

Danmarks  
Tekniske  
Universitet



---

# Stateless Firewall

---

## AUTHORS

Hell Felix - s243376  
Santucci Edoardo - s243100  
Staropoli Gea - s243967

May 14, 2025

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Overview</b>	<b>1</b>
<b>3</b>	<b>Blocks specification</b>	<b>4</b>
3.1	GMII to MAC . . . . .	4
3.2	FCS . . . . .	5
3.3	MAC RX Control . . . . .	6
3.4	Packet Analyzer . . . . .	7
3.5	Check Rules . . . . .	9
3.6	FIFO . . . . .	10
<b>4</b>	<b>Blocks description</b>	<b>12</b>
4.1	GMII to MAC . . . . .	12
4.2	FCS . . . . .	13
4.3	MAC RX Control . . . . .	13
4.4	Packet Analyzer . . . . .	14
4.5	Check Rules . . . . .	16
4.6	FIFO . . . . .	18
<b>5</b>	<b>Simulation and verification</b>	<b>20</b>
5.1	GMII to MAC . . . . .	20
5.2	FCS . . . . .	24
5.3	MAC RX Control . . . . .	28
5.4	Packet Analyzer . . . . .	32
5.5	Check Rules . . . . .	35
5.6	FIFO . . . . .	40
<b>6</b>	<b>Firewall System</b>	<b>45</b>
6.1	Design Compilation . . . . .	45
6.2	Testbench Simulation . . . . .	49
<b>7</b>	<b>Conclusions</b>	<b>56</b>
<b>A</b>	<b>Code</b>	<b>58</b>
A.1	GMII to MAC: <code>mac_rx.vhd</code> . . . . .	58
A.2	GMII to MAC testbench: <code>mac_rx_tb.vhd</code> . . . . .	59
A.3	FCS: <code>fcs.vhd</code> . . . . .	62
A.4	FCS testbench: <code>tb_fcs.vhd</code> . . . . .	64
A.5	MAC-RX Control: <code>MAC_RX_CONTROL.vhd</code> . . . . .	67
A.6	MAC-RX Control testbench: <code>MAC_RX_CONTROL_tb.vhd</code> . . . . .	68
A.7	Packet Analyzer: <code>PacketAnalyzer.vhd</code> . . . . .	70
A.8	Packet Analyzer testbench: <code>PacketAnalyzer_tb.vhd</code> . . . . .	73

A.9 Check Rules: <code>CheckRules.vhd</code> . . . . .	76
A.10 Check Rules testbench: <code>tb_CheckRules.vhd</code> . . . . .	79
A.11 FIFO: <code>async_fifo.vhd</code> . . . . .	82
A.12 FIFO testbench: <code>async_fifo_tb.vhd</code> . . . . .	85
A.13 Full System: <code>TopPackCheck.vhd</code> . . . . .	88
A.14 Full System testbench: <code>tb_TopPackCheck.vhd</code> . . . . .	92
<b>B Contributions</b>	<b>96</b>

## 1 Introduction

In this project, we developed a firewall simulation using FPGA technology, aiming to recreate the fundamental behavior of a stateless firewall in a modular and hardware-based environment. The system was implemented using VHDL and deployed on the Intel MAX 10 FPGA (specifically, model 10M50DAF484C7G), allowing us to explore both the theoretical and practical aspects of digital system design.

A stateless firewall is a type of network security device that filters packets based solely on predefined rules, without keeping track of the state of network connections. This means that each packet is evaluated individually, regardless of the traffic that came before or after it. Stateless firewalls are generally simpler and faster than stateful ones, making them suitable for high-performance environments, making it an ideal match for hardware implementation on FPGA.

The architecture of our system was divided into several interconnected blocks, each responsible for a specific stage of packet processing. The first block analyzes the incoming packet and generates control signals that guide the behavior of subsequent modules. The second block performs a CRC check to verify the integrity of the packet. Once the packet is validated, it is passed to a third block that assigns a sequential index to each byte, enabling the structured extraction of specific fields in the next stage. These fields are then evaluated in the firewall Check Rule block, which determines whether the packet complies with the predefined access rules. If all conditions are met, the packet is allowed to proceed; otherwise, it is rejected. For simulation, compilation, and waveform analysis, we used the free edition of ModelSim, which provided a reliable and accessible environment for functional verification. Additionally, we employed the Quartus Prime Lite Edition to gain deeper technical insights into the project. This included flow summary reports, analysis and synthesis processes, fitter results, and detailed timing analysis.

Through this approach, we were able to design a functional and organized stateless firewall simulation that closely mirrors the behavior of real-world systems.

## 2 Overview

The stateless firewall processes incoming Ethernet frames and either allows or blocks them according to pre-defined static rules. The block diagram shown in Figure 1 represents the top-level architecture of the system, with each functional unit encapsulated in a dedicated module and connected via clearly defined signal interfaces. Each block is represented as a rectangular module, with arrows indicating input and output signals. The signal names in the diagram are chosen for improved clarity and understanding, and do not necessarily reflect the actual names used in the VHDL implementation. However, they represent the same functional connections defined in the top-level entity of the system. The firewall begins processing each Ethernet frame starting from the preamble. To enable accurate frame delimitation, we assume that an external module provides the total length of the packet via

