Danmarks Tekniske Universitet

Ethernet Switch Final Report

34349: FPGA design for Communication Systems
Spring 2025

Group

Dimitrios Vlachos - s243192
Theodoros Pontzouktzidis - s250239

Contents

1 Tools and Design Requirements **1**

2 Introduction **2**

3 Ethernet Switch Overview **4**
3.1 Input Unit 4
3.2 MAC Learning Module 4
3.3 Switch Fabric and Output Unit 5
3.4 Diagram 6

4 Ethernet Switch Functional Breakdown **7**
4.1 Input Unit 7
4.1.1 Parallel FCS Implementation 7
4.1.2 SOF and EOF Generation Module 8
4.1.3 Input FSM Module 8
4.1.4 FIFO for Source and Destination MAC Addresses 9
4.2 MAC Learning Module 9
4.2.1 Simple XOR Hashing Algorithm 10
4.2.2 MAC Finite State Machine (FSM) 10
4.3 Switch Fabric and Output Unit 11
4.3.1 HOL Blocking Avoidance 11
4.3.2 FIFO Buffering and States 11
4.3.3 Input and Output Control Signals 11
4.3.4 Multicast and Broadcast Support 12
4.3.5 Output Unit 12
4.3.6 Round-Robin Arbiter 12
4.3.7 Interaction Between Output Unit and Arbiter 13

5 Modules specification **14**
5.1 Input Module 14
5.2 FCS Module 16
5.3 MAC Module 17
5.4 Crossbar Module 19
5.5 Arbiter Module 20
5.6 Output Module 21

6 Simulation **23**
6.1 Test Input Unit 23
6.2 Test MAC Learn Unit 23
6.3 Test Crossbar with Output Unit 24
6.4 Test Output Unit with Arbiter 25
6.5 Test with Same Packet in All Ports 25
6.6 Test with Bad Packet (FCS Failure) 26

7 Conclusion **28**

8 Distribution of Work **29**

9 Appendix **30**
9.1 Ethernet Switch Package 30
9.2 Synchronous FIFO 31
9.3 Ethernet Switch 34
9.4 Input Unit 37
9.5 Input Unit FCS check 40
9.6 Input Unit SOF EOF 43
9.7 Input Unit FSM 44
9.8 Input Unit Address FIFO 47
9.9 Input Unit Length FIFO 49
9.10 MAC Learning Module 51
9.11 Crossbar 55
9.12 Output Unit 58
9.13 Output Unit FSM 59
9.14 Arbiter 61
9.15 Test Bench 62

References **65**

1 Tools and Design Requirements

The proposed Ethernet switch architecture was implemented using Xilinx Vivado as the primary design, simulation, and synthesis tool. The design follows a modular and pipelined approach targeting a 4-port Gigabit Ethernet switch. It was required to support up to 8K MAC addresses stored in a MAC address table, handle Ethernet frame sizes ranging from 64 bytes to 1518 bytes (excluding the preamble and Inter-Frame Gap), and interface through a GMII (Gigabit Media Independent Interface) standard. The switch must be non-blocking, ensuring that it can forward traffic on all ports simultaneously without packet loss. Additionally, it must preserve the order of frames within the same traffic flow, preventing any reordering. A complete SystemVerilog testbench was developed to verify the implemented functionality, including frame parsing, MAC learning, switching logic, and output buffering. The implementation was primarily done in SystemVerilog, while a VHDL-based parallel Frame Check Sequence (FCS) checker module was reused from Exercise 1 to ensure proper CRC validation of incoming frames.

2 Introduction

The Ethernet frame is received in octets and it is formatted like shown in Figure 1:

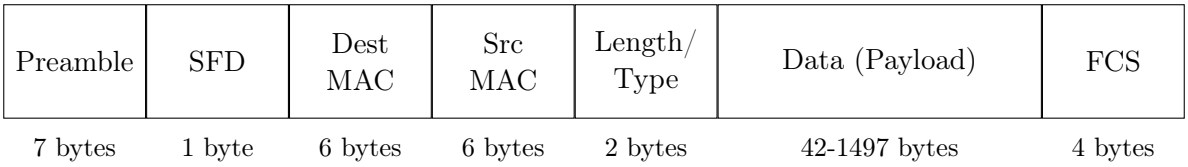


Figure 1: Ethernet 802.3 Raw Frame Format

In Ethernet-based switch designs, frame processing begins at the input unit, where incoming frames are received serially in octets and parsed according to the standard Ethernet frame format, as shown in Figure 1. A critical component of this stage is the **Frame Check Sequence (FCS)**, a 32-bit cyclic redundancy check used for error detection. In our design, a dedicated parallel FCS check module validates each frame in real-time, ensuring data integrity without introducing latency in the processing pipeline.

After the FCS check, the frame enters the **MAC learning module**. This module extracts both the source and destination MAC addresses and uses a hashing mechanism to index the source MAC into a dedicated MAC address table stored in RAM. If the destination MAC address is already known, the frame is forwarded to the corresponding output port. Otherwise, it is broadcasted to all other ports. This behavior allows the switch to dynamically learn the network topology and efficiently direct traffic.

The validated and classified frames are then routed through the **switch fabric**, which interconnects input and output ports. To handle traffic contention and prevent packet loss, each output port is equipped with a FIFO-based **output buffer**. These buffers decouple ingress and egress operations, enabling non-blocking operation under high-load scenarios and preserving packet ordering[1]. In Table 1 are the descriptions of each field:

Field	**Size**	**Description**
Preamble	7 bytes	Pattern of alternating 1s and 0s (10101010...) that allows devices to synchronize
SFD (Start Frame Delimiter)	1 byte	10101011 - indicates the start of the frame
Destination MAC Address	6 bytes	Physical address of the destination device
Source MAC Address	6 bytes	Physical address of the source device
Length/Type	2 bytes	Indicates either the length of the data or the protocol type
Data (Payload)	42-1497 bytes	The actual data being transmitted
FCS (Frame Check Sequence)	4 bytes	Cyclic Redundancy Check for error detection

Table 1: Ethernet Frame Field Descriptions

This report provides an overview of the Ethernet switch design and its internal architecture. It includes a detailed explanation of the main modules involved in the frame processing

pipeline, such as the input unit with FCS verification, the MAC learning mechanism, the switch fabric, and the output buffering logic. Each component is described in detail, followed by simulation results and test scenarios used to verify correct behavior and functionality. The report concludes with a summary of findings and observations. Finally, an appendix is included with the full SystemVerilog/VHDL source code used in the implementation.

3 Ethernet Switch Overview

This section provides a detailed overview of the key modules constituting the implemented Ethernet switch architecture. The design adopts a modular structure and supports four full-duplex Gigabit Ethernet ports. The main functional units include the input unit, the MAC address learning and forwarding module, and the switch fabric with output buffering. Each module operates concurrently and communicates with others via well-defined interfaces to support pipelined processing and high throughput.

3.1 Input Unit

The input unit is responsible for receiving Ethernet frames from the physical interfaces and preparing them for switching. There are 4 input ports, each receiving 8 bits of data. Each input port contains a dedicated FIFO buffer of size 2048 bytes to temporarily store incoming frames, ensuring data is not lost during processing.

To support real-time frame integrity checking, the design incorporates a parallel Frame Check Sequence (FCS) verification module. This module, implemented in VHDL, was reused from Exercise 1 and is responsible for computing and verifying the CRC32 of incoming frames in parallel with data reception. If the frame is not valid, then the corresponding FIFO is flushed.

The `inputFSM` module performs frame parsing by identifying Ethernet header fields and controlling subsequent module interactions. Specifically, it extracts the source and destination MAC addresses and collaborates with the MAC address learning module to update or query the lookup table. Once the frame is validated and parsed, the input control module forwards it to the switch fabric, along with the designated output port information.

Additional FIFO structures are used to store extracted source and destination addresses as well as to keep track of each packet's length. A dedicated counter tracks the number of bytes received per frame to facilitate length-based operations and to detect potential protocol violations.

3.2 MAC Learning Module

The MAC address module maintains a dynamic lookup table using a dual-port RAM of size 8192 entries. Each entry contains a 48-bit MAC address and a 4-bit port identifier, supporting the learning and forwarding functionalities of the switch.

Address lookup and storage operations are performed using a custom XOR-based hash function which maps 48-bit MAC addresses to 13-bit indices.

During the *Learning Phase*, the source MAC address of a received frame is hashed to generate a memory index. The corresponding entry in the table is then updated with the MAC address and the input port on which the frame was received. This process allows the switch to learn the association between MAC addresses and ports dynamically.

During the *Forwarding Phase*, the destination MAC address is hashed in the same manner to produce a lookup index. If a valid entry exists, the switch uses the stored port

value to forward the frame to the appropriate output. If no matching entry is found (e.g., due to collisions or a first-time unseen address), the frame is broadcast to all output ports.

The 4-bit encoded port signal is interpreted as follows:

- 0000 : Transmit to port 0
- 0001 : Transmit to port 1
- 0010 : Transmit to port 2
- 0011 : Transmit to port 3
- 1111 : Broadcast to all ports

3.3 Switch Fabric and Output Unit

To mitigate Head-Of-Line (HOL) blocking and maintain high throughput, a cross-point queuing switch architecture is adopted. As illustrated in Figure 2, this structure places FIFO buffers at each intersection of input and output ports, allowing for concurrent transmission and buffering across multiple paths.

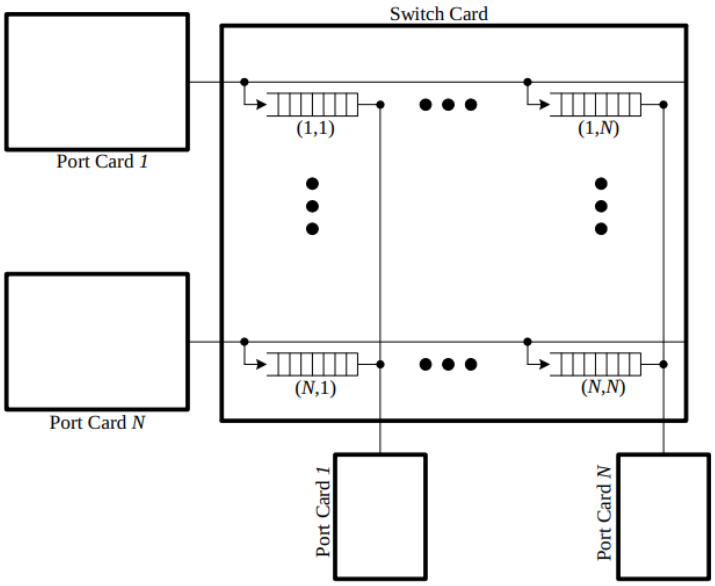


Figure 2: Cross-point Queuing

Incoming packets are routed from the input unit to the appropriate cross-point FIFO based on the destination port determined by the MAC module. Each output port then dequeues packets from its respective cross-point FIFOs, enabling independent arbitration and minimizing contention. For this design, each input FIFO is configured with a 4096-byte capacity to ensure sufficient buffering under bursty traffic conditions.

3.4 Diagram

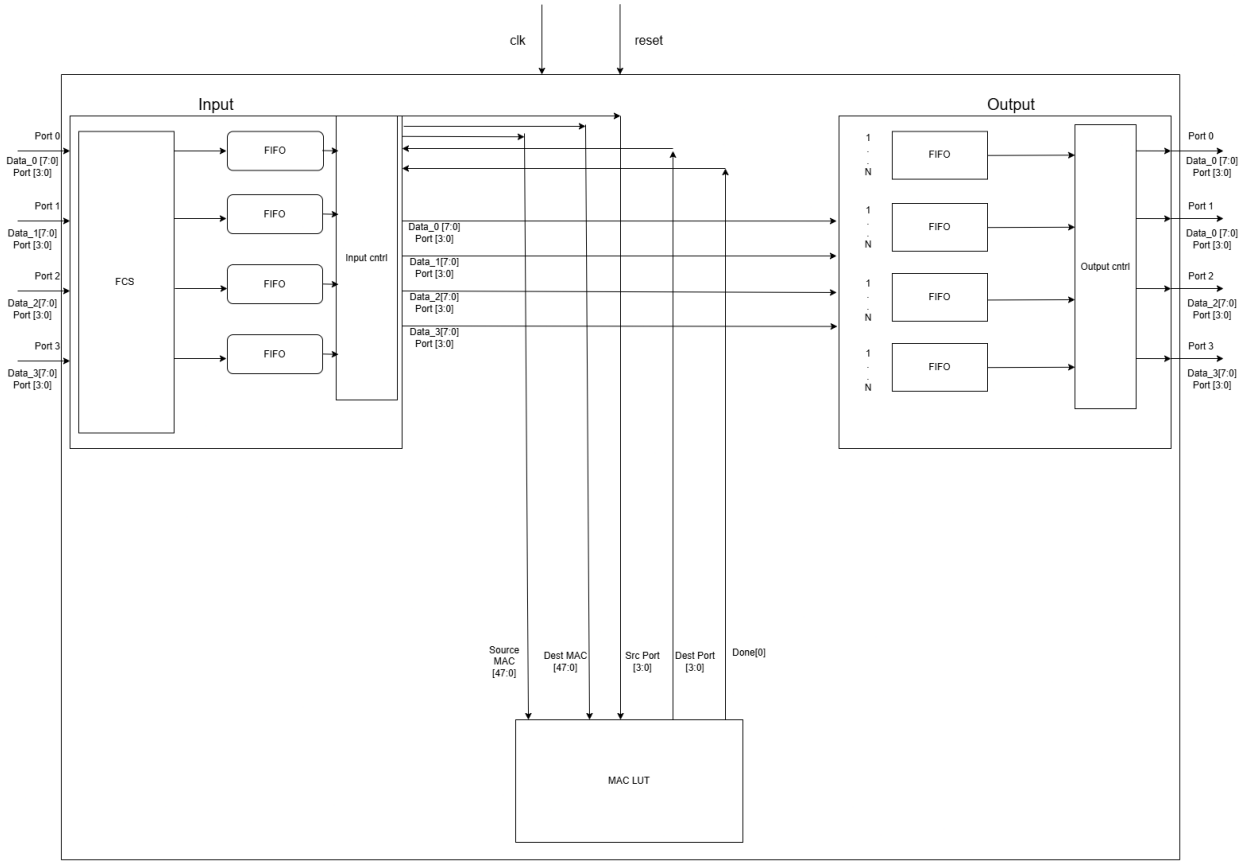


Figure 3: Overview of the Gigabit Ethernet Switch

4 Ethernet Switch Functional Breakdown

4.1 Input Unit

The Input Unit of the Ethernet Switch is responsible for receiving Ethernet frames, validating them, parsing relevant metadata, and preparing them for further processing. It includes several key components as outlined below:

- **Input FIFOs (2048 bytes)** buffer incoming Ethernet frames, accommodating the full Ethernet frame range from 64 bytes to 1518 bytes.
- **Parallel FCS Implementation** performs real-time Frame Check Sequence verification.
- **SOF/EOF Generation Module** detects the start and end of incoming frames.
- **Input FSM** orchestrates metadata extraction and flow control.
- **Source/Destination Address FIFO** captures and stores MAC addresses.
- **Length FIFO** tracks and stores frame length information.

