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.