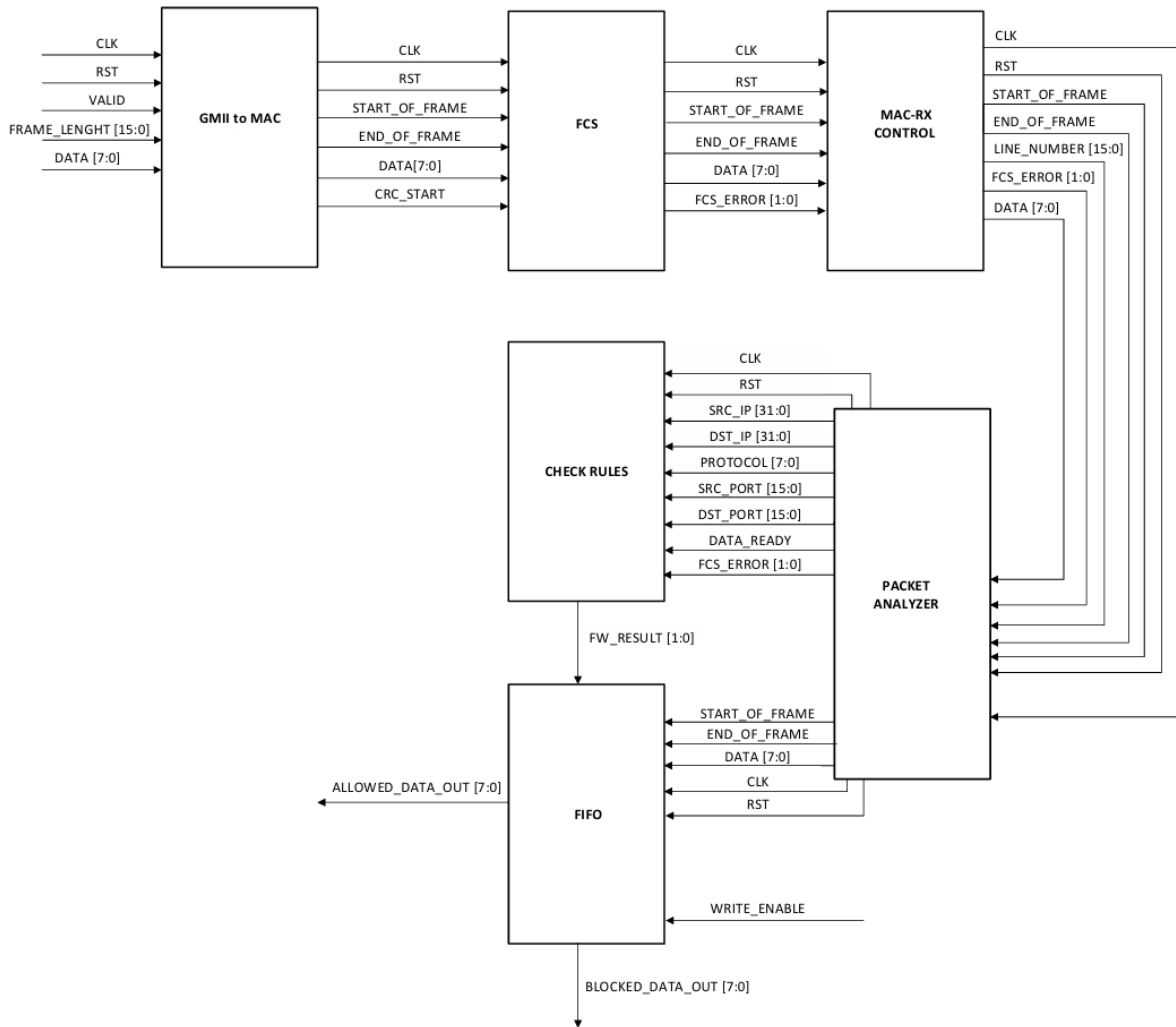


Figure 1: Block Diagram of the Stateless Firewall

the `FRAME_LENGTH` signal.

The **GMII to MAC** block receives the Ethernet frame from the physical interface. It uses the `FRAME_LENGTH` input signal to calculate the start and end of the frame, generating `START_OF_FRAME` and `END_OF_FRAME` signals. This assumption was necessary to identify the boundaries of each frame, enabling proper synchronization for downstream modules. The **FCS (Frame Check Sequence)** block is responsible for calculating the CRC to detect frame corruption. It outputs the result via the `FCS_ERROR` signal, indicating whether the frame passed the integrity check. The **MAC-RX Control** block aligns and formats the received data and generates additional metadata such as the `LINE_NUMBER`, which denotes the byte position within the frame. It passes processed data to the **PACKET ANALYZER** along with control signals. The **Packet Analyzer** extracts key fields from the Ethernet, IP, and transport layer headers: source IP, destination IP, source port, destination port, and protocol. These fields are required for rule checking and are forwarded to the **CHECK RULES** block.

The full packet data is also forwarded to the FIFO. The **Check Rules** block contains the stateless firewall logic. It checks the extracted fields against a set of predefined rules. If a matching **ALLOW** rule is found, the packet is accepted; otherwise, it is rejected. The firewall applies a default deny policy when no rules match. Wildcard rules are supported, such as allowing all TCP traffic. The decision is sent via the **FW\_RESULT** signal. The **FIFO** temporarily stores the processed packet data and forwards it through either **ALLOWED\_DATA\_OUT** or **BLOCKED\_DATA\_OUT**, depending on the result of the firewall rule check.

Finally, this report is organized into the following chapters:

- **Block Specification:** Technical specifications for each module, including descriptions of input and output signals.
- **Block Description:** Functional behavior and internal logic of each block.
- **Simulation and Verification:** Individual simulation and verification of each module using Quartus and ModelSim.
- **System-Level Testing:** Evaluation of the complete firewall system in three different scenarios:
  1. Packet is allowed by matching an **ALLOW** rule.
  2. Packet is blocked due to a specific **DENY** rule.
  3. Packet is blocked due to an **FCS\_ERROR**.

## 3 Blocks specification

### 3.1 GMII to MAC

#### Overview

The **MAC-RX** block handles the reception and initial processing of Ethernet frames from the GMII (Gigabit Media Independent Interface). When valid data is detected via the **GMII\_RX\_DV** signal, the block begins counting incoming bytes to track the structure of the Ethernet frame. It recognizes the end of the standard Ethernet **preamble** sequence by identifying the first 7 bytes, and considers the 8th byte as the Start Frame Delimiter (SFD). The byte following the SFD is identified as the start of the actual Ethernet frame and is flagged with the **MAC\_RX\_FIRST** signal.

The total length of the expected Ethernet packet, including preamble and FCS, is provided dynamically via the **TOTAL\_LENGTH** input. This value is used to determine when the Frame Check Sequence (FCS) field begins and ends. The **MAC\_RX\_VALID** signal is asserted only during the transmission of the FCS bytes, providing a precise indication to the downstream FCS checker. The **MAC\_RX\_LAST** signal marks the final byte of the packet. This coordination ensures the integrity of the frame validation process.

#### Interface Description

##### Inputs

- **GMII\_RX\_CLK**: Clock signal for incoming data from GMII.
- **GMII\_RXD[7:0]**: 8-bit received data from the GMII interface.
- **GMII\_RX\_DV**: Data Valid signal indicating active reception.
- **GMII\_RX\_RESET**: Asynchronous reset signal for the MAC-RX block.
- **TOTAL\_LENGTH[15:0]**: Total expected packet length (including preamble and FCS), provided at runtime.

##### Outputs

- **MAC\_RX\_CLK**: Clock associated with MAC frame data (forwarded from GMII).
- **MAC\_RXD[7:0]**: 8-bit MAC frame data output (directly from GMII).
- **MAC\_RX\_FIRST**: Asserted for one clock cycle on the first byte of the actual Ethernet frame (after preamble and SFD).
- **MAC\_RX\_VALID**: Asserted only during the transmission of the FCS bytes to signal the FCS block.
- **MAC\_RX\_LAST**: Asserted on the final byte of the packet.
- **MAC\_RX\_RESET**: Reset signal forwarded to downstream MAC logic.

## 3.2 FCS

### Overview

The *FCS* (Frame Check Sequence) block receives input bytes from the previous block (MAC-RX) and applies the *CRC-32 algorithm*, using the *polynomial 0x04C11DB7*, to detect transmission errors. During this process, it continues forwarding the data stream. Once the check is complete, it also outputs a variable indicating the result of the verification.

More specifically, once received the active `start_of_frame` signal, the block takes the first 4 incoming bytes and complements them to comply with the Ethernet standard. An internal signal, `R`, is then generated and updated at each clock cycle based on a set of `XOR` logical operations that implement the CRC matrix. When the `valid` signal is asserted, signaling the first byte of the CRC validation, the last 4 bytes are complemented, following the same Ethernet rule, and `R` is recalculated accordingly. Finally, once detected the `end_of_frame` flag, which marks the end of the packet, the content of `R` is evaluated: if it contains only zeros, the packet is considered valid; otherwise, it is considered corrupted. The outcome is then represented using 2 bits on the `fcs_error` signal: the first bit indicates the validity of the result (1 for valid, 0 for invalid), while the second bit reflects the integrity of the frame (1 indicates an error, 0 means error-free).

### Interface Description

#### Inputs:

- `clk` (in `std_logic`): clock signal used to synchronize the operations of the system.
- `reset` (in `std_logic`): reset signal used to initialize the system to a known state.
- `valid` (in `std_logic`): indicates that the data on the input is the first byte of the final 4 bytes of the CRC.
- `start_of_frame` (in `std_logic`): indicates the first byte of the Ethernet frame.
- `end_of_frame` (in `std_logic`): indicates the last byte of the Ethernet frame.
- `data_in` (in `std_logic_vector(7 downto 0)`): input data signal in byte format.

#### Outputs to MAC RX Control:

- `last_of_frame` (out `std_logic`): indicates the last byte of the Ethernet frame for the next block.
- `first_of_frame` (out `std_logic`): indicates the first byte of the Ethernet frame for the next block.
- `data_out` (out `std_logic_vector(7 downto 0)`): output data signal in byte format.



- `fcs_error` (out `std_logic_vector(1 downto 0)`): two bits, the first indicates the validity of the result, and the second indicates if there is an error in the FCS.