4.1.1 Parallel FCS Implementation

The Frame Check Sequence (FCS) module is designed to verify the integrity of Ethernet frames in a streaming, parallel fashion. Unlike a serial validation scheme, this implementation checks the frame's integrity concurrently with its reception, without requiring the entire frame to be buffered first.

Our implementation uses the `rx_ctrl` signal, which stays high during the reception of a frame and goes low once the frame ends. This control signal drives both the CRC computation and the generation of key control signals such as Start-of-Frame (SOF) and End-of-Frame (EOF).

Internally, the FCS module maintains a byte counter that increments on every clock cycle when `rx_ctrl` is high. This counter tracks how many bytes of the frame have been processed and can be used for alignment, partial frame handling, or CRC complementing logic at the beginning of the frame.

To compute the CRC, incoming data is conditionally complemented during the first four bytes or when the frame is ending. This is a common Ethernet CRC technique used to match the expected frame check sequence value, typically `0xFFFFFFFF`. The data is then passed to a register entity responsible for performing the CRC operation. The `compute_crc` signal, derived directly from `rx_ctrl`, enables this computation.

Once the frame ends (i.e., when `rx_ctrl` goes low), the output of the CRC register is examined. If the result equals the expected value (commonly all ones), the frame is considered valid and the `fcs_error` signal is cleared. Otherwise, an error is flagged, indicating that the frame may have been corrupted during transmission.

4.1.2 SOF and EOF Generation Module

The Start of Frame (SOF) and End of Frame (EOF) detection module identifies the boundaries of each Ethernet frame. This is accomplished by detecting edges on the rx control signal that indicates valid frame data.

Specifically, the SOF is generated when a rising edge is detected (transition from 0 to 1), indicating that a new frame has begun. Conversely, the EOF is detected on a falling edge (transition from 1 to 0), signaling the end of the frame.

4.1.3 Input FSM Module

The Input Unit Finite State Machine (FSM) orchestrates the early-stage processing of incoming Ethernet frames at each port in the system. It is designed to ensure only valid and complete packets are processed, while invalid frames are safely discarded.

Each instance of this FSM handles a single port and operates based on frame control signals and acknowledgments from other submodules such as the MAC learning module and the packet length handler. The FSM transitions through a series of well-defined states that represent the sequential stages of packet validation and metadata extraction.

FSM States and Transitions

- **IDLE**:
  - Waits for a new packet indicated by SOF.
  - On detection, transitions to `FCS_CHECK`.
- **FCS_CHECK**:
  - Waits for EOF to complete packet reception.
  - If there is no FCS error, transitions to `PARSE_ADDR`.
  - If FCS error is detected, transitions to `DELETE_PACKET`.
- **PARSE_ADDR**:
  - Initiates address parsing.
  - Waits for address acknowledgment (`i_addr_ack`) before moving to `MAC_LEARN`.
- **MAC_LEARN**:
  - Issues a MAC learning request with source and destination addresses.
  - Once acknowledgment is received (`mac_ack`), transitions to `GET_LENGTH`.
- **GET_LENGTH**:
  - Requests packet length metadata.
  - Waits for length acknowledgment (`length_ack`) before initiating forwarding.

  - Proceeds to `OUT_SEND`.

- **OUT_SEND**:
  - Sends a request to the switch fabric.
  - Waits for acknowledgment (`i_switch_ack`) before streaming the packet.
- **OUT_SENDING**:
  - Streams packet data.
  - Each cycle decrements an internal counter tracking remaining length.
  - When transmission completes, transitions back to `IDLE`.
  - Sets `o_packet_done` flags for the appropriate output port(s) and switch.
- **DELETE_PACKET**:
  - Drops corrupted packets.
  - Transitions back to `IDLE`.

The FSM uses a number of control signals to synchronize with surrounding modules:

- `o_addr_req` is asserted to request address parsing.
- `o_mac_req` and `o_mac_addr` are used to interact with the MAC learning module.
- `length_req` is used to fetch the frame length, while `fetch_en` enables packet forwarding once processing is complete.
- Internal flags such as `activate_mac`, `activate_length`, and `activate_send` control when each transition is allowed based on acknowledgments from the respective modules.

This FSM-based design ensures that packet processing follows a strict sequence of validation, parsing, and dispatching.

4.1.4 FIFO for Source and Destination MAC Addresses

This FIFO captures the destination and source MAC addresses of each frame. Data is selectively written into the FIFO based on a byte counter that tracks the incoming frame's position. Only the relevant bytes corresponding to MAC addresses (byte 8 to byte 20)

4.2 MAC Learning Module

The MAC Learning module is a critical component of the Ethernet switch that manages the association between MAC addresses and switch ports. It consists of three main components:

- A RAM capable of storing 8000 MAC-Port pairs (48-bit MAC address + 4-bit port identifier)
- A simple XOR hashing algorithm for efficient MAC address lookup
- A finite state machine (FSM) that handles requests from each port

4.2.1 Simple XOR Hashing Algorithm

The hashing algorithm is designed to efficiently map 48-bit MAC addresses to indices in the MAC table. The algorithm works as follows:

1. The MAC address is divided into its six constituent bytes (48 bits total)
2. These bytes are XORed together to produce an initial hash value
3. The hash undergoes final mixing by XORing it with its right-shifted version (shifted by half the table address width)
4. The result is masked to ensure it fits within the table address width

4.2.2 MAC Finite State Machine (FSM)

The MAC learning FSM handles requests from all switch ports in a round-robin fashion. Its operation can be described as:

- **Port Checking States**: The FSM sequentially checks each port for incoming requests. When a request is detected, it captures the source MAC, destination MAC, and source port information.
- **Hashing States**: The FSM computes hash values for both the source and destination MAC addresses using the XOR hashing algorithm.
- **Table Processing**: The FSM performs two main operations:
  - Learns the source MAC by storing it in the table with its associated port
  - Looks up the destination MAC to determine the egress port
- **Comparison and Acknowledgment**: The FSM:
  - Compares the retrieved MAC entry with the destination MAC
  - Sends an acknowledgment to the requesting port
  - Provides either the destination port (if found) or the broadcast indicator 1111 (if not found)

The FSM ensures fair handling of requests from all ports.

4.3 Switch Fabric and Output Unit

The Crossbar module is a core switching component in the Ethernet switch architecture. It is responsible for routing incoming packets from a given input port to the appropriate output ports. The module handles multiple output paths and uses separate FIFOs to mitigate Head-of-Line (HOL) blocking.

4.3.1 HOL Blocking Avoidance

To solve HOL blocking, each `input port` maintains a set of `NUM_OF_PORTS-1` FIFOs (one per output port except its own port). Thus, each input port has a dedicated path to each output port, and packets destined for different ports are queued independently. This approach ensures that a blocked packet heading to one port does not prevent packets for other ports from progressing.

4.3.2 FIFO Buffering and States

Each FIFO buffer has a status associated with it, defined by an enumerated type `BUFFER_STATUS_t`, which includes the following states:

- `PACKET_EMPTY`: The FIFO is empty and ready to receive a packet.
- `PACKET_FILLING`: A packet is currently being written into the FIFO.
- `PACKET_RECEIVED`: The packet has been fully written and is ready for output.
- `PACKET_SENDING`: The packet is currently being read and sent to the output unit.

These states allow the crossbar to track the progress of packet transmission through each FIFO and prevent overwriting or reading incomplete packets.

4.3.3 Input and Output Control Signals

- `i_switch_req`: Indicates a request from the input unit to send a packet.
- `o_switch_ack`: Sent by the crossbar to acknowledge the switch request, enabling the input unit to start transferring the packet.
- `i_packet_done`: One-hot vector indicating that a complete packet has been written into a FIFO.
- `i_out_ack[i]`: Acknowledge signal from the output unit `i`, signaling readiness to receive a packet.
- `o_out_req[i]`: Request to output unit `i` to start transmission.
- `o_switch_data[i]`: Data to be transferred to output unit `i`.
- `packet_done[i]`: Indicates that the FIFO serving output port `i` has completed transmission.

4.3.4 Multicast and Broadcast Support

The crossbar handles unicast and broadcast transfers. In case of a broadcast (i.e., `target_port == ALL_PORTS`), the crossbar ensures all output FIFOs are in the `PACKET_EMPTY` state before transitioning them to `PACKET_FILLING`. This ensures that all ports are synchronized and will receive the broadcast packet simultaneously.

4.3.5 Output Unit

The **Output Unit** is responsible for forwarding packets from the internal router logic to the external transmission interface. It handles multiple incoming requests from various input ports and transmits data accordingly. Its operation is governed by a finite state machine (FSM) with two primary states:

- **OUT_IDLE:** In this state, the unit listens for incoming transmission requests from input ports. A request indicates that a flit (flow control digit) is ready to be transmitted.
- **OUT_ACTIVE:** Upon granting a request, the Output Unit enters this state, enabling data transmission. It continues sending flits from the selected input port until the end of the packet is signaled. Once complete, the unit returns to the `OUT_IDLE` state.

A key internal signal is the *requesting port*, which identifies which input port currently holds the transmission grant. During each clock cycle, the Output Unit checks all ports for valid transmission requests. When a port is selected via arbitration, it asserts an acknowledge signal to that port, and enables transmission signals including the outgoing flit and transmit control signal.

To manage fair access among multiple contenders, the Output Unit leverages a **round-robin arbiter**, which ensures that no single port can monopolize the output channel.

4.3.6 Round-Robin Arbiter

The **Round-Robin Arbiter** is a combinational and sequential logic block that fairly distributes access to a shared resource among multiple requesters. It operates over a set of request signals and generates a corresponding grant vector. Its logic ensures that:

- Every requester gets a fair chance over time.
- The next granted request starts from the one immediately following the last granted request, in a circular fashion.

Internally, the arbiter maintains a rotating pointer indicating the priority starting position. At each clock cycle, if a request is granted, the pointer advances to ensure the next round begins from the subsequent port. This mechanism eliminates starvation and promotes fairness in systems with multiple active ports.

The arbiter evaluates the requests in the order specified by the current pointer and issues a single active grant signal, corresponding to the first active request it encounters. If no requests are active, the grant output remains zero.

4.3.7 Interaction Between Output Unit and Arbiter

The Output Unit uses the arbiter to determine which input port gets access to the output channel when multiple ports request to transmit. The arbiter's decision is fed into the FSM of the Output Unit, which then sets the appropriate control and acknowledgment signals for the selected port.

This architecture ensures that packet transmission is both controlled and fair.

5 Modules specification

5.1 Input Module

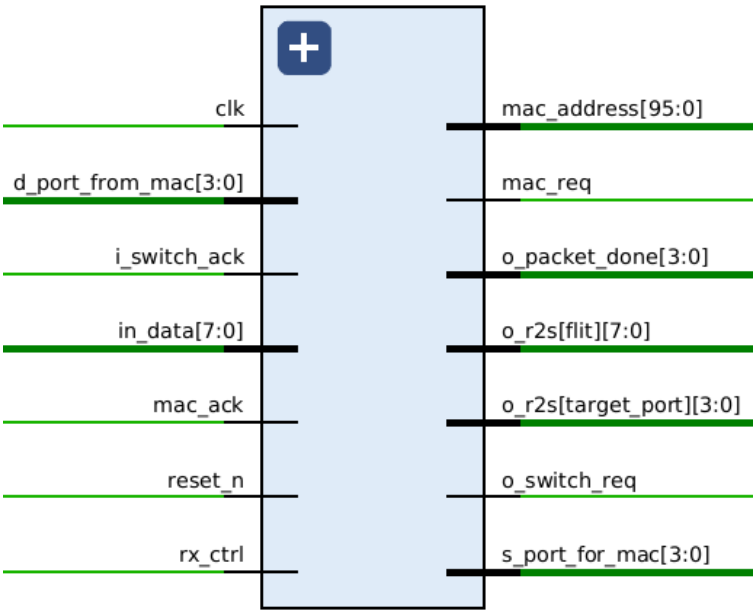


Figure 4: Specification of the Input Unit Module

- **clk**
- **reset_n**: Active-low reset signal to initialize the module.
- **in_data**: Input data bus carrying incoming data.
- **rx_ctrl**: Control signal indicating valid data reception and frame boundaries.
- **mac_ack**: Acknowledgment from MAC module confirming MAC out port.
- **d_port_from_mac**: 4-bit destination port identifier provided by the MAC module.
- **s_port_for_mac**: 4-bit source port identifier sent to the MAC module.
- **mac_req**: Request signal to the MAC module to initiate address processing.
- **mac_address**: 96-bit MAC address (source and destination combined) for MAC learning.
- **o_r2s**: Output bus struct carrying flit data and routing info to the switch.
- **o_switch_req**: Request signal to the crossbar switch to forward data.

