with the `data_in` value under specific conditions.

### Summary

- **Data Out Logic:** Handles the output data based on various states.
- **Low Packet Valid Logic:** Manages the `low_packet_valid` signal based on reset and load state conditions.
- **Parity Done Logic:** Updates the `parity_done` signal based on packet and load states.
- **Parity Calculate Logic:** Computes the internal parity by XORing header and data values.
- **Error Logic:** Compares internal and external parity to set the error signal.
- **External Parity Logic:** Updates the external parity register with the input data.

This module manages data output, error detection, and parity calculations to ensure correct packet handling in the router system.
```