### 3.3 MAC RX Control

#### Overview

The *MAC-RX Control* block receives data from the FCS block. It does not validate the correctness of the data itself, but instead forwards a status variable that tracks it (`MAC_RX_ERR` as input and `FW_OUT` as output). This variable will later be evaluated by the Packet Analyzer block, allowing for better timing optimization. In addition, this block counts the number of bytes within each packet, a value that will be used by the subsequent block to identify different roles within the packet.

Within the code, a new signal named `line_number` is introduced. This signal is implemented as a vector that assigns a sequential number to each received byte. Once the `MAC_RX_VALID` signal is asserted, `line_number` begins incrementing with each byte until the `MAC_RX_LAST` signal is received, at which point it resets to 0. Since `line_number` must carry and propagate the numerical position of each byte, it is expressed in bits, requiring it to be defined as a vector. The vector is initialized with a length of 16 bits to ensure it can represent the maximum allowed packet length for IPv4.

#### Interface Description

##### Inputs:

- `MAC_RX_CLK` (in `STD_LOGIC`): clock signal used to synchronize the operations of the system.
- `MAC_RXD` (in `STD_LOGIC_VECTOR(7 downto 0)`): input data signal in byte format.
- `MAC_RX_VALID` (in `STD_LOGIC`): indicates the first byte of the Ethernet frame.
- `MAC_RX_LAST` (in `STD_LOGIC`): indicates the last byte of the Ethernet frame.
- `MAC_RX_ERR` (in `STD_LOGIC_VECTOR(1 downto 0)`): input 2-bit signal representing an error found in the received data, which will be forwarded to the next block.
- `reset` (in `std_logic`): reset signal used to initialize the system to a known state.

##### Outputs to Packet Analyzer:

- `LINE_NUMBER` (out `STD_LOGIC_VECTOR(15 downto 0)`): a 16-bit vector that keeps track of the number of bytes present within the packet.
- `DATA` (out `STD_LOGIC_VECTOR(7 downto 0)`): output data signal in byte format.

- **FW\_OUT** (out STD\_LOGIC\_VECTOR(1 downto 0)): output 2-bit signal representing an error found in the received data, which will be forwarded to the next block..
- **START\_OF\_FRAME** (out STD\_LOGIC): indicates the start of the Ethernet frame for the next block.
- **END\_OF\_FRAME** (out STD\_LOGIC): indicates the end of the Ethernet frame for the next block.

### 3.4 Packet Analyzer

#### Overview

The *Packet Analyzer* is one of the core components of the stateless firewall architecture. This block receives the Ethernet frame and extracts specific fields from the IP and transport layer headers. These fields are then forwarded to the rule-checking logic for the subsequent rule-lookup process.

In this implementation, the block captures the following fields from the IP and transport layer headers:

- **Protocol type** (1 byte)
- **Source IP address** (4 bytes)
- **Destination IP address** (4 bytes)
- **Source port** (2 bytes)
- **Destination port** (2 bytes)

This module operates on a clock-synchronous finite state machine (FSM) driven by the `mac_rx_clk` signal and utilizes the `line_number` input to identify the exact position of each byte within the incoming packet. Based on the line number, the corresponding bytes are assigned to internal registers, each one representing a relevant field of the packet. The fields are extracted between byte indices 0x0017 and 0x0025, corresponding to 15 clock cycles (from the protocol field to the last byte of the destination port). At the end of the capturing process, the block asserts the `data_ready` signal for one clock cycle, indicating that the extracted data can be read by the next block, the `CheckRules` block. Moreover, the signal `fw_out` coming from the FCS module is passed directly to this next block through the output port `fw_out_check`.

In parallel, the block also forwards each incoming byte to the FIFO subsystem, along with control signals `fifo_sof` and `fifo_eof` to denote the frame boundaries.

The FSM transitions through three states:

- **IDLE:** Waits for the start of a new frame.
- **CAPTURING:** Monitors the frame and captures the desired fields based on the `line_number` input.
- **READY:** Signals that all fields have been captured, and the data is valid.

## Interface Description

### Inputs:

- `mac_rx_clk` (STD\_LOGIC): Clock signal from the MAC RX block.
- `rst` (STD\_LOGIC): Synchronous reset signal.
- `line_number` (STD\_LOGIC\_VECTOR(15 downto 0)): Indicates the byte index within the Ethernet frame.
- `data_fw` (STD\_LOGIC\_VECTOR(7 downto 0)): Current byte of the Ethernet frame.
- `start_of_frame` (STD\_LOGIC): High when the beginning of the Ethernet frame is detected.
- `end_of_frame` (STD\_LOGIC): High when the end of the Ethernet frame is detected.
- `fw_out` (STD\_LOGIC\_VECTOR(1 downto 0)): The 2 bit input flag indicating the result of the error check made by the FCS block.

### Outputs to CheckRules:

- `source_ip` (STD\_LOGIC\_VECTOR(31 downto 0)): Extracted source IP address.
- `dest_ip` (STD\_LOGIC\_VECTOR(31 downto 0)): Extracted destination IP address.
- `source_port` (STD\_LOGIC\_VECTOR(15 downto 0)): Extracted source port.
- `dest_port` (STD\_LOGIC\_VECTOR(15 downto 0)): Extracted destination port.
- `protocol` (STD\_LOGIC\_VECTOR(7 downto 0)): Extracted protocol field from the IP header.
- `data_ready` (STD\_LOGIC): High for one clock cycle when all fields have been successfully extracted.
- `fw_out_check` (STD\_LOGIC\_VECTOR(1 downto 0)): Forwarding the error check to the Check Rules block

**Outputs to FIFO:**

- `fifo_data` (STD\_LOGIC\_VECTOR(7 downto 0)): Current byte of the frame forwarded to the FIFO.
- `fifo_sof` (STD\_LOGIC): Start of the Ethernet frame for the FIFO block.
- `fifo_eof` (STD\_LOGIC): End of the Ethernet frame for the FIFO block.

**Internal Signals:**

- `src_ip_reg`, `dst_ip_reg` (STD\_LOGIC\_VECTOR(31 downto 0)): Registers to store the source and destination IP addresses.
- `src_port_reg`, `dst_port_reg` (STD\_LOGIC\_VECTOR(15 downto 0)): Registers to store the source and destination ports.
- `protocol_reg` (STD\_LOGIC\_VECTOR(7 downto 0)): Register to store the protocol field.
- `ready_reg` (STD\_LOGIC): Internal flag used to signal when all required fields have been captured.
- `current_state` (`state_type`): FSM state signal, with possible states: IDLE, CAPTURING, and READY.

### 3.5 Check Rules

**Overview**

The *CheckRules* block is responsible for evaluating the extracted fields from the *Packet Analyzer* and determining whether the packet should be allowed or denied based on predefined firewall rules. Additionally, it processes the error check signal (`fw_out_check`) from the Frame Check Sequence (FCS) to ensure the integrity of the packet.

The firewall table is defined within the block using a record type (`rule_t`) and an array of records (`rule_array_t`). Each record contains the source IP address, destination IP address, protocol, source port, destination port, and an allow/deny flag. The block performs a lookup process to check if the packet matches any of the rules in the table. If a match is found, the corresponding allow/deny value is stored in the `rule_result_value` internal signal. However, the computation of the result for the FIFO is computed only when both the rule lookup and the FCS check have been completed.

The `fw_result` signal is a 2-bit output that encodes both the result of the firewall rule check and the outcome of the Frame Check Sequence (FCS). It is obtained through a logical AND operation between the `rule_result_value` and the negation of the second bit of `fw_out_check` (which indicates whether there is an error). Specifically, `fw_result(0)` is set to the result of `rule_result_value AND NOT fw_out_check(0)`, ensuring that the packet is allowed only if there are no errors. `fw_result(1)` is set to 1 to indicate that the result is ready to be read from the FIFO.