- **i_switch_ack**: Acknowledge from the switch indicating readiness to accept data.
- **o_packet_done**: indicating the completion of packet forwarding per port to the crossbar.

5.2 FCS Module

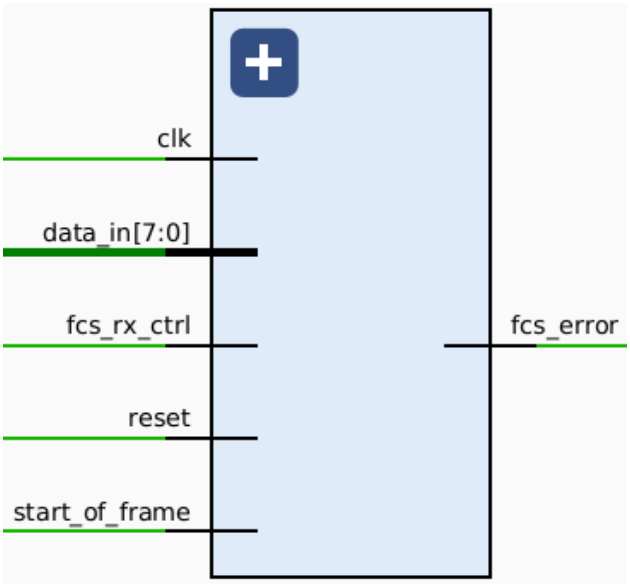


Figure 5: Specification of the FCS (Frame Check Sequence) Module

- **clk**
- **reset**
- **start_of_frame** Indicates arrival of the first bit of a frame.
- **fcs_rx_ctrl** Signal active for the entire duration of the frame reception.
- **data_in** 8-bit input data bus.
- **fcs_error** Output flag indicating FCS (Frame Check Sequence) error detection.

5.3 MAC Module

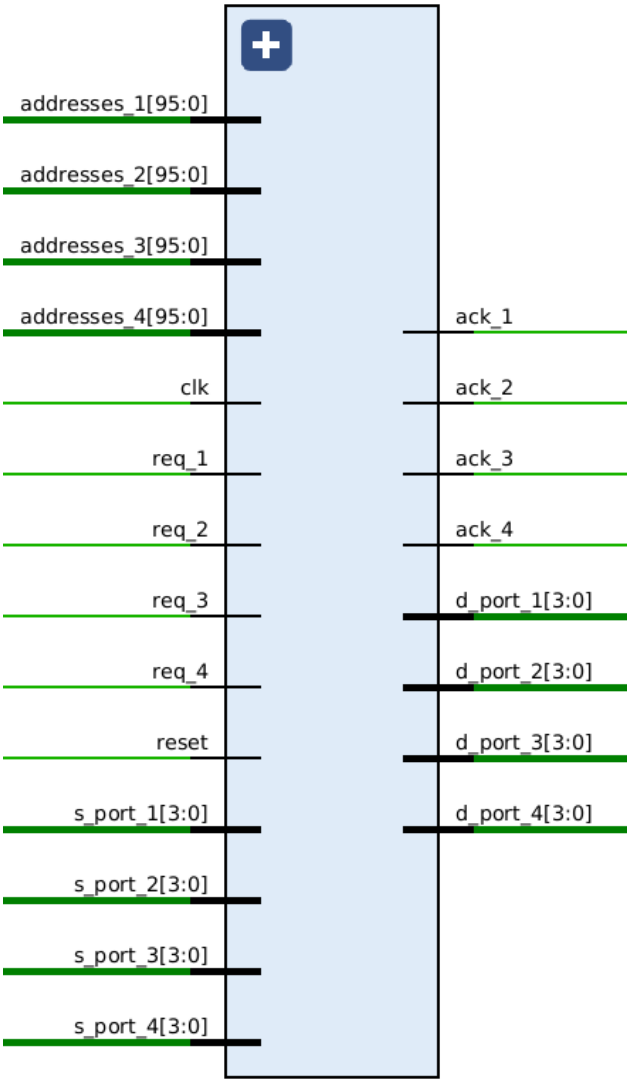


Figure 6: Specification of the MAC Learning Module

- **clk**
- **reset**
- **addresses_X** (X = 1..4): 96-bit input vector containing concatenated MAC addresses; bits [95:48] for source MAC and bits [47:0] for destination MAC.
- **s_port_X** (X = 1..4): 4-bit input indicating the source port number for the incoming frame on port X.
- **req_X** (X = 1..4): Input request signal indicating a new frame is ready on port X.

- **ack_X** (X = 1..4): Output acknowledge signal indicating the request from port X has been processed.
- **d_port_X** (X = 1..4): 4-bit output indicating the destination port for forwarding the frame from port X; outputs 4'b1111 if destination MAC is unknown (broadcast).

5.4 Crossbar Module

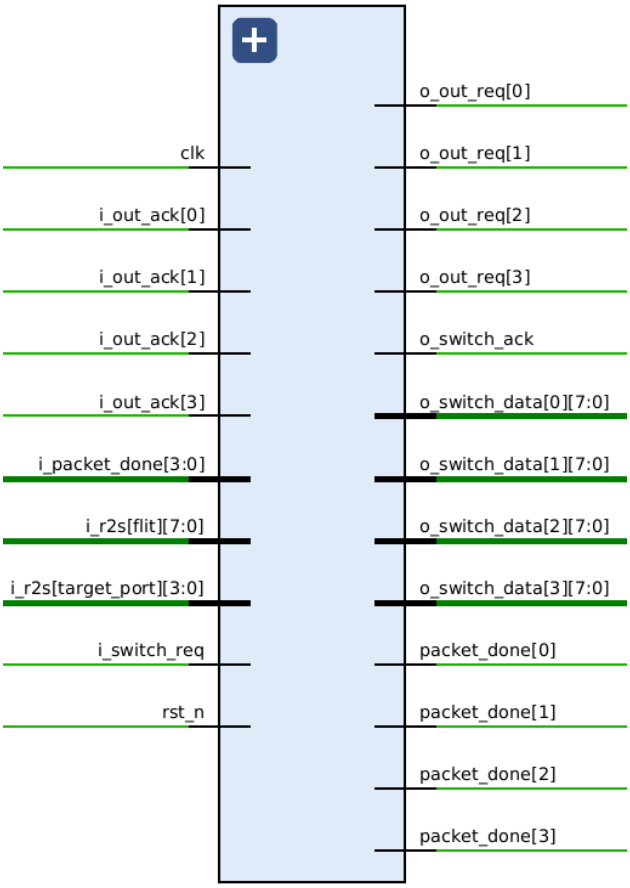


Figure 7: Specification of the Crossbar Module

- **clk**
- **rst_n**
- **i_r2s**: Input bus carrying incoming switch data, including flit and metadata.
- **i_switch_req**: Input signal indicating a new switch request is made.
- **i_out_ack[NUM_OF_PORTS]**: Array of acknowledgments from output ports confirming receipt of data.
- **i_packet_done[NUM_OF_PORTS]**: Flags indicating completion of packet transmission on each output port.
- **o_switch_ack**: Output signal acknowledging the switch request has been accepted.
- **o_out_req[NUM_OF_PORTS]**: Array of Output requests signaling each output port to send data.

- **o_switch_data[NUM_OF_PORTS]**: Output data buses providing flits to each output port from each FIFO.
- **packet_done[NUM_OF_PORTS]**: Flags indicating that packet transmission has finished on the output ports.

5.5 Arbiter Module

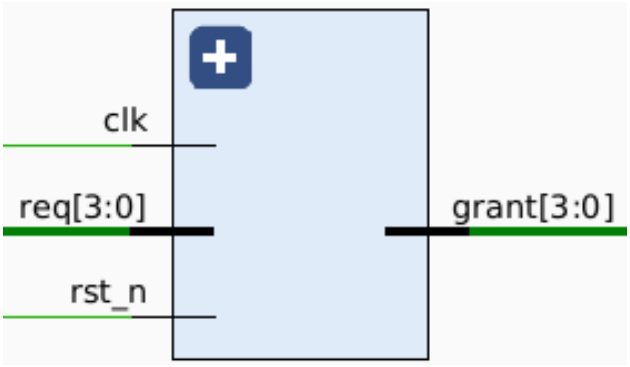


Figure 8: Specification of the Arbiter Module

- **clk**
- **rst_n**:
- **req**: 4-bit input vector where each bit represents a request from one of the four ports.
- **grant**: 4-bit one-hot output indicating which port is currently granted access based on round-robin arbitration.

5.6 Output Module

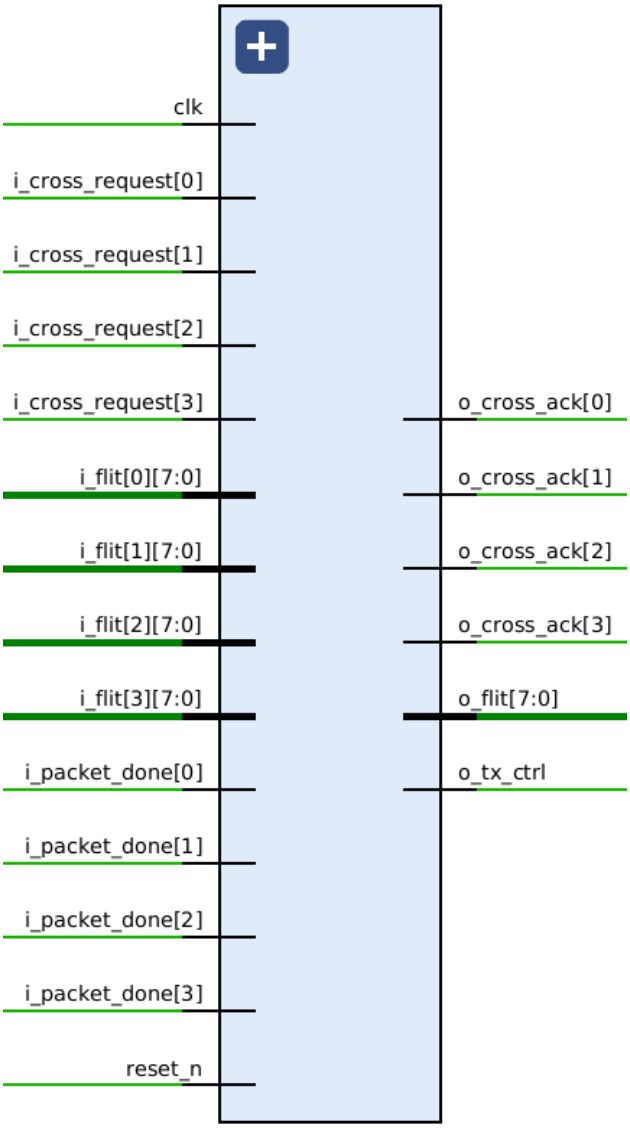


Figure 9: Specification of the Output Unit Module

- **clk**
- **reset_n**
- **i_flit**: Input data flits from all ports (array indexed by port).
- **i_cross_request**: Input requests from each port to send data.
- **o_cross_ack**: Output acknowledgments indicating granted ports.

- **i_packet_done**: Input signals indicating packet transmission completion per port.
- **o_flit**: Output data 8 bits currently transmitted.
- **o_tx_ctrl**: Output control signal indicating transmission in progress.

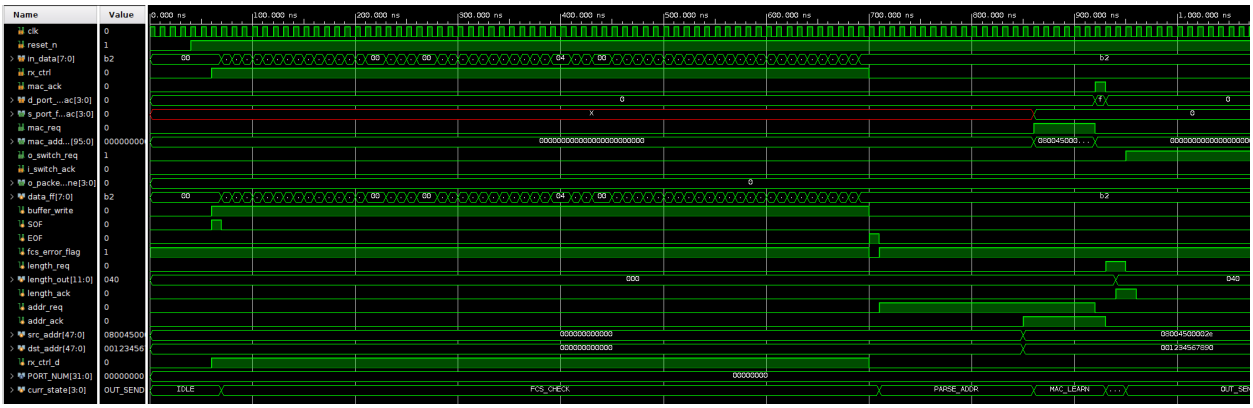
6 Simulation

In this section, we evaluate the Ethernet switch design through several simulation scenarios using a known test packet:

0010A47BEA8000123456789008004500002EB3FE000080110540C0A8002CC0A8000
4040004000001A2DE80001020304050607080 90A0B0C0D0E0F1011E6C53DB2

6.1 Test Input Unit

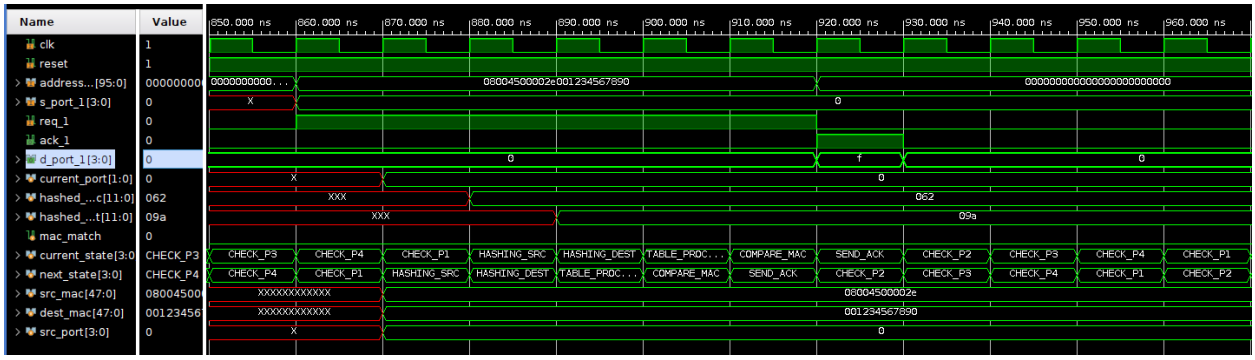
- As shown below, the packet is correctly received on input port 0.
- The `fcs_rx_ctrl` signal correctly goes low, indicating the start of packet reception.
- The FSM transitions to the `PARSE_ADDR` state, and the source and destination MAC addresses are successfully extracted.
- It then enters the `MAC_LEARN` state, where a MAC lookup request is issued.
- As expected (no entry in the MAC table), a broadcast destination is returned along with an acknowledgment.
- The FSM transitions to `OUT_SEND`, requesting access to the crossbar.



6.2 Test MAC Learn Unit

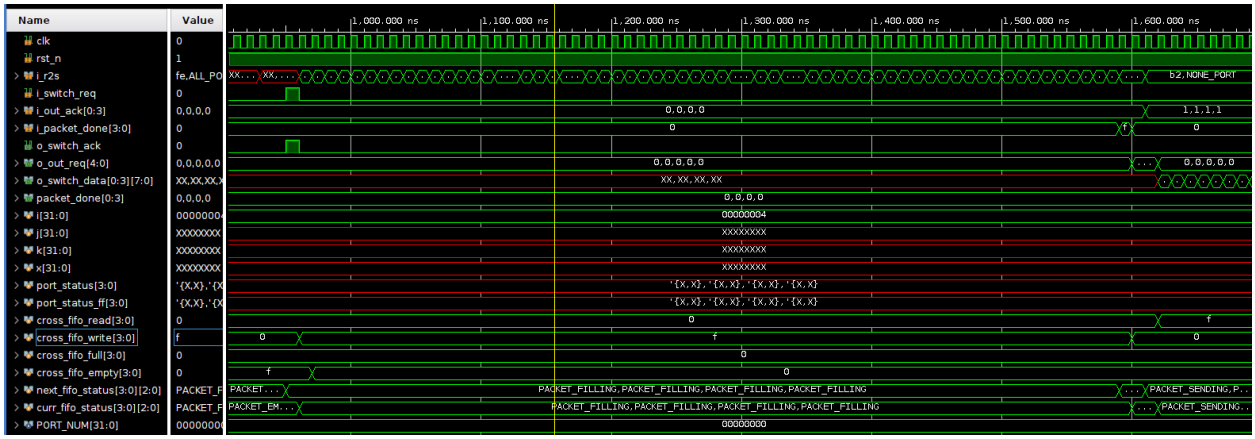
- A MAC lookup request is issued using the concatenated address and port.
- When the FSM checks the port index, it begins the search using the hash of the MAC address.
- Since the table is initially empty, the comparison fails.
- The unit stores the source MAC and port into the table, then returns a broadcast address as the destination.

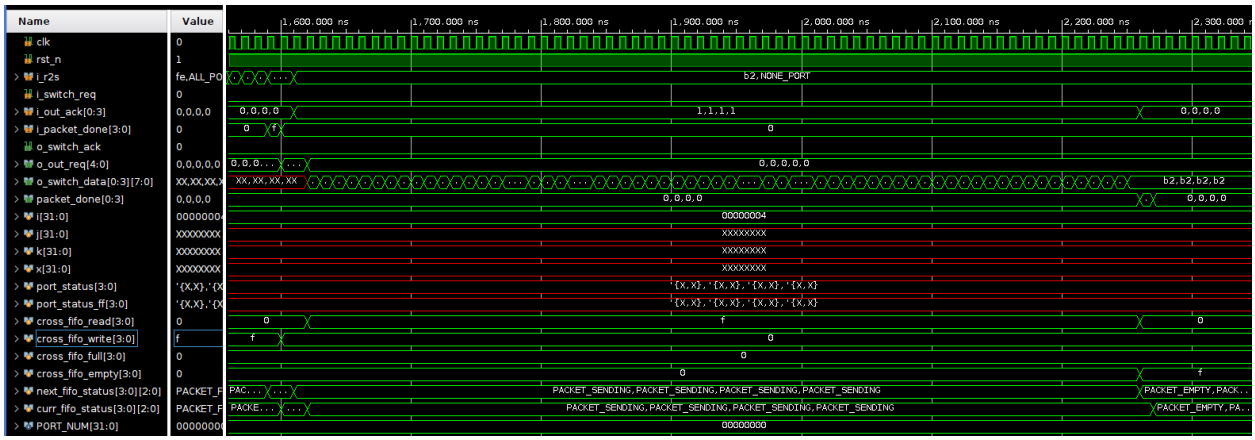
- An acknowledgment is sent back with the broadcast destination.



6.3 Test Crossbar with Output Unit

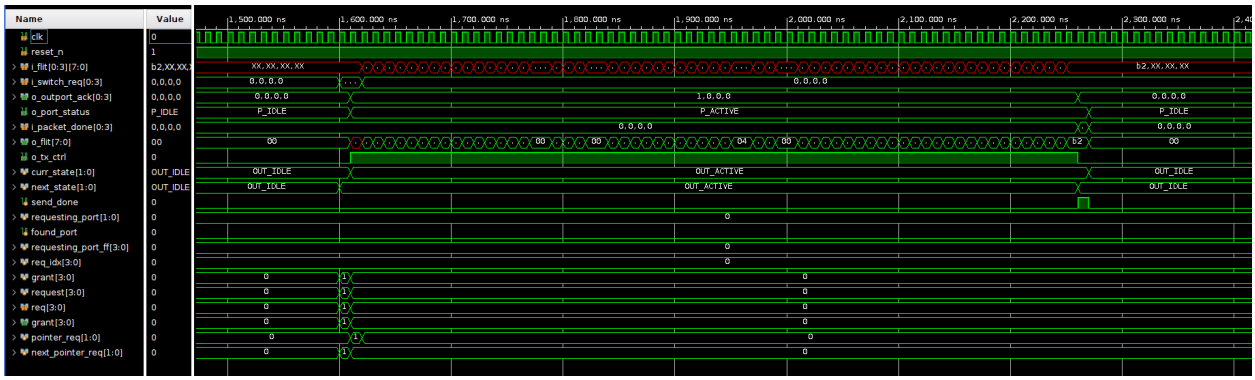
- The crossbar receives the request from the input unit.
- All FIFO inputs of the crossbar start filling up since the destination is broadcast.
- Once the end of packet is detected via `i_packet_done`, the filling stops.
- The crossbar sends requests to all output ports.
- Upon receiving acknowledgments from output units, the packet starts transmitting out through each port.





6.4 Test Output Unit with Arbiter

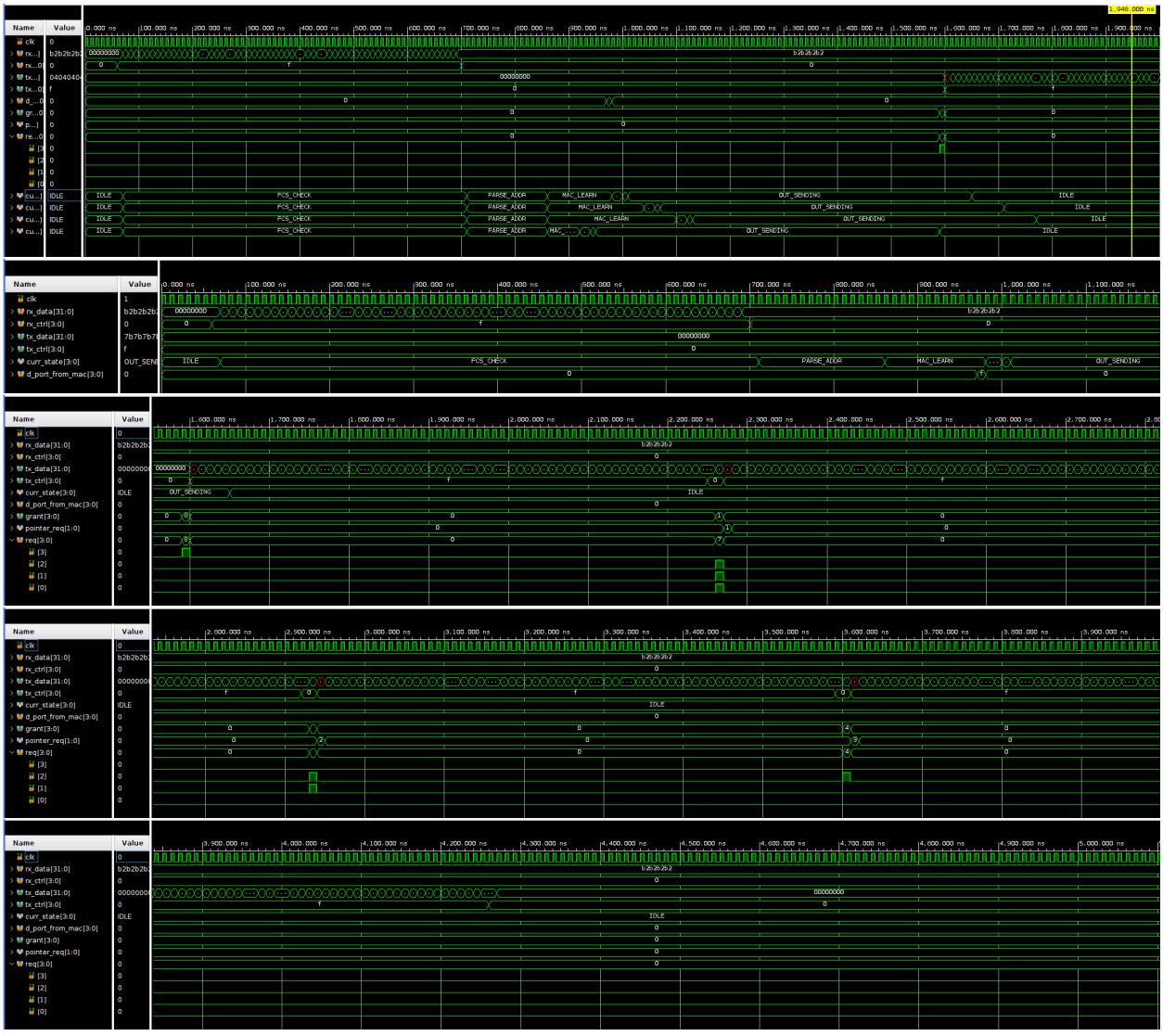
- A request from the crossbar arrives at the output unit.
- Since the output unit is in the `IDLE` state, the arbiter grants access and an acknowledgment is issued.
- The output unit begins transmission of the packet.
- Once the end of the packet is reached, the `o_tx_ctrl` signal goes low and the FSM transitions back to `IDLE`.



6.5 Test with Same Packet in All Ports

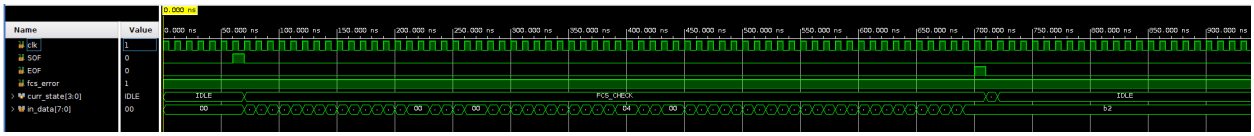
- The same test packet is input simultaneously on all ports (`rx_ctrl = 1111`).
- Each input unit independently processes the packet, performs FCS validation, extracts addresses, and executes MAC learning.
- After computing frame length and verifying correctness, each unit requests access to the crossbar.
- Once crossbar FIFOs are filled, the input units return to the `IDLE` state.

- The crossbar begins requesting the output unit for each input port one by one.
- The arbiter grants access to each requesting port in a round-robin manner.
- As each output unit finishes sending its packet, the arbiter proceeds to grant the next.
- The process continues until all FIFOs are emptied.