## Interface Description

### Inputs:

- `clk` (STD\_LOGIC): Clock signal.
- `rst` (STD\_LOGIC): Synchronous reset signal.
- `data_ready` (STD\_LOGIC): High when all packet fields are ready.
- `fw_out_in` (STD\_LOGIC\_VECTOR(1 downto 0)): 2 bit FCS result flag.
  - `fw_out_in(1)`: Indicates whether the FCS result is ready (1 = ready).
  - `fw_out_in(0)`: Indicates whether an error was detected (1 = error, 0 = no error).
- `src_ip` (STD\_LOGIC\_VECTOR(31 downto 0)): Extracted source IP.
- `dst_ip` (STD\_LOGIC\_VECTOR(31 downto 0)): Extracted destination IP.
- `protocol` (STD\_LOGIC\_VECTOR(7 downto 0)): Extracted protocol field.
- `src_port` (STD\_LOGIC\_VECTOR(15 downto 0)): Extracted source port.
- `dst_port` (STD\_LOGIC\_VECTOR(15 downto 0)): Extracted destination port.

### Outputs:

- `fw_result` (STD\_LOGIC\_VECTOR(1 downto 0)): Final decision:
  - `fw_result(1)` = 1 if the result is ready.
  - `fw_result(0)` = 1 to allow the packet, 0 to deny it.

### Internal Signals:

- `rule_result_ready` (STD\_LOGIC): Internal flag indicating that the rule lookup result is ready.
- `rule_result_value` (STD\_LOGIC): Holds the allow (1) or deny (0) result from rule matching.

## 3.6 FIFO

### Overview

The **ASYNC\_FIFO** block is an asynchronous FIFO buffer that allows the storage and transfer of Ethernet frames between two independent clock domains: the write clock domain and the read clock domain. It facilitates communication between the GMII reception layer and the system processing layer. The block stores incoming Ethernet frames under the control of the write clock, while the read clock controls data output.

The frame data is stored in the FIFO, and once the entire packet is received, the block awaits a validation signal from the Firewall and FCS checker (**FW\_RESULT**) to determine whether the frame is valid. Valid frames are forwarded through the `read_data_out` port, while invalid frames are silently discarded via the `dummy_data_out` port.

## Interface Description

### Inputs

- **reset**: Asynchronous reset signal that clears internal states and resets pointers.
- **wclk**: Write clock signal used for synchronizing data writes into the FIFO.
- **rclk**: Read clock signal used for synchronizing data reads from the FIFO.
- **write\_enable**: Enables writing of data into the FIFO.
- **write\_data\_in[7:0]**: 8-bit data input for writing packet payload into the FIFO.
- **SOP**: Start-of-Packet signal indicating the start of a new Ethernet frame.
- **EOP**: End-of-Packet signal indicating the end of the current Ethernet frame.
- **FW\_RESULT[1:0]**: Frame validation result from the Firewall and FCS checker. `FW_RESULT(1)` signals readiness, and `FW_RESULT(0)` indicates whether the frame should be forwarded or discarded.

### Outputs

- **fifo\_occu\_in[PTR\_WIDTH-1:0]**: FIFO occupancy level as observed from the write domain.
- **fifo\_occu\_out[PTR\_WIDTH-1:0]**: FIFO occupancy level as observed from the read domain.
- **read\_data\_out[7:0]**: 8-bit data output for valid and authorized frames.
- **dummy\_data\_out[7:0]**: 8-bit data output for invalid or unauthorized frames.
- **fifo\_full**: Indicates that the FIFO is full and cannot accept more data.
- **fifo\_empty**: Indicates that the FIFO is empty and no data is available for reading.

## 4 Blocks description

### 4.1 GMII to MAC

#### Overview

The **MAC-RX** block is responsible for receiving Ethernet frames from the GMII (Gigabit Media Independent Interface) and signaling frame boundaries and integrity check timing to downstream components. This block monitors incoming data streams, identifies the end of the preamble, and tracks the full frame length using an externally supplied length value. Unlike traditional preamble-removal logic, the MAC-RX does not remove the preamble data but instead identifies key frame positions based on byte counting. It marks the first byte of the actual Ethernet payload with the **MAC\_RX\_FIRST** signal and the final byte with the **MAC\_RX\_LAST** signal. The **MAC\_RX\_VALID** signal is specifically used to indicate the period during which the FCS (Frame Check Sequence) bytes are being received, so the downstream FCS checker can begin and end its validation logic accurately.

#### Functional Description

The **MAC-RX** block operates synchronously on the rising edge of **GMII\_RX\_CLK**. Upon detecting valid data through the **GMII\_RX\_DV** signal, it begins incrementing a byte counter to track the position within the frame. The preamble is identified by its position in the stream: once the counter reaches byte index 6, the preamble is considered complete. The byte at position 8 is identified as the first byte of the actual Ethernet frame payload. At this point, the **MAC\_RX\_FIRST** signal is asserted for one clock cycle. The value of **TOTAL\_LENGTH** is used to calculate the expected frame end. When the counter reaches the total length minus one, the **MAC\_RX\_LAST** signal is asserted, indicating the last byte. For the last 4 bytes (typically the FCS field), the **MAC\_RX\_VALID** signal is asserted, allowing the downstream FCS block to latch the correct data range for integrity checking. After the final byte, the internal byte counter is reset, and the block is ready to process the next frame.

#### Reset Behavior

The block responds to the **GMII\_RX\_RESET** signal asynchronously. When asserted:

- The internal byte counter is reset to 0.
- The **preamble\_done** flag is cleared.
- Output signals **MAC\_RX\_VALID**, **MAC\_RX\_FIRST**, and **MAC\_RX\_LAST** are deasserted.

This ensures a clean initialization and consistent frame boundary tracking following any reset.

## Clock Domain Synchronization

The **MAC-RX** block operates entirely in the GMII clock domain. To maintain timing consistency, the input clock **GMII\_RX\_CLK** is forwarded directly as **MAC\_RX\_CLK** for use by downstream modules, ensuring proper synchronization and eliminating the need for cross-domain logic.

## 4.2 FCS

The FCS block ensures data integrity by verifying whether the received Ethernet frame is error-free. This verification is done using the CRC-32 algorithm, which detects transmission errors before the frame is passed to the MAC-RX Control block. The block operates based on the standardized polynomial 0x04C11DB7.

When a frame is received, the FCS block uses the system clock to synchronize data operations. It processes the 8-bit received MAC data, using the start-of-frame and end-of-frame signals to identify the beginning and end of the frame. Since the CRC bytes used for error detection are located in the last four bytes of the packet, the packet is always forwarded for further handling to the next block.

However, if an error is detected, the error flag signal is activated. This flag is then sent to the packet analyzer block, which, based on the final value of this signal, will decide whether or not to allow the final transmission of the packet.

As previously mentioned, the CRC consists of 4 bytes computed using a specific algorithm based on a predefined generator polynomial. This mechanism allows verification of the integrity of the received packet. Conceptually, the data is treated as a polynomial and divided by the generator polynomial, then the remainder of this division constitutes the CRC code, which is appended to the end of the data before transmission.

Since we are operating at the bit level, polynomial division is implemented through successive XOR operations. If the data were received bit by bit, each XOR operation could be performed sequentially using the incoming bit, allowing straightforward logic to compute the R vector that reflects the division remainder. However, in our case, data is processed in parallel, 8 bits at a time, instead than serially. This introduces some more complexity: updating the R value requires taking into account simultaneous updates of multiple bits, which, according to the algorithm, should ideally occur in sequence. To ensure the CRC computation remains accurate under parallel processing, we rely on a precomputed CRC matrix. In our implementation, this matrix was generated using MATLAB due to its large size and complexity. From this matrix, we extract the necessary logical equations to construct the logic that verifies packet correctness.