6.6 Test with Bad Packet (FCS Failure)

- The packet is received, but a CRC error is detected and `fcs_error` remains high after packet completion.
- The FSM transitions to the `DELETE_PACKET` state, where the invalid frame is discarded.
- Finally, the FSM returns to the `IDLE` state without forwarding the packet.


Name
Value
clk
SOF
EOF
fcs_error
curr_state[3:0]
in_data[7:0]
IDLE
FCS_CHECK
IDLE

7 Conclusion

This work presents the design and simulation of a modular and synthesizable Ethernet switch in Verilog. Through targeted simulations, each component—input unit, MAC learning module, crossbar, and output unit with arbiter—was verified for correct behavior. The input unit successfully parsed incoming frames, performed FCS validation, and initiated MAC learning when necessary. The MAC module demonstrated correct operation by hashing addresses, detecting table misses, and defaulting to broadcast in unresolved cases. The crossbar reliably routed packets to appropriate output FIFOs, and the output unit, governed by a round-robin arbiter, ensured fair and ordered transmission.

System-level simulations confirmed correct operation under both normal and stress conditions, including simultaneous input from all ports. Additionally, error-handling mechanisms were validated using corrupted frames, where faulty packets were detected and dropped as expected.

8 Distribution of Work

The distribution of work was equal between the participants. Theodoros was primarily responsible for the development of the input unit and the MAC learning module, while Dimitrios focused on the design of the switch fabric and output unit. Both participants maintained consistent collaboration and coordination throughout the project, contributing to all aspects of the design, implementation, and verification. This report was a joint effort, with both participants equally involved in its preparation.

9 Appendix

9.1 Ethernet Switch Package

```
    package eth_switch_pkg;
localparam PACKET_LEN = 512;
localparam FIFO_DEPTH = 2048;
localparam DATA_IN_SIZE = 8;
localparam ADDR_BUFFER_DEPTH = 12;
localparam ADDR_LEN = DATA_IN_SIZE * (ADDR_BUFFER_DEPTH/2);
localparam FULL_ADDR_LEN = DATA_IN_SIZE * ADDR_BUFFER_DEPTH;
localparam NUM_OF_PORTS = 4;
localparam NUM_OF_PORTS_BITS = $clog2(NUM_OF_PORTS);
localparam RXTX_DATA_SIZE = 32;
localparam RXTXCTRL_BITS_SIZE = 4;

localparam SRC_MAC_LEN = 50;
localparam DST_MAC_LEN = 50;
localparam FULL_MAC_LEN = SRC_MAC_LEN + DST_MAC_LEN;

localparam TABLE_SIZE = 8000;
localparam TABLE_ADDR_WIDTH = 13;

typedef enum logic [3:0] {
        IDLE = 0,
        FCS_CHECK,
        CHECK_ERROR,
        MAC_LEARN,
        GET_LENGTH,
        OUT_SEND,
        PARSE_ADDR,
        DELETE_PACKET,
        OUT_SENDING
    } GLOBAL_STATE_t;

     typedef enum logic [3:0] {
        CHECK_P1,
        CHECK_P2,
        CHECK_P3,
        CHECK_P4,
        HASHING_SRC,
        HASHING_DEST,
        TABLE_PROCESSES,
        COMPARE_MAC,
        SEND_ACK
    } state_t;
```

```
   typedef struct packed {
       logic [47:0] mac;
       logic [3:0] port;
   } mac_table_entry_t;

    typedef enum logic [2:0] {
   PORT0  = 3'd0,
   PORT1  = 3'd1,
   PORT2  = 3'd2,
   PORT3   = 3'd3,
   ALL_PORTS= 3'd7,
   NONE_PORT   = 3'd5
} PORT_t;

typedef enum logic {
       P_IDLE=0,
       P_ACTIVE=1
} P_STATUS;
typedef struct packed{
     P_STATUS target_port;
     PORT_t   pair;
} SW_PORT_STATUS;

    typedef enum logic [2:0] {
       PACKET_SENT = 0,
       PACKET_RECEIVED  = 1,
       PACKET_FILLING =2,
       PACKET_SENDING = 3,
       PACKET_EMPTY    =4
   } BUFFER_STATUS_t;

typedef logic [DATA_IN_SIZE-1:0] FLIT_t;
typedef struct packed {
   FLIT_t flit;
   PORT_t target_port;
} sw_bus_t;
```

9.2 Synchronous FIFO

```
module sfifo
#(parameter FIFO_WIDTH=5,
parameter FIFO_DEPTH = 4)
(input logic clk ,
input  logic rst_n,
```

```
input  logic i_fifo_write,
input  logic i_fifo_read,
input  logic [FIFO_WIDTH-1:0] i_fifo_write_data,
output logic o_fifo_full,
output logic [FIFO_WIDTH-1:0] o_fifo_read_data,
output logic o_fifo_empty,
output logic [FIFO_DEPTH:0] o_rd_out,
output logic [FIFO_DEPTH:0] o_wr_out);


    logic [FIFO_WIDTH-1:0] mem [(2**FIFO_DEPTH)-1:0];
    logic [FIFO_DEPTH:0] wr_ptr;
    logic [FIFO_DEPTH:0] rd_ptr;

    logic [FIFO_DEPTH:0] wr_ptr_ff ;
    logic [FIFO_DEPTH:0] rd_ptr_ff ;



    assign o_rd_out = rd_ptr_ff;
    assign o_wr_out = wr_ptr_ff;



assign  o_fifo_empty = (wr_ptr_ff[FIFO_DEPTH:0] == rd_ptr_ff[FIFO_DEPTH:0]);
assign o_fifo_full = (wr_ptr_ff[FIFO_DEPTH-1:0] == rd_ptr_ff[FIFO_DEPTH-1:0] &
                        wr_ptr_ff[FIFO_DEPTH] != rd_ptr_ff[FIFO_DEPTH]);
`endif
    always_comb begin : incr_rd
        if(~o_fifo_empty & i_fifo_read)
            rd_ptr=rd_ptr_ff+1;
        else
            rd_ptr = rd_ptr_ff;

    end


    always_comb begin : incr_wr

        if(~o_fifo_full & i_fifo_write)
            wr_ptr = wr_ptr_ff +1 ;
        else
            wr_ptr = wr_ptr_ff;
```

```
end

always_ff @( posedge clk , negedge rst_n ) begin : read_ff
    if(!rst_n)
        rd_ptr_ff <= 0;
    else if(i_fifo_read)
        rd_ptr_ff <= rd_ptr;
    else
        rd_ptr_ff <= rd_ptr_ff;

end


always_ff @( posedge clk , negedge rst_n ) begin : write_ff
    if(!rst_n)
        wr_ptr_ff <= 0;
    else if(i_fifo_write)
        wr_ptr_ff <= wr_ptr;
    else
        wr_ptr_ff <= wr_ptr_ff;

end

always_comb begin : read_mem
    if(~o_fifo_empty & i_fifo_read) begin
        o_fifo_read_data = mem[rd_ptr_ff[FIFO_DEPTH-1:0]];
    end
end



always_ff @(posedge clk ,negedge rst_n ) begin : write_mem

    if(!rst_n)
        mem[wr_ptr_ff[FIFO_DEPTH-1:0]] <= 'x;
    else if(i_fifo_write & ~o_fifo_full) begin
        mem[wr_ptr_ff[FIFO_DEPTH-1:0]] <= i_fifo_write_data;
    end
    else if(i_fifo_read & ~o_fifo_empty)
        mem[rd_ptr_ff[FIFO_DEPTH-1:0]] <= {FIFO_WIDTH{1'bx}}; //deq
    else
        mem[wr_ptr_ff[FIFO_DEPTH-1:0]] <= mem[wr_ptr_ff[FIFO_DEPTH-1:0]];

end
```

```
endmodule
```

9.3 Ethernet Switch

```
import eth_switch_pkg::*;
module eth_switch(
    input   clk,
    input   reset_n,
    input   [RXTX_DATA_SIZE-1:0] rx_data,
    input   [RXTXCTRL_BITS_SIZE-1:0] rx_ctrl,
    output  [RXTX_DATA_SIZE-1:0] tx_data,
    output  [RXTXCTRL_BITS_SIZE-1:0] tx_ctrl
    );
    logic mac_ack[NUM_OF_PORTS];
    logic [3:0]s_port_for_mac[NUM_OF_PORTS];
    logic [3:0]d_port_from_mac[NUM_OF_PORTS];
    logic mac_req[NUM_OF_PORTS];
    logic [95:0] mac_address[NUM_OF_PORTS];




    logic [NUM_OF_PORTS-1:0] switch_req,switch_ack;
    logic [NUM_OF_PORTS-1:0] packet_done [NUM_OF_PORTS];
    sw_bus_t r2s [NUM_OF_PORTS];
    wire [DATA_IN_SIZE-1:0] rx_data_arr [0:NUM_OF_PORTS-1];
    wire [DATA_IN_SIZE-1:0] tx_data_arr [0:NUM_OF_PORTS-1];


    FLIT_t switch_data [NUM_OF_PORTS][NUM_OF_PORTS];
    FLIT_t switch_data2 [NUM_OF_PORTS][NUM_OF_PORTS];
    logic  out_ack [NUM_OF_PORTS][NUM_OF_PORTS];
    logic  out_req [NUM_OF_PORTS][NUM_OF_PORTS];

    logic  cb_packet_done [NUM_OF_PORTS][NUM_OF_PORTS];
    logic  cb_packet_done_tr [NUM_OF_PORTS][NUM_OF_PORTS];

    logic  out_ack2 [NUM_OF_PORTS][NUM_OF_PORTS];
    logic  out_req2 [NUM_OF_PORTS][NUM_OF_PORTS];
    genvar i;
    generate
        for (i = 0; i < NUM_OF_PORTS; i = i + 1) begin : gen_assign
            assign rx_data_arr[i] = rx_data[(i+1)*DATA_IN_SIZE-1 -: DATA_IN_SIZE];
            assign tx_data[(i+1)*DATA_IN_SIZE-1 -: DATA_IN_SIZE] = tx_data_arr[i];
        end
    endgenerate
```

```
genvar rows,cols;
generate
    for(rows = 0; rows<NUM_OF_PORTS; rows++) begin
        for(cols = 0; cols<NUM_OF_PORTS; cols++) begin
            assign switch_data2[rows][cols] = switch_data[cols][rows];
            assign out_ack2[rows][cols] = out_ack[cols][rows];
            assign out_req2[rows][cols] = out_req[cols][rows];
            assign cb_packet_done_tr[rows][cols] = cb_packet_done[cols][rows];
        end

    end
endgenerate

genvar j;
generate
    for (j = 0; j < NUM_OF_PORTS; j++) begin : gen_ports
        InputUnit #(
        .PORT_NUM(j)
    )in_inst
        (
            .clk(clk),
            .reset_n(reset_n),
            .in_data(rx_data_arr[j]),
            .rx_ctrl(rx_ctrl[j]),
            .mac_ack(mac_ack[j]),
            .s_port_for_mac(s_port_for_mac[j]),
            .d_port_from_mac(d_port_from_mac[j]),
            .mac_req(mac_req[j]),
            .mac_address(mac_address[j]),
            .o_r2s(r2s[j]),
            .o_switch_req(switch_req[j]),
            .i_switch_ack(switch_ack[j]),
            .o_packet_done(packet_done[j])
        );

        crossbar  #(
        .PORT_NUM(j)
        ) crossbar_inst (
            .clk(clk),
            .rst_n(reset_n),
            .i_r2s(r2s[j]),
            .i_switch_req(switch_req[j]),
            .i_out_ack(out_ack2[j]),
            .i_packet_done(packet_done[j]),
            .o_switch_ack(switch_ack[j]),
```

```
                .o_out_req(out_req[j]),
                .o_switch_data(switch_data[j]),
                .packet_done(cb_packet_done[j])
            );

            OutputUnit out_inst(
                .clk(clk),
                .reset_n(reset_n),
                .i_flit(switch_data2[j]),
                .i_cross_request(out_req2[j]),
                .o_cross_ack(out_ack[j]),
                .i_packet_done(cb_packet_done_tr[j]),
                .o_flit(tx_data_arr[j]),
                .o_tx_ctrl(tx_ctrl[j])
            );

        end


    endgenerate
    mac_learning_fsm mac_fsm_inst (
        .clk(clk),
        .reset(reset_n),

        .addresses_1(mac_address[0]),
        .s_port_1(s_port_for_mac[0]),
        .req_1(mac_req[0]),
        .ack_1(mac_ack[0]),
        .d_port_1(d_port_from_mac[0]),

        .addresses_2(mac_address[1]),
        .s_port_2(s_port_for_mac[1]),
        .req_2(mac_req[1]),
        .ack_2(mac_ack[1]),
        .d_port_2(d_port_from_mac[1]),

        .addresses_3(mac_address[2]),
        .s_port_3(s_port_for_mac[2]),
        .req_3(mac_req[2]),
        .ack_3(mac_ack[2]),
        .d_port_3(d_port_from_mac[2]),

        .addresses_4(mac_address[3]),
        .s_port_4(s_port_for_mac[3]),
        .req_4(mac_req[3]),
        .ack_4(mac_ack[3]),
```

```
        .d_port_4(d_port_from_mac[3])
    );
endmodule
```

9.4 Input Unit

```
module InputUnit#(
    parameter int PORT_NUM = 0  // Unique ID for this port instance
)(
    input clk,
    input   reset_n,
    input   [DATA_IN_SIZE-1:0] in_data,
    input   logic rx_ctrl,
    input   logic mac_ack,
    input  logic [3:0]d_port_from_mac,
    output   logic [3:0]s_port_for_mac,
    output  [DATA_IN_SIZE-1:0] out_data,
    output  logic mac_req,
    output  [95:0] mac_address,
    output sw_bus_t o_r2s,
    output logic o_switch_req,
    input logic i_switch_ack,
    output logic [NUM_OF_PORTS-1:0] o_packet_done
    );

    logic [DATA_IN_SIZE-1:0] data_ff;
    logic buffer_write;
    logic fetch_en = 0;
    logic buffer_full;
    logic buffer_empty;
    logic [DATA_IN_SIZE-1:0] buffer_odata;
    logic SOF;
    logic EOF;
    logic fcs_error_flag;
    logic length_req=0;
    logic [11:0]length_out = '0;
    logic length_ack=0;

    assign o_r2s.flit = buffer_odata;
    // address buffer
    logic addr_req=0;
    logic addr_ack=0;
    logic [ADDR_LEN-1:0] src_addr='0;
    logic [ADDR_LEN-1:0] dst_addr='0;
```

```
 sfifo #(DATA_IN_SIZE,$clog2(FIFO_DEPTH)) INPUT_BUFFER
(
  .clk(clk),
  .rst_n(reset_n),
  .i_fifo_write(buffer_write),
  .i_fifo_read(fetch_en),
  .i_fifo_write_data(data_ff),
  .o_fifo_full(buffer_full),
  .o_fifo_read_data(buffer_odata),
  .o_fifo_empty(buffer_empty)
);

addr_buffer addr_buffer_handle(
    .clk(clk),
    .reset_n(reset_n),
    .data_in(data_ff),
    .rx_ctrl(rx_ctrl),
    .addr_req(addr_req),
    .addr_ack(addr_ack),
    .src_addr(src_addr),
    .dst_addr(dst_addr)
);

length_count length_counter(
    .clk(clk),
    .reset_n(reset_n),
    .rx_ctrl(rx_ctrl),
    .EOF(EOF),
    .length_ack(length_ack),
    .length_req(length_req),
    .length_out(length_out)
);

inputFSM #(
        .PORT_NUM(PORT_NUM)
    )inFSM(
  .clk(clk),
  .reset_n(reset_n),
  .fetch_en(fetch_en),
  .SOF(SOF),
  .EOF(EOF),
  .fcs_error(fcs_error_flag),
  .o_addr_req(addr_req),
  .i_addr_ack(addr_ack),
  .i_src_addr(src_addr),
  .i_dst_addr(dst_addr),
```

```
      .d_port_from_mac(d_port_from_mac),
      .s_port_for_mac(s_port_for_mac),
      .o_mac_addr(mac_address),
      .o_mac_req(mac_req),
      .length_req(length_req),
      .length_ack(length_ack),
      .mac_ack(mac_ack),
      .input_length(length_out),
      .o_target_port(o_r2s.target_port),
      .o_switch_req(o_switch_req),
      .i_switch_ack(i_switch_ack),
      .o_packet_done(o_packet_done)
    );

    SOF_EOF_ctrl SOF_EOF_ctrl_inst(
      .clk(clk),
      .reset_n(reset_n),
      .rx_ctrl(rx_ctrl),
      .SOF(SOF),
      .EOF(EOF)
    );

    fcs_check_parallel fcs_inst (
        .clk(clk),
        .reset(~reset_n),
        .start_of_frame(SOF),
        .fcs_rx_ctrl(rx_ctrl),
        .data_in(data_ff[7:0]),
        .fcs_error(fcs_error_flag)
    );

    logic rx_ctrl_d;


always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        rx_ctrl_d <= 0;
        data_ff <= '0;
    end
    else begin
        rx_ctrl_d <= rx_ctrl;
        data_ff <= in_data;
    end
end

assign buffer_write = rx_ctrl_d;
```

```
endmodule
```

9.5 Input Unit FCS check

```
entity fcs_check_parallel is
 generic (
  CRC_SIZE   : integer := 32
 );
 port (
  clk            : in std_logic; -- system clock
  reset          : in std_logic; -- asynchronous reset
  start_of_frame : in std_logic; -- arrival of the first bit
  fcs_rx_ctrl    : in std_logic; -- active from start to end of frame
  data_in      : in std_logic_vector(7 downto 0); -- serial input data
  fcs_error      : out std_logic -- indicates an error
 );

end fcs_check_parallel;

Library ieee;
USE ieee.std_logic_1164.all ;
use IEEE.NUMERIC_STD.ALL;

entity Regs is
 generic (
  REG_NUM : integer := 32
 );
 port (
   data_in  : in  std_logic_vector(REG_NUM-1 downto 0);
      fcs_rx_ctrl    : in std_logic; -- active from start to end of frame
   clk    : in  std_logic;
   reset   : in  std_logic;
   data_out : out std_logic_vector(REG_NUM-1 downto 0);
   comp_crc : in  std_logic
   );
 signal regFile : std_logic_vector(REG_NUM-1 downto 0);
end entity;



ARCHITECTURE Behavior OF fcs_check_parallel IS
 constant C_REG_NUM : integer := CRC_SIZE;
 constant POLY    : std_logic_vector(CRC_SIZE-1 downto 0) := x"04C11DB7";
 signal regFileOut  : std_logic_vector(c_REG_NUM-1 downto 0);
```

```
 signal g     : std_logic_vector(c_REG_NUM-1 downto 0);
 signal compute_crc   : std_logic := '0';
 signal compl_en   : std_logic := '0';
 signal data    : std_logic_vector (7 downto 0);
 signal byte_count    : unsigned (7 downto 0);
 signal fcs_rx_ctrl_ff : std_logic := '0';
BEGIN

   process(reset, clk)
 begin
  if(reset = '1') then
   fcs_rx_ctrl_ff <= '0';
  elsif(rising_edge(clk)) then
   fcs_rx_ctrl_ff <= fcs_rx_ctrl;
  end if;
 end process;

 process(reset, clk)
   begin
       if (reset = '1') then
           byte_count <= (others => '0');
       elsif rising_edge(clk) then
           if (fcs_rx_ctrl = '1') then
               byte_count <= byte_count + 1;
           elsif (fcs_rx_ctrl = '0') then
               byte_count <= (others => '0');
           end if;
       end if;
   end process;


 -- Control signals
 compute_crc <= '1' when fcs_rx_ctrl = '1' else '0';

 --regFileOut <= (others => '0') when fcs_rx_ctrl_ff = '0' else regFileOut;

 -- FCS error determination - no longer using bit count but checking if register is zero
 -- after the frame is complete
 fcs_error <= '0' when (fcs_rx_ctrl = '0' and regFileOut = x"ffffffff") else
              '1' when fcs_rx_ctrl = '0' else '1';

 -- Handle first 32 bits of frame with complementing
 compl_en <= '1' when (byte_count <= 4) or (fcs_rx_ctrl = '0') else '0';
 data <= not data_in when compl_en = '1' else data_in;

 -- CRC polynomial implementation
```

```
g(0)  <= regFileOut(24) xor regFileOut(30) xor data(0);
g(1)  <= regFileOut(24) xor regFileOut(25) xor regFileOut(30) xor regFileOut(31) xor data(1);
g(2)  <= regFileOut(24) xor regFileOut(25) xor regFileOut(26) xor regFileOut(30) xor regFileOu
g(3)  <= regFileOut(25) xor regFileOut(26) xor regFileOut(27) xor regFileOut(31) xor data(3);
g(4)  <= regFileOut(24) xor regFileOut(26) xor regFileOut(27) xor regFileOut(28) xor regFileOu
g(5)  <= regFileOut(24) xor regFileOut(25) xor regFileOut(27) xor regFileOut(28) xor regFileOu
g(6)  <= regFileOut(25) xor regFileOut(26) xor regFileOut(28) xor regFileOut(29) xor regFileOu
g(7)  <= regFileOut(24) xor regFileOut(26) xor regFileOut(27) xor regFileOut(29) xor regFileOu
g(8)  <= regFileOut(0) xor regFileOut(24) xor regFileOut(25) xor regFileOut(27) xor regFileOut
g(9)  <= regFileOut(1) xor regFileOut(25) xor regFileOut(26) xor regFileOut(28) xor regFileOut
g(10) <= regFileOut(2) xor regFileOut(24) xor regFileOut(26) xor regFileOut(27) xor regFileOut
g(11) <= regFileOut(3) xor regFileOut(24) xor regFileOut(25) xor regFileOut(27) xor regFileOut
g(12) <= regFileOut(4) xor regFileOut(24) xor regFileOut(25) xor regFileOut(26) xor regFileOut
g(13) <= regFileOut(5) xor regFileOut(25) xor regFileOut(26) xor regFileOut(27) xor regFileOut
g(14) <= regFileOut(6) xor regFileOut(26) xor regFileOut(27) xor regFileOut(28) xor regFileOut
g(15) <= regFileOut(7) xor regFileOut(27) xor regFileOut(28) xor regFileOut(29) xor regFileOut
g(16) <= regFileOut(8) xor regFileOut(24) xor regFileOut(28) xor regFileOut(29);
g(17) <= regFileOut(9) xor regFileOut(25) xor regFileOut(29) xor regFileOut(30);
g(18) <= regFileOut(10) xor regFileOut(26) xor regFileOut(30) xor regFileOut(31);
g(19) <= regFileOut(11) xor regFileOut(27) xor regFileOut(31);
g(20) <= regFileOut(12) xor regFileOut(28);
g(21) <= regFileOut(13) xor regFileOut(29);
g(22) <= regFileOut(14) xor regFileOut(24);
g(23) <= regFileOut(15) xor regFileOut(24) xor regFileOut(25) xor regFileOut(30);
g(24) <= regFileOut(16) xor regFileOut(25) xor regFileOut(26) xor regFileOut(31);
g(25) <= regFileOut(17) xor regFileOut(26) xor regFileOut(27);
g(26) <= regFileOut(18) xor regFileOut(24) xor regFileOut(27) xor regFileOut(28) xor regFileOu
g(27) <= regFileOut(19) xor regFileOut(25) xor regFileOut(28) xor regFileOut(29) xor regFileOu
g(28) <= regFileOut(20) xor regFileOut(26) xor regFileOut(29) xor regFileOut(30);
g(29) <= regFileOut(21) xor regFileOut(27) xor regFileOut(30) xor regFileOut(31);
g(30) <= regFileOut(22) xor regFileOut(28) xor regFileOut(31);
g(31) <= regFileOut(23) xor regFileOut(29);


reg_instance : entity work.Regs
  generic map (
   REG_NUM => C_REG_NUM
  )
  port map (
   data_in  => g,
   clk    => clk,
   fcs_rx_ctrl => fcs_rx_ctrl,
   reset   => reset,
   data_out => regFileOut,
   comp_crc => compute_crc
  );
```

```
END Behavior;



architecture Behavior of Regs is
begin
 process(reset,clk)
 begin
  if(reset = '1') then
   regFile<= (others => '0');
  elsif (rising_edge(clk)) then
    if(comp_crc = '1') then
   regFile <= data_in;
    elsif (fcs_rx_ctrl = '0') then
       regFile <= (others => '0');
    else
   regFile <= regFile;
    end if;
  end if;
 end process;
 data_out <= regFile;
end Behavior;
```

9.6 Input Unit SOF EOF

```
module SOF_EOF_ctrl(
    input  logic clk,
    input  logic reset_n,
    input  logic rx_ctrl,
    output logic SOF,  // Start of frame: rx_ctrl rising edge
    output logic EOF   // End of frame:   rx_ctrl falling edge
);

    logic rx_ctrl_ff;

    always_ff @(posedge clk, negedge reset_n) begin
        if (~reset_n) begin
            SOF <= 0;
            EOF <= 0;
            rx_ctrl_ff <= 0;
        end
        else begin
            rx_ctrl_ff <= rx_ctrl;
            SOF <= (rx_ctrl == 1'b1 && rx_ctrl_ff == 1'b0);
            EOF <= (rx_ctrl == 1'b0 && rx_ctrl_ff == 1'b1);
```

```
        end
    end

endmodule
```