## 4.3 MAC RX Control

The MAC-RX Control block is responsible for handling MAC packets. It ensures that the packets, along with a validity flag, are forwarded for further processing.

The block receives data from the FCS block and uses various control signals to identify the start and end of frames. It also forwards the FCS error flag, which will be used later in the



packet analyzer block to determine whether the packet is valid and should be sent.

The block assign to each incoming byte a number, which are later used in the packet analyzer block to determine the positions of specific data fields, such as the payload, source IP address, destination IP address, and port number.

#### 4.4 Packet Analyzer

As introduced in the block specification, the *Packet Analyzer* is responsible for processing incoming Ethernet frames and extracting the essential fields required for rule checking. These include the protocol, source and destination IP addresses and source and destination ports. The module operates synchronously with the incoming data stream, analyzing one byte per clock cycle, and uses a counter signal (`line_number`) to track the current byte position within the frame.

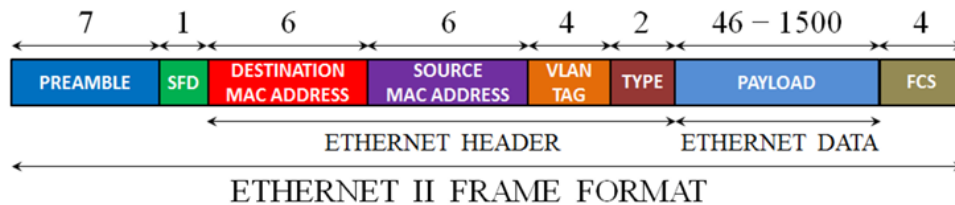


Figure 2: Ethernet frame format

#### The Ethernet Frame

In order to understand the assignment logic, it is useful to first introduce how an Ethernet frame is structured (Figure 2). The typical structure of an Ethernet frame (without VLAN tag) is as follows:

- **Preamble (7 bytes)**: Used for synchronization
- **Start Frame Delimiter (1 byte)**: Indicates start of frame
- **Destination MAC Address (6 bytes)**: Starting at byte 0
- **Source MAC Address (6 bytes)**: Starting at byte 6
- **EtherType (2 bytes)**: Starting at byte 12 (e.g., 0x0800 for IPv4)
- **Payload**: Includes the IP header and the encapsulated TCP/UDP segment
- **Frame Check Sequence (FCS, 4 bytes)**: CRC

In our implementation, the byte counting process excludes the preamble and the Start Frame Delimiter (SFD), meaning the count begins from the first byte of the destination MAC address, which is considered as byte 0. The EtherType field in the Ethernet frame header specifies the protocol encapsulated within the payload. Common values include IPv4 (0x0800), IPv6 (0x86DD), and ARP (0x0806).

This firewall system is designed to handle IPv4 packets. An IPv4 packet consists of two primary components: the IP header and the payload of the transport layer. Within the IP header, which starts at byte 14, we find fields such as the protocol type (byte 23), which determines whether the transport payload carries TCP (0x06) or UDP (0x11) data, as well as the source IP address (bytes 26 to 29) and destination IP address (bytes 30 to 33). Following the IP header is the transport layer segment, where the source and destination ports are located. These typically start at byte 34 and 36, respectively, assuming a standard IP header length of 20 bytes (without the *Options* field). Table 1 below shows each field of the Ethernet Frame with the respective byte index and length.

Offset (Byte)	Field	Length
0	Destination MAC Address	6 bytes
6	Source MAC Address	6 bytes
12	EtherType (0x0800 = IPv4)	2 bytes
14	IP Version, IHL, TOS	2 bytes
16	Total Length	2 bytes
18	Identification	2 bytes
20	Flags, Fragment Offset	2 bytes
22	TTL	1 byte
23	<b>Protocol</b>	1 byte
24	Header Checksum	2 bytes
26	<b>Source IP Address</b>	4 bytes
30	<b>Destination IP Address</b>	4 bytes
34	<b>Source Port</b>	2 bytes
36	<b>Destination Port</b>	2 bytes

Table 1: Relevant fields in an IPv4 Ethernet frame

## Functional Description

As the FSM moves into the `CAPTURING` state, it monitors the `line_number` signal to perform the following assignments:

- Byte 23  $\rightarrow$  `protocol`
- Bytes 26 to 29  $\rightarrow$  `src_ip`
- Bytes 30 to 33  $\rightarrow$  `dst_ip`
- Bytes 34 to 35  $\rightarrow$  `src_port`

- Bytes 36 to 37 → `dst_port`

Once all the fields are collected (at `line_number = 37`), the module moves to the **READY** state and sets the `data_ready` signal high for one clock cycle, indicating that the extracted data is valid. The Packet Analyzer outputs data to two different blocks in parallel:

1. **CheckRules Block:** The extracted fields (`src_ip`, `dst_ip`, `src_port`, `dst_port`, and `protocol`) are forwarded together with the `fw_out_check` signal.
2. **FIFO Buffer:** All incoming bytes on the `data_fw` input are directly forwarded to the FIFO buffer, along with the `fifo_sof` and `fifo_eof` signals indicating the start and end of the frame.

### Reset Behavior

The block includes a reset mechanism that clears its internal state signals. When asserted, this signal resets all internal registers and state variables to their initial values. Specifically, the reset operation clears the registers storing the source IP address, destination IP address, source port, destination port, and protocol. It also resets the output signals toward the FIFO and CheckRules blocks, and sets the internal FSM state back to **IDLE**. This ensures the module starts in a consistent and known state upon reset, mirroring the approach used in the **CheckRules** block.

## 4.5 Check Rules

The *CheckRules* block serves as the core decision unit of the firewall. Its main function is to evaluate incoming packet field extracted by the *Packet Analyzer* and determine whether the packet should be allowed or denied, based on a predefined set of firewall rules. Additionally, it verifies the integrity of the packet using the FCS result. The outcome of this evaluation is encoded in a 2-bit signal `fw_result`, which is then sent and used by the FIFO as an input flag indicating if the packet has to be discarded or forwarded.

The firewall rules are implemented internally as an array of records, where each record contains the following fields: source IP address, destination IP address, protocol, source port, destination port, and an allow/deny flag. Table 2 lists the rules implemented in this project.

Source IP	Dest. IP	Protocol	Src Port	Dst Port	Action
192.168.1.1	192.168.1.2	UDP (11)	8080	80	Block
192.168.0.44	192.168.0.4	UDP (11)	1024	1024	Allow
192.168.1.1	192.168.1.2	TCP (06)	8080	80	Block
Any	Any	TCP (06)	Any	Any	Allow

Table 2: Firewall Rule Table

## Functional Description

Upon receiving a new packet (signaled by `data_ready`), the block performs a comparison between the packet fields and each rule in the table. If a matching rule is found, the corresponding decision (allow or deny) is stored in the internal signal `rule_result_value`. To support flexible and scalable rule matching, the implementation also introduces the concept of **wildcards**. Each field in a rule can be optionally ignored during the matching process by enabling its corresponding wildcard bit. This allows for generalized rules (e.g., "allow all TCP packets") by masking irrelevant fields. Internally, each rule includes a set of 5 wildcard bits (one per field), and a field is considered a match either if it equals the packet's value or if its wildcard is active. If no match is found, the packet is denied by default (default deny policy).

In parallel, the block receives the `fw_out_in` signal from the FCS unit, which is a 2-bit vector indicating:

- `fw_out_in(1)`: FCS result ready flag (set to 1 when the FCS check has been completed).
- `fw_out_in(0)`: FCS error flag (set to 1 if the packet contains errors).