9.7 Input Unit FSM

```
module inputFSM#(
    parameter int PORT_NUM = 0  // Unique ID for this port instance
)(
    input clk,
    input reset_n,
    input logic SOF,
    input logic EOF,
    input logic fcs_error,
    input logic i_addr_ack,
    input logic [ADDR_LEN-1:0] i_src_addr,
    input logic [ADDR_LEN-1:0] i_dst_addr,
    input logic [11:0] input_length,
    input logic length_ack,
    input logic mac_ack,
    input logic [3:0]d_port_from_mac,
    output logic [3:0]s_port_for_mac,
    output logic fetch_en,
    output logic o_addr_req,
    output logic [95:0] o_mac_addr,

    output logic o_mac_req,
    output logic length_req,
    output PORT_t o_target_port,
    output logic o_switch_req,
    input logic i_switch_ack,
    output logic [NUM_OF_PORTS-1:0] o_packet_done
    );

    GLOBAL_STATE_t curr_state;
    GLOBAL_STATE_t next_state;
    logic [11:0]curr_length='0;
    logic [11:0]curr_length_ff;
    logic activate_mac = 0;
    logic activate_send = 0;
    logic activate_length = 0;

    PORT_t target_port;
      assign o_target_port = target_port;
```

```
    always_comb begin
         next_state = IDLE;
         fetch_en = 0;
         o_packet_done = '0;

         unique case(curr_state)
                IDLE : next_state = SOF == 1 ? FCS_CHECK : IDLE;
                FCS_CHECK:   next_state = (EOF == 1) ? (fcs_error == 0 ? PARSE_ADDR : DELETE_PA
                PARSE_ADDR : next_state = activate_mac == 1 ? MAC_LEARN : PARSE_ADDR;
                MAC_LEARN : next_state = activate_length == 1 ? GET_LENGTH : MAC_LEARN;
                GET_LENGTH : next_state = activate_send == 1 ? OUT_SEND: GET_LENGTH;
                OUT_SEND : next_state = i_switch_ack ? OUT_SENDING : OUT_SEND;
                OUT_SENDING : begin
                     next_state = $signed(curr_length) > 0 ? OUT_SENDING : IDLE;
                       if($signed(curr_length) > 0) begin
                         fetch_en = 1;
                    end
                    else begin
                        fetch_en = 0;
                        if(target_port == ALL_PORTS) o_packet_done = '1;
                        else o_packet_done[target_port] = 1;
                    end
                  end
                DELETE_PACKET : next_state = IDLE;
                default : ;
         endcase
    end

    always_comb begin
    activate_mac = 0;
    activate_send = 0;
    o_mac_req = 0;
    o_mac_addr = '0;
    activate_length = 0;
    o_switch_req = 0;
    case(curr_state)
                IDLE : begin
//                    fetch_en = 0;
                    o_mac_req = 0;
                    o_mac_addr = '0;
                    activate_mac = 0;
                    activate_send = 0;
                    activate_length = 0;
                end
                FCS_CHECK : begin
```

```
                end
                PARSE_ADDR : begin
                    o_addr_req = 1;
                    if (i_addr_ack)
                        activate_mac = 1;

                end
                MAC_LEARN : begin
                    if (mac_ack) begin
                        activate_length = 1;
                        o_addr_req = 0;
                        o_mac_req = 0;
                    end else begin
                        activate_length = 0;
                        s_port_for_mac = PORT_NUM[3:0];
                        o_mac_addr = {i_src_addr, i_dst_addr};
                        o_mac_req = 1;
                    end
                end
                GET_LENGTH : begin
                    length_req = 1;
                    if(length_ack) begin
                        curr_length = input_length;
                        activate_send = 1;
                    end
                end
                OUT_SEND : begin
                    length_req = 0;
                    o_switch_req = 1;

                end
                OUT_SENDING : begin


                    curr_length = $unsigned(curr_length_ff) - 1;

                end
                DELETE_PACKET : begin //delete packet and metadata from ALL FIFOS!

                end
        endcase
    end

    always_ff @(posedge clk, negedge reset_n) begin
        if(~reset_n) begin
            curr_state <= IDLE;
```

```
            curr_length_ff <= '0;
            target_port <= NONE_PORT;
        end
        else begin
            curr_state <= next_state;
            curr_length_ff <= curr_length;
            if(mac_ack) target_port <= d_port_from_mac;
            if(curr_state == IDLE) target_port <= NONE_PORT;
        end

    end
endmodule
```

9.8 Input Unit Address FIFO

```
module addr_buffer(
    input logic clk ,
    input  logic reset_n,
    input  logic [DATA_IN_SIZE-1:0] data_in,
    input logic rx_ctrl,
    input logic addr_req,
    output logic addr_ack,
    output logic [ADDR_LEN-1:0] src_addr,
    output logic [ADDR_LEN-1:0] dst_addr
    );


    logic w_en=0;
    logic buffer_full;
    logic buffer_empty;
    logic [DATA_IN_SIZE-1:0] buffer_odata;
    logic [11:0] wr_add_counter = '0;

    logic r_en = 0;
    logic [ADDR_LEN-1:0] tmp_src = '0;
    logic [ADDR_LEN-1:0] tmp_dst = '0;
    logic [11:0] r_add_counter = '0;

    logic r_counter_en = 0;

    always_ff @(posedge clk or negedge reset_n) begin
        if (!reset_n)
            r_counter_en <= 0;
        else if (addr_req)
            r_counter_en <= 1;
```

```
    else if (addr_ack)
        r_counter_en <= 0;
end


sfifo #(DATA_IN_SIZE,$clog2(ADDR_BUFFER_DEPTH)) ADDR_BUFFER
(
  .clk(clk),
  .rst_n(reset_n),
  .i_fifo_write(w_en),
  .i_fifo_read(r_en),
  .i_fifo_write_data(data_in),
  .o_fifo_full(buffer_full),
  .o_fifo_read_data(buffer_odata),
  .o_fifo_empty(buffer_empty)
);



always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        w_en <= 0;
        wr_add_counter <= 0;
    end else begin
        if (rx_ctrl) begin
            if (wr_add_counter >= 6 && wr_add_counter < 18) begin
                w_en <= 1;
            end else begin
                w_en <= 0;
            end
            wr_add_counter <= wr_add_counter + 1;
        end else begin
            wr_add_counter <= 0;
            w_en <= 0;
        end
    end
end

always_ff @(posedge clk or negedge reset_n) begin
    if (!reset_n) begin
        r_en <= 0;
        r_add_counter <= 0;
        tmp_src <= '0;
        tmp_dst <= '0;
        addr_ack <= 0;
        src_addr <= '0;
```

```
            dst_addr <= '0;
        end else begin
            if (addr_req ) begin
                r_en <= 1;
                addr_ack <= 0;
                if (r_counter_en) begin
                    if (r_add_counter >= 0 && r_add_counter < 6) begin
                        r_en <= 1;
                        addr_ack <= 0;
                        tmp_dst[ADDR_LEN-1 - 8*r_add_counter -: 8] <= buffer_odata;
                        r_add_counter <= r_add_counter + 1;
                    end else if (r_add_counter >= 6 && r_add_counter < 12) begin
                        r_en <= 1;
                        addr_ack <= 0;
                        tmp_src[ADDR_LEN-1 - 8*(r_add_counter - 6) -: 8] <= buffer_odata;
                        r_add_counter <= r_add_counter + 1;
                    end else begin
                        r_en <= 1;
                        addr_ack <= 1;
                        src_addr <= tmp_src;
                        dst_addr <= tmp_dst;
                    end
                end
            end else begin
                    r_en <= 0;
                    addr_ack <= 0;
                    r_add_counter <= 0;
                    tmp_src <= '0;
                    tmp_dst <= '0;
                end
        end
    end



endmodule
```

9.9 Input Unit Length FIFO

```
module length_count(
    input logic clk ,
    input  logic reset_n,
    input logic rx_ctrl,
    input logic EOF,
```

```
input logic length_req,
output logic length_ack,
output logic [11:0]length_out
);

logic [11:0] byte_counter = '0;
logic w_en=0;
logic r_en = 0;
logic buffer_full;
logic buffer_empty;
logic EOF_ff = 0;


sfifo #(12,$clog2(4)) PACKET_LENGTH_FIFO
(
  .clk(clk),
  .rst_n(reset_n),
  .i_fifo_write(w_en),
  .i_fifo_read(r_en),
  .i_fifo_write_data(byte_counter),
  .o_fifo_full(buffer_full),
  .o_fifo_read_data(length_out),
  .o_fifo_empty(buffer_empty)
);

always_ff @(posedge clk or negedge reset_n) begin
    if (~reset_n) begin
         EOF_ff <= 0;
    end
    EOF_ff <= EOF;
end

always_ff @(posedge clk or negedge reset_n) begin
    if (~reset_n) begin
         byte_counter <= 0;
    end
    if (rx_ctrl) begin
        byte_counter <= byte_counter + 1;
    end else if (EOF == 0 && EOF_ff == 1) begin
        byte_counter = 0;
    end
end

always_ff @(posedge clk or negedge reset_n) begin
    if (~reset_n) begin
         w_en <= 0;
```

```
        end
        if (EOF == 1) begin
            w_en <= 1;
        end else begin
            w_en <= 0;
        end
    end

    always_ff @(posedge clk or negedge reset_n) begin
        if (~reset_n) begin
             r_en <= 0;
             length_ack <=0;
        end
        if (length_req) begin
            r_en <= 1;
            length_ack <= 1;
        end else begin
            r_en <= 0;
            length_ack <=0;
        end
    end

endmodule
```

9.10 MAC Learning Module

```
module mac_learning_fsm (
    input  logic             clk,
    input  logic             reset,

    // Port 1
    input  logic [95:0]      addresses_1,
    input  logic [3:0]       s_port_1,
    input  logic             req_1,
    output logic             ack_1,
    output logic [3:0]       d_port_1,

    // Port 2
    input  logic [95:0]      addresses_2,
    input  logic [3:0]       s_port_2,
    input  logic             req_2,
    output logic             ack_2,
    output logic [3:0]       d_port_2,

    // Port 3
    input  logic [95:0]      addresses_3,
```

```
    input  logic [3:0]       s_port_3,
    input  logic             req_3,
    output logic             ack_3,
    output logic [3:0]       d_port_3,

    // Port 4
    input  logic [95:0]      addresses_4,
    input  logic [3:0]       s_port_4,
    input  logic             req_4,
    output logic             ack_4,
    output logic [3:0]       d_port_4
);

    mac_table_entry_t mac_table [0:TABLE_SIZE-1];

    logic [1:0] current_port;
    logic [11:0] hashed_index_src, hashed_index_dst;
    logic [51:0] read_mac_entry;
    logic mac_match=0;

    state_t current_state, next_state;
    logic [47:0] src_mac, dest_mac;
    logic [3:0] src_port;

    function automatic logic [TABLE_ADDR_WIDTH-1:0] hash_mac(input logic [47:0] mac);
        logic [TABLE_ADDR_WIDTH-1:0] hash;

        // Split the MAC into 6 bytes and XOR them folded into the hash width
        hash = 0;
        hash ^= mac[47:40];
        hash ^= mac[39:32];
        hash ^= mac[31:24];
        hash ^= mac[23:16];
        hash ^= mac[15:8];
        hash ^= mac[7:0];

        // Final mixing
        hash = hash ^ (hash >> (TABLE_ADDR_WIDTH / 2));

        // Limit the result to TABLE_ADDR_WIDTH bits
        hash = hash & ((1 << TABLE_ADDR_WIDTH) - 1);

        return hash;
    endfunction
```

```
always_ff @(posedge clk or negedge reset) begin
    if (~reset) begin
        current_state <= CHECK_P1;
    end else begin
        current_state <= next_state;
    end
end

always_comb begin
// default assignments
ack_1 = 0;
ack_2 = 0;
ack_3 = 0;
ack_4 = 0;
d_port_1 = 4'b0000;
d_port_2 = 4'b0000;
d_port_3 = 4'b0000;
d_port_4 = 4'b0000;

case (current_state)
    CHECK_P1: begin
        if (req_1) begin
            src_mac = addresses_1[95:48];
            dest_mac = addresses_1[47:0];
            src_port = s_port_1;
            current_port = 2'd0;
            next_state = HASHING_SRC;
        end else begin
            next_state = CHECK_P2;
        end
    end
    CHECK_P2: begin
        if (req_2) begin
            src_mac = addresses_2[95:48];
            dest_mac = addresses_2[47:0];
            src_port = s_port_2;
            current_port = 2'd1;
            next_state = HASHING_SRC;
        end else begin
            next_state = CHECK_P3;
        end
    end
    CHECK_P3: begin
        if (req_3) begin
            src_mac = addresses_3[95:48];
```

```
            dest_mac = addresses_3[47:0];
            src_port = s_port_3;
            current_port = 2'd2;
            next_state = HASHING_SRC;
        end else begin
            next_state = CHECK_P4;
        end
    end
    CHECK_P4: begin
        if (req_4) begin
            src_mac = addresses_4[95:48];
            dest_mac = addresses_4[47:0];
            src_port = s_port_4;
            current_port = 2'd3;
            next_state = HASHING_SRC;
        end else begin
            next_state = CHECK_P1;
        end
    end
    HASHING_SRC: begin
        hashed_index_src = hash_mac(src_mac);
        next_state = HASHING_DEST;
    end
    HASHING_DEST: begin
        hashed_index_dst = hash_mac(dest_mac);
        next_state = TABLE_PROCESSES;
    end
    TABLE_PROCESSES: begin
        mac_table[hashed_index_src] = {src_mac, src_port};
        read_mac_entry = mac_table[hashed_index_dst];
        next_state = COMPARE_MAC;
    end
    COMPARE_MAC: begin
        if (read_mac_entry[51:4] == dest_mac)
            mac_match = 1;
        else
            mac_match = 0;
        next_state = SEND_ACK;
    end
    SEND_ACK: begin
        case (current_port)
            2'd0: begin
                ack_1 = 1;
                d_port_1 = mac_match ? read_mac_entry[3:0] : 4'b1111;
                next_state = CHECK_P2;
            end
```

```
                2'd1: begin
                    ack_2 = 1;
                    d_port_2 = mac_match ? read_mac_entry[3:0] : 4'b1111;
                    next_state = CHECK_P3;
                end
                2'd2: begin
                    ack_3 = 1;
                    d_port_3 = mac_match ? read_mac_entry[3:0] : 4'b1111;
                    next_state = CHECK_P4;
                end
                2'd3: begin
                    ack_4 = 1;
                    d_port_4 = mac_match ? read_mac_entry[3:0] : 4'b1111;
                    next_state = CHECK_P1;
                end
            endcase
        end
        default: next_state = CHECK_P1;
    endcase
end


endmodule
```