Once both the rule lookup and the FCS check have completed, the block computes the final result:

- `fw_result(1)` is set to 1 when the rule look-up process is terminated (`rule_result_ready = 1` and the FCS result is available (`fw_out_in(1) = 1`), signaling that the output is valid.
- `fw_result(0)` is set to 1 only if the packet matches an allow rule (`rule_result_value = 1`) and the FCS indicates no error (`fw_out_in(0) = 0`).

This logic ensures that a packet is forwarded only if it is explicitly allowed by the firewall rules and is free of transmission errors. The output `fw_result` is then passed to the FIFO block, which queues the result for further handling by downstream components.

## Security oriented approach

In this firewall implementation, a security oriented approach was adopted when resolving conflicts between multiple matching rules. Specifically, when two or more rules match an incoming packet, potentially due to the presence of wildcard fields, the decision logic gives priority to the most restrictive action (deny), regardless of the number of exact matches in the rule. This design choice ensures that if a general "allow" rule (e.g., allow all UDP traffic) and a more specific "deny" rule (e.g., deny UDP from a specific IP and port) both match a packet, the firewall will default to denying the traffic. While some systems prioritize the most specific rule (i.e., the rule with the highest number of exact field matches), this implementation prioritizes a security oriented approach over a more flexible one.

## Reset Behavior

The block includes a reset mechanism that clears its internal state signals. When the `rst` signal is asserted, the module resets the output vector `fw_result` and internal registers such as `rule_result_ready` and `rule_result_value`. This ensures that any pending rule-checking operation is safely aborted and the block is ready to begin fresh evaluation upon the next rising clock edge. The firewall rules themselves are implemented as architectural constants and therefore remain unaffected by the reset, as they are not stored in resettable memory elements.

## 4.6 FIFO

### Overview

The **ASYNC\_FIFO** is an asynchronous First-In, First-Out buffer designed to store a complete Ethernet frame while it undergoes validation checks. It bridges two independent clock domains, write and read, ensuring reliable buffering and transfer of data across the GMII reception and system processing layers.

Incoming frame data is written into the FIFO under control of the write clock domain. The packet boundaries are marked using Start-of-Packet (SOP) and End-of-Packet (EOP) signals. After an entire packet has been received and stored, the FIFO awaits a validation signal from the Firewall and Frame Check Sequence (FCS) logic, jointly represented by the **FW\_RESULT** signal.

Based on this result:

- If the packet is valid and authorized ( $\text{FW\_RESULT}(1) = 1$  and  $\text{FW\_RESULT}(0) = 1$ ), the frame is forwarded through the `read_data_out` port.
- If the packet is invalid or unauthorized ( $\text{FW\_RESULT}(0) = 0$ ), the frame is silently discarded via the `dummy_data_out` port.

The FIFO exposes occupancy levels from both the write and read domains and provides full and empty status indicators to help manage flow control and prevent data loss.

### Functional Description

The **ASYNC\_FIFO** operates as a dual-clock circular buffer. During writing, incoming bytes are stored sequentially in memory until the EOP signal marks the end of the frame. The memory write pointer is incremented on each valid write and synchronized across domains using Gray code encoding to prevent metastability.

The size of the **FIFO** module can be adjusted in the testbench by changing the **FIFO DEPTH** variable. This will be more explained in the Simulation, which sizes are chosen.

Once a frame is fully stored, the block waits for  $\text{FW\_RESULT}(1)$  to be asserted, signaling that the frame has been validated. The decision in  $\text{FW\_RESULT}(0)$  determines whether the frame is to be read from the FIFO for processing or silently discarded.

On the read side, data is extracted one byte at a time on each rising edge of `rc1k`. Depending on the validation result:

- If `FW_RESULT(0) = 1`, the data is output through `read_data_out`.
- If `FW_RESULT(0) = 0`, the data is redirected to `dummy_data_out`.

The read operation continues until the internal read pointer reaches the stored end-of-packet address, at which point the reading phase ends and the FIFO prepares for the next packet.

### **Reset Behavior**

When `reset` is asserted:

- All internal pointers and state machines are reset.
- The validation tracking signals (`fw_result_active`, `packet_ready`, etc.) are cleared.
- Output signals are deasserted.

This ensures that both clock domains are safely reinitialized before further operation.

### **Clock Domain Synchronization**

The FIFO uses Gray code conversion and two-stage synchronization registers to safely transfer pointer values between clock domains. This design guarantees reliable operation and prevents data hazards caused by metastability during clock domain crossing.

## 5 Simulation and verification

### 5.1 GMII to MAC

#### Design Compilation

The VHDL design was compiled using Quartus Prime Lite Edition 20.1.1, targeting the Intel MAX 10 FPGA device (part number 10M50DAF484C7G). The compilation process completed successfully with zero errors. This process provided detailed reports regarding the analysis and synthesis, place-and-route (fitter), and timing analysis stages of the design.

**Flow Summary** Figure 3 presents the flow summary for the GMII to MAC module. The design demonstrates efficient use of FPGA resources. Specifically, the total number of logic elements utilized is less than 1% of the available resources, and only 11% of the available pins are used. These values indicate that the design is highly resource-efficient and well within the capacity of the chosen FPGA device.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed May 07 10:58:58 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MAC_RX
Top-level Entity Name	MAC_RX
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	96 / 49,760 ( < 1 % )
Total registers	27
Total pins	40 / 360 ( 11 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 3: Flow summary of the GMII to MAC module compilation.

**Analysis and Synthesis** The analysis and synthesis report, shown in Figure 4, confirms the same utilization figures for logic elements, registers, and pins as reported in the flow summary. This consistency verifies that the synthesis stage accurately translated the VHDL code into hardware without any issues or resource constraints.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed May 07 10:58:22 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MAC_RX
Top-level Entity Name	MAC_RX
Family	MAX 10
Total logic elements	93
Total registers	27
Total pins	40
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 4: Analysis and synthesis report of the GMII to MAC module compilation.

**Place and Route (Fitter)** The place-and-route (fitter) stage successfully completed, as shown in Figure 5. All necessary connections to the I/O pins were correctly established, and no routing or placement issues were encountered. This indicates a reliable physical implementation of the design within the FPGA.

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Wed May 07 10:58:49 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MAC_RX
Top-level Entity Name	MAC_RX
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	96 / 49,760 ( < 1 % )
Total registers	27
Total pins	40 / 360 ( 11 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 5: Place and route report of the GMII to MAC module compilation.



**Timing Analysis** The timing analysis evaluates the maximum operating frequency achievable under different thermal conditions. As illustrated in Figure 6, the design exceeds the specified  $F_{\max}$  constraint in one scenario, suggesting that the module is capable of operating at even higher frequencies than required. At a junction temperature of 85 °C, the maximum frequency is 243.90 MHz, while at 0 °C, it reaches 266.45 MHz. This demonstrates that the design maintains reliable timing margins under varying environmental conditions.

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	243.9 MHz	243.9 MHz	GMII_RX_CLK

(a) Slow 1200mV 85 °C

Slow 1200mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	266.45 MHz	250.0 MHz	GMII_RX_CLK

(b) Slow 1200mV 0 °C

Figure 6: Timing analysis of the GMII to MAC module under different thermal conditions.

## Testbench Simulation

This section presents the simulation results of the GMII to MAC module using ModelSim. To verify the correct functionality of the module, a dedicated testbench was developed. The testbench exercises all implemented features to ensure the block behaves as expected and can be reliably integrated into the complete system.

To thoroughly validate the design, the testbench transmits three Ethernet packets sequentially, each with a different length. Specifically, the lengths of the packets are 72, 64, and 80 bytes. These relatively short packet sizes were chosen to enhance waveform visibility in the simulation results shown in Figure 7. During simulation, the testbench verifies that the module correctly asserts the control signals indicating the start and end of a frame, and accurately flags the location of the Frame Check Sequence (FCS). This information is necessary for the downstream FCS block to identify and invert the final four bytes of each frame accordingly.