9.11 Crossbar

```
module crossbar
    #(parameter unsigned PORT_NUM ) (
        input clk,
        input rst_n,
        input  sw_bus_t i_r2s,
        input  logic i_switch_req,
        input  logic  i_out_ack [NUM_OF_PORTS],
        input logic [NUM_OF_PORTS-1:0] i_packet_done,
        output logic o_switch_ack,
        output logic  o_out_req[NUM_OF_PORTS:0],
        output FLIT_t o_switch_data [NUM_OF_PORTS],
        output logic packet_done [NUM_OF_PORTS]
    );
    integer i,j,k,x;
    SW_PORT_STATUS  [NUM_OF_PORTS-1:0] port_status;
    SW_PORT_STATUS [NUM_OF_PORTS-1:0] port_status_ff;
    logic [NUM_OF_PORTS-1:0] cross_fifo_read = '0 ;
    logic [NUM_OF_PORTS-1:0] cross_fifo_write = '0;
    logic [NUM_OF_PORTS-1:0] cross_fifo_full ='0;
    logic [NUM_OF_PORTS-1:0] cross_fifo_empty = '0;
```

```
 BUFFER_STATUS_t [NUM_OF_PORTS-1:0] next_fifo_status = '{default: PACKET_EMPTY};
 BUFFER_STATUS_t [NUM_OF_PORTS-1:0] curr_fifo_status = '{default: PACKET_EMPTY};


sfifo #(DATA_IN_SIZE,$clog2(FIFO_DEPTH)) cross_fifo0 (
                .clk(clk),
                .rst_n(rst_n),
                .i_fifo_write(cross_fifo_write[0]),
                .i_fifo_read(cross_fifo_read[0]),
                .i_fifo_write_data(i_r2s.flit),
                .o_fifo_full(cross_fifo_full[0]),
                .o_fifo_read_data(o_switch_data[0]),
                .o_fifo_empty(cross_fifo_empty[0])
            );

 sfifo #(DATA_IN_SIZE,$clog2(FIFO_DEPTH)) cross_fifo1 (
                .clk(clk),
                .rst_n(rst_n),
                .i_fifo_write(cross_fifo_write[1]),
                .i_fifo_read(cross_fifo_read[1]),
                .i_fifo_write_data(i_r2s.flit),
                .o_fifo_full(cross_fifo_full[1]),
                .o_fifo_read_data(o_switch_data[1]),
                .o_fifo_empty(cross_fifo_empty[1])
            );

 sfifo #(DATA_IN_SIZE,$clog2(FIFO_DEPTH)) cross_fifo2 (
                .clk(clk),
                .rst_n(rst_n),
                .i_fifo_write(cross_fifo_write[2]),
                .i_fifo_read(cross_fifo_read[2]),
                .i_fifo_write_data(i_r2s.flit),
                .o_fifo_full(cross_fifo_full[2]),
                .o_fifo_read_data(o_switch_data[2]),
                .o_fifo_empty(cross_fifo_empty[2])
            );
 sfifo #(DATA_IN_SIZE,$clog2(FIFO_DEPTH)) cross_fifo3 (
                .clk(clk),
                .rst_n(rst_n),
                .i_fifo_write(cross_fifo_write[3]),
                .i_fifo_read(cross_fifo_read[3]),
                .i_fifo_write_data(i_r2s.flit),
                .o_fifo_full(cross_fifo_full[3]),
                .o_fifo_read_data(o_switch_data[3]),
                .o_fifo_empty(cross_fifo_empty[3])
```

```
            );


 always_ff @(posedge clk, negedge rst_n) begin
     if(~rst_n) begin
         curr_fifo_status <= '{default: PACKET_EMPTY};
     end
     else begin
         curr_fifo_status <= next_fifo_status;
     end
end


 always_comb begin
     next_fifo_status = curr_fifo_status;
     o_switch_ack = 0;
     o_out_req = '{default:0};
     cross_fifo_write = '{default: 0};
     cross_fifo_read = '{default: 0};
     packet_done = '{default:0};
     for(i=0; i < NUM_OF_PORTS; i++) begin
           if(next_fifo_status[i] == PACKET_FILLING) begin
                 cross_fifo_write[i] = 1'b1;
             end

           if( curr_fifo_status[i] == PACKET_FILLING && i_packet_done[i])
             next_fifo_status[i] = PACKET_RECEIVED;

           if(curr_fifo_status[i] == PACKET_RECEIVED)
              o_out_req[i] = 1;

           if(curr_fifo_status[i] == PACKET_SENDING)
                 cross_fifo_read[i] = 1;

           if(curr_fifo_status[i] == PACKET_RECEIVED && i_out_ack[i])
             next_fifo_status[i] = PACKET_SENDING;

           if(curr_fifo_status[i] == PACKET_SENDING && cross_fifo_empty[i]) begin
             cross_fifo_read[i] = 0;
             next_fifo_status[i] = PACKET_EMPTY;
             packet_done[i] = 1;
           end

     end
```

```
        if(i_switch_req && i_r2s.target_port != NONE_PORT) begin
            if(i_r2s.target_port == ALL_PORTS) begin
                if(curr_fifo_status[0] == PACKET_EMPTY &&
                   curr_fifo_status[1] == PACKET_EMPTY &&
                   curr_fifo_status[2] == PACKET_EMPTY &&
                   curr_fifo_status[3] == PACKET_EMPTY) begin

                    next_fifo_status[0] = PACKET_FILLING;
                    next_fifo_status[1] = PACKET_FILLING;
                    next_fifo_status[2] = PACKET_FILLING;
                    next_fifo_status[3] = PACKET_FILLING;
                    o_switch_ack = 1;

                end
            end
            else if( curr_fifo_status[i_r2s.target_port] == PACKET_EMPTY) begin
                next_fifo_status[i_r2s.target_port] = PACKET_FILLING;
                o_switch_ack = 1;
            end
        end

        end
endmodule
```

9.12 Output Unit

```
module OutputUnit
    import eth_switch_pkg::*;

    (
    input   clk,
    input   reset_n,
    input   FLIT_t i_flit [NUM_OF_PORTS],
    input   logic  i_cross_request [NUM_OF_PORTS],
    output  logic o_cross_ack [NUM_OF_PORTS],
    output  P_STATUS o_port_status,
    input i_packet_done[NUM_OF_PORTS],
    output FLIT_t o_flit,
    output logic o_tx_ctrl
    );

    logic switch_ack_ff;
    OutputUnitFSM ofsm (
        .clk(clk),
```

```
        .reset_n(reset_n),
        .i_flit(i_flit),
        .i_switch_req(i_cross_request),
        .o_outport_ack(o_cross_ack),
        .o_port_status(o_port_status),
        .i_packet_done(i_packet_done),
        .o_flit(o_flit),
        .o_tx_ctrl(o_tx_ctrl)
    );


endmodule
```

9.13 Output Unit FSM

```
module OutputUnitFSM
    import eth_switch_pkg::*;

    (
    input clk,
    input reset_n,
    input FLIT_t i_flit[NUM_OF_PORTS],
    input logic  i_switch_req [NUM_OF_PORTS],
    output logic o_outport_ack [NUM_OF_PORTS],
    output P_STATUS o_port_status,
    input i_packet_done [NUM_OF_PORTS],
    output FLIT_t o_flit,
    output logic o_tx_ctrl
    );

    OUT_STATE_t curr_state;
    OUT_STATE_t next_state;
    logic send_done;
    logic [$clog2(NUM_OF_PORTS)-1:0] requesting_port;
    logic found_port;
    logic [NUM_OF_PORTS-1:0] requesting_port_ff;
    logic [NUM_OF_PORTS-1:0] req_idx;
    logic [NUM_OF_PORTS-1:0] grant;
    logic [NUM_OF_PORTS-1:0] request;

    arbiter arb (
        .clk(clk),
        .rst_n(reset_n),
        .req(request),
        .grant(grant)
    );
```

```
always_comb begin
    next_state = OUT_IDLE;
    found_port = 0;
    req_idx = '0;
    request = '0;
    if(curr_state == OUT_IDLE) begin
        for (int i = 0; i < NUM_OF_PORTS; i++) begin
            request[i] = i_switch_req[i];
        end

    end

    casez (grant)
       4'b???1: req_idx = 2'd0;
       4'b??10: req_idx = 2'd1;
       4'b?100: req_idx = 2'd2;
       4'b1000: req_idx = 2'd3;
       default: req_idx = 2'd0;
   endcase

     unique case(curr_state)
            OUT_IDLE : next_state    = |grant ? OUT_ACTIVE : OUT_IDLE;
            OUT_ACTIVE : next_state  = send_done ? OUT_IDLE : OUT_ACTIVE;
            default : ;
     endcase
end
   assign o_port_status = curr_state == OUT_IDLE ? P_IDLE : P_ACTIVE;
   always_comb begin
     o_outport_ack = '{default:0};
     send_done = 0;
     requesting_port = requesting_port_ff;
     o_tx_ctrl = 0;
     o_flit = '0;
     case(curr_state)
            OUT_IDLE : begin
             requesting_port = req_idx;

            end

            OUT_ACTIVE : begin
                o_outport_ack[requesting_port] = 1;
                o_tx_ctrl = 1;
                o_flit = i_flit[requesting_port];
               if(i_packet_done[requesting_port]) begin
                  o_outport_ack[requesting_port] = 0;
```

```
                    send_done = 1;
                    o_tx_ctrl = 0;
                end
            end
        endcase

    end
```

9.14 Arbiter

```
module arbiter(
    input clk,
    input rst_n,
    input logic [3:0] req,
    output logic [3:0] grant
    );
    logic [1:0] pointer_req, next_pointer_req;

  always @(posedge clk,negedge rst_n) begin
    if (~rst_n) pointer_req <= '0;
    else     pointer_req <= next_pointer_req;

  end

  always_comb begin
    next_pointer_req = 3'b000;
    case (grant)
      4'b0001: next_pointer_req =  2'b01 ;
      4'b0010: next_pointer_req =  2'b10 ;
      4'b0100: next_pointer_req =  2'b11 ;
      4'b1000: next_pointer_req =  2'b00 ;

    endcase
  end

    always_comb begin
 case (pointer_req)
  2'b00 :
   if (req[0]) grant = 4'b0001;
   else if (req[1]) grant = 4'b0010;
   else if (req[2]) grant = 4'b0100;
   else if (req[3]) grant = 4'b1000;
   else grant = 4'b0000;
  2'b01 :
   if (req[1]) grant = 4'b0010;
   else if (req[2]) grant = 4'b0100;
```

```
   else if (req[3]) grant = 4'b1000;
   else if (req[0]) grant = 4'b0001;
   else grant = 4'b0000;
        2'b10 :
   if (req[2]) grant = 4'b0100;
   else if (req[3]) grant = 4'b1000;
   else if (req[0]) grant = 4'b0001;
   else if (req[1]) grant = 4'b0010;
   else grant = 4'b0000;
  2'b11 :
   if (req[3]) grant = 4'b1000;
   else if (req[0]) grant = 4'b0001;
   else if (req[1]) grant = 4'b0010;
   else if (req[2]) grant = 4'b0100;
   else grant = 4'b0000;
 endcase // case(req)
end
endmodule
```

9.15 Test Bench

```
`define CLK_PERIOD 8

module Input_tb();
    logic clk = 1;
    logic reset_n = 0;
    logic [RXTX_DATA_SIZE-1:0] rx_data_tb;
    logic [RXTXCTRL_BITS_SIZE-1:0] rx_ctrl_tb;
    logic [RXTX_DATA_SIZE-1:0] tx_data_tb;
    logic [RXTXCTRL_BITS_SIZE-1:0] tx_ctrl_tb;
    logic [DATA_IN_SIZE-1:0] current_byte;
    always #(`CLK_PERIOD/2) clk = ~clk;

    eth_switch switch(
        .clk(clk),
        .reset_n(reset_n),
        .rx_data(rx_data_tb),
        .rx_ctrl(rx_ctrl_tb),
        .tx_data(tx_data_tb),
        .tx_ctrl(tx_ctrl_tb)
    );

    const logic [0:PACKET_LEN-1] goodpacket = 512'h0010A47BEA800012345678
                                              9008004500002EB3FE00008
                                              0110540C0A8002CC0A80004
                                              04000400001A2DE80001020
```

```
                                                    30405060708090A0B0C0D0E
                                                    0F1011E6C53DB2;
 const logic [0:PACKET_LEN-1] badpacket = 512'h0010A47BEB8000123456789
                                                    008004500002EB3FE000080
                                                    110540C0A8002CC0A800040
                                                    4000400001A2DE800010203
                                                    0405060708090A0B0C0D0E0
                                                    F1011E6C53DB2;

 // Stimulus task to send packets
task send_packets(input logic [0:PACKET_LEN-1] packet);
     int i;
     rx_ctrl_tb = '0;
     rx_data_tb = '0;
     #(2*`CLK_PERIOD);

     // Send packet data byte-by-byte
     i = 0;
     while (i < (PACKET_LEN / 8)) begin
         @(posedge clk);
         current_byte = packet[i*8 +: 8];
         rx_data_tb = {current_byte, current_byte, current_byte, current_byte};
         rx_ctrl_tb = 4'b1111;

         i = i + 1;
     end
     // End of packet: reset rx_ctrl_tb to indicate EOF
     @(posedge clk);
     rx_ctrl_tb = 4'b0000;
     #(1*`CLK_PERIOD);

     $display("Packet transmission completed");
 endtask


 initial begin
     reset_n = 0;
     rx_data_tb = '0;
     rx_ctrl_tb = '0;
     #(4*`CLK_PERIOD);
     reset_n = 1;

     send_packets(goodpacket);
     //send_packets(goodpacket);
     //send_packets(badpacket);
```

```
        // Run for some additional time to observe results
        #(500*`CLK_PERIOD);

        $display("Simulation completed");
        $finish;
    end
endmodule
```

References

[1] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.