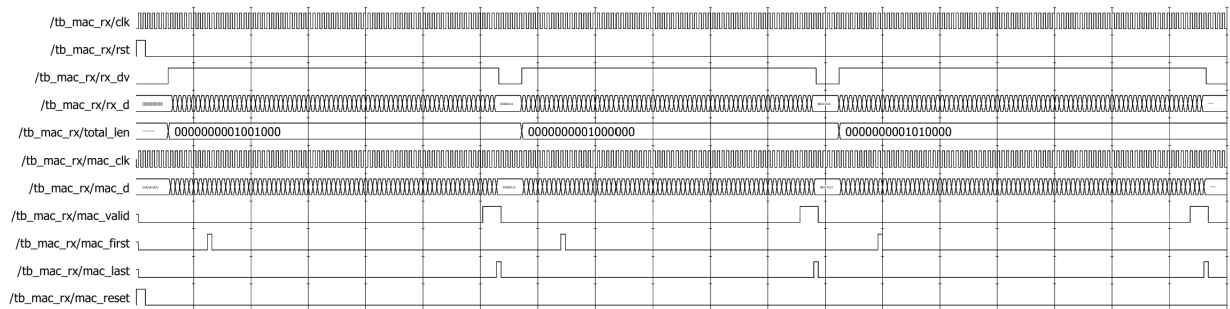


Figure 7: Simulated waveform of the GMII to MAC module showing transmission of three packets between 0 ns and 2000 ns.

Figure 7 illustrates the simulation timeline, ranging from 0 ns to 2000 ns, during which all three packets are transmitted. The signal `rx_dv` indicates valid incoming data on `rx_d`.

[illegible]

A detailed view of the beginning of the first frame is provided in Figure 8. As shown, the signal `mac_first` is asserted at the ninth byte of the input stream, which corresponds to the first byte following the Start Frame Delimiter (SFD) and marks the start of the destination MAC address. This confirms that the module correctly identifies the beginning of an Ethernet frame.

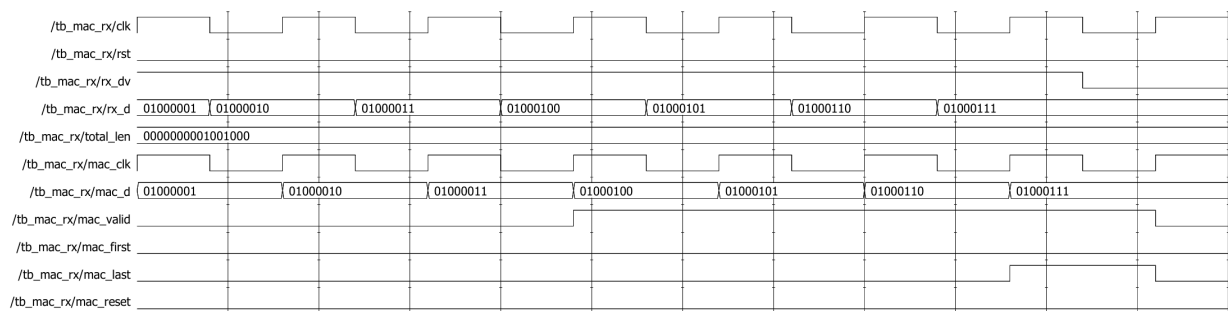


Figure 9 highlights the end of the first packet transmission. Here, the signal `mac_last` is asserted alongside `mac_valid` on the final byte of the frame, accurately identifying the end of the data payload. This ensures proper timing and coordination for the FCS processing in the following logic block.

Page 23 of 57

## 5.2 FCS

### Design Compilation

The compilation, simulation, analysis and synthesis of the FCS block were completed without any errors, confirming the correctness of the design and the corresponding test bench setup.

**Flow summary** The flow summary results, shown in the figure 10 below, reveal minimal use of logic elements, only 76, which is less than 1% of the available resources. This low utilization highlights the design's high resource efficiency and strong scalability potential. A total of 48 registers were used, indicating that only a moderate number of flip-flops were employed. Regarding pins, only 25 were used, corresponding to just 7% of the total available. As for memory bits, none were utilized, which suggests that the block does not rely on internal RAM or ROM. Similarly, embedded multipliers, PLLs, UFM, and ADC blocks were not used, indicating that no particularly complex or specialized operations are required by the design.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed May 07 10:34:01 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	fcs
Top-level Entity Name	fcs
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	76 / 49,760 ( < 1 % )
Total registers	48
Total pins	25 / 360 ( 7 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 10: Flow Summary of FCS block

**Analysis and Synthesis** The resource usage reported at this stage closely matches the results of the final compilation. A total of 76 logic elements were used, indicating minimal occupation on the FPGA. As in the compilation results, 48 registers were instantiated, reflecting the use of a moderate number of sequential elements.

25 pins were required, consistent with the interface specifications of the block. The synthesis results also show that no virtual pins, memory bits, or embedded hardware components, such as multipliers, PLLs, UFM, or ADC blocks, were utilized. This result reinforces the

conclusion that the FCS block has a compact and resource-efficient design, free of memory and computationally intensive logic. The figure 11 below shows all the obtained results.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed May 07 10:33:19 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	fcs
Top-level Entity Name	fcs
Family	MAX 10
Total logic elements	76
Total registers	48
Total pins	25
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 11: Analysis and Synthesis summary of FCS block

**Place and Route (Fitter)** The fitter report shows that 76 logic elements were used, less than 1%, confirming the design's minimal area footprint. Consistent with previous stages, the number of registers remained at 48. While, 25 pins were assigned, representing 7% of the total available on the device. Finally it confirms that no virtual pins were generated, and no memory blocks or specialized hardware resources such as multipliers, PLLs, UFM, or ADC blocks were used, confirming once again a lightweight design. The figure 12 below show the summary of the fitter obtained.

**Timing analysis** The results (figure 13) indicate that the design meets the timing requirements. For the slow 1200 85°C model, the maximum operating frequency is estimated at 478.93 MHz, while for the slow 1200 OC model, it reaches 520.83 MHz. In both cases, the design was constrained by the minimum period restriction associated with the maximum I/O toggle rate. The target clock frequency for the design was 250.0 MHz, which is well below the reported limits, confirming safe and reliable operation under the defined conditions. Additionally, the fast 1200 model shows a very low clock delay (as low as 0.108 ns), further reinforcing the timing efficiency of the implementation.

Slow 1200mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	478.93 MHz	250.0 MHz	clk

(a) Slow 1200mV 85C

Slow 1200mV OC Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	520.83 MHz	250.0 MHz	clk

(b) Slow 1200mV OC

Fast 1200mV OC Model Setup Summary			
<<Filter>>			
	Clock	Slack	End Point TNS
1	clk	0.108	0.000

(c) Fast 1200mV OC

Figure 13: Time analysis of FCS block

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Wed May 07 10:33:50 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	fcs
Top-level Entity Name	fcs
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	76 / 49,760 ( < 1 % )
Total registers	48
Total pins	25 / 360 ( 7 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 12: Fitter summary of FCS block

## Testbench simulation

Below, the simulation executed on ModelSim for the FCS block, along with its corresponding testbench, is analyzed. To demonstrate the correctness of the code, the system is first tested with a valid packet, highlighting its ability to recognize correct transmissions, and then with a corrupted packet to showcase its error detection capabilities.

First, it is important to note that the signal responsible for evaluating the packets, `fcs_error`, is a 2-bit signal. The first bit indicates whether the result is valid, while the second indicates the presence or absence of an error. Based on the logic implemented in the code, three outcomes are possible:

- *01*: result not valid and potentially erroneous. This is the default state, appearing before the complete analysis of the packet. The possibility of an error is assumed by default in this case to always account for the worst-case scenario.
- *11*: result valid and packet erroneous. This occurs when the entire packet has been analyzed and an error has been detected.
- *10*: result valid and packet error-free. This value appears when the packet has been fully analyzed and no errors are found through the CRC check.

Error detection is based on the internal signal `R`, which is updated at each clock cycle using logical operations involving the incoming data. This signal implements the CRC algorithm and determines whether the packet is corrupted. If, at the end of the packet, `R` contains only zeros, the packet is considered valid; otherwise, it is deemed erroneous.

It is essential to mention that the CRC result can only be output one clock cycle after receiving the last byte. This delay is due to VHDL's concurrent nature, where statements are

In more detail, the simulation begins with the `fcs_error` signal set to its default value of 01, as shown in the figure 14 below. When the CRC calculation starts, the internal signal R is initialized to all zeros. After the first clock cycle, an XOR operation is performed with the complemented first 8 bits. Given that the first byte sent is 00000000, the operation is performed with 11111111. Consequently, on the next clock cycle, R becomes 0000000000000000000000000011111111, and this process continues throughout the packet reception.

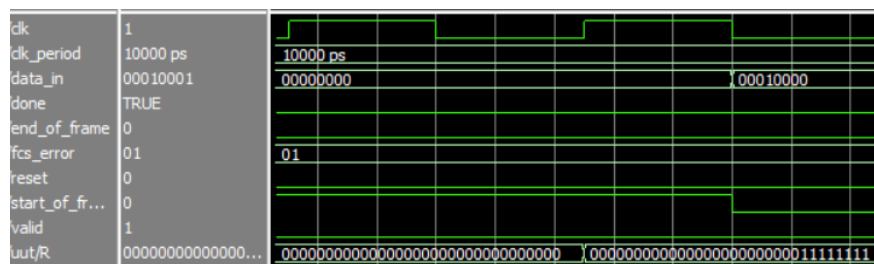


Figure 14: FCS block: start of the wave

Once the first byte of the CRC code is received and the `valid` signal is asserted, as shown in the corresponding figure 15, the code complements the incoming bytes. This behavior is observable by comparing the `data` and `data_in` signals. At the clock cycle where `end_of_frame` is high, `R` becomes all zeros, and `fcs_error` transitions to 10, indicating a valid, error-free packet.

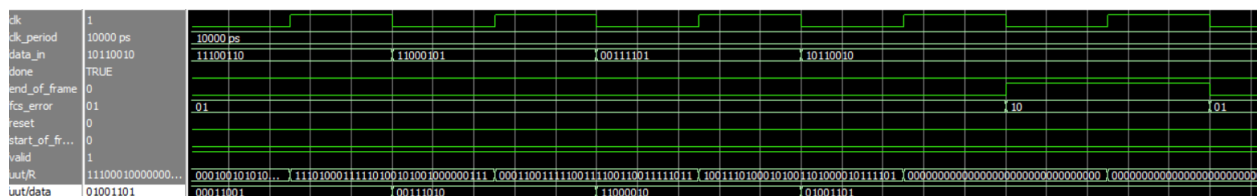


Figure 15: FCS block: end of the wave, correct result

Conversely, when the same code is executed with a corrupted packet, as illustrated in the waveform 16, the final bytes are altered, no longer being *00111101* and *10110010*, but rather *00110101* and *0001001*. This leads to an incorrect CRC result, causing the `fcs_error` signal to take the value 11, meaning the result is valid but the packet contains an error.

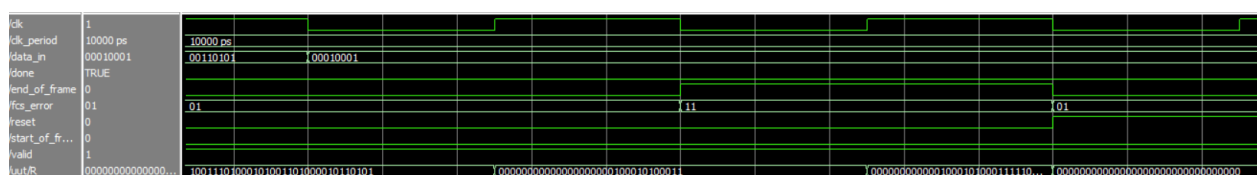


Figure 16: FCS block: end of the wave, erroneous result

Finally, in both cases, the signal is reset to 01, indicating an invalid state, and R returns to all zeros, ensuring the system is ready to correctly process the next packet.

### 5.3 MAC RX Control

#### Design Compilation

The compilation, simulation, analysis and synthesis of the MAC-RX Control block were completed without any errors, confirming the correctness of the design and the corresponding test bench setup.

**Flow summary** According to the flow summary in figure 17 below, the design makes very limited use of the available FPGA resources. Only 64 logic elements were used out of 49,760 (<1%), indicating a compact and efficient implementation. A total of 46 registers were employed, reflecting a moderate use of sequential logic.

The block uses 41 pins, which corresponds to approximately 11% of the device's pin resources, which is higher than previous designs (FCS). As with the previous block, no memory bits, embedded multipliers, PLLs, UFM, or ADC blocks were utilized, suggesting that the design avoids complex or memory-demanding tasks. These results confirm that MAC\_RX\_CONTROL is a lightweight, register-based design with modest area requirements and a relatively rich I/O interface.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed May 07 09:45:29 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MAC_RX_CONTROL
Top-level Entity Name	MAC_RX_CONTROL
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	64 / 49,760 ( < 1 % )
Total registers	46
Total pins	41 / 360 ( 11 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 17: Flow Summary of MAC-RX Control block

**Analysis and Synthesis** The synthesis report indicates that 63 logic elements were used, demonstrating a compact logic implementation. The design includes 46 registers, pointing to a moderate use of flip-flops.

A total of 41 I/O pins were used, which is slightly high compared to the logic resource usage,

and suggests that the block interfaces with a broader set of external signals.

No virtual pins, memory bits, or specialized hardware resources such as multipliers, PLLs, UFM, or ADC blocks were utilized, reinforcing the design's lightweight and logic-based nature.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed May 07 09:45:01 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MAC_RX_CONTROL
Top-level Entity Name	MAC_RX_CONTROL
Family	MAX 10
Total logic elements	63
Total registers	46
Total pins	41
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 18: Analysis and Synthesis summary of MAC-RX Control block

**Place and Route (Fitter)** As we can see in the figure 19 below, the fitter report confirms that 64 logic elements were used, which is less than 1% of the available resources, indicating a compact implementation. The number of registers remained at 46, in line with the synthesis stage, while 41 pins were assigned, accounting for 11% of the available device pins, this pin usage reflects the relatively rich interface of the module.

As in the previous stages, no virtual pins, on-chip memory, or specialized hardware resources (such as multipliers, PLLs, UFM, or ADC blocks) were used. This confirms that the design is entirely based on simple logic and register elements, with no need for complex or resource-intensive hardware features.

**Timing analysis** The analysis results (figure 20) show that, under the slow 1200 mV 85° C model, the maximum operating frequency for the MAC\_RX\_CLK signal is 252.14 MHz. Under the slow 1200 mV OC model, this value slightly increases to 275.18 MHz. Both values represent safe operational frequencies under conservative conditions and confirm that the design is not timing-critical at moderate clock rates.

In the fast 1200 mV OC model, the analyzer reports a positive slack of 18.388 ns, with no negative timing at any endpoints. This indicates that the design easily meets the required timing constraints, leaving a safe margin for operation, even under faster clock conditions. These results confirm that MAC\_RX\_CONTROL operates reliably within its expected frequency range, with no violations of setup or hold timing, and significant headroom for potential clock speed increases if needed.



Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Wed May 07 09:45:20 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	MAC_RX_CONTROL
Top-level Entity Name	MAC_RX_CONTROL
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	64 / 49,760 ( < 1 % )
Total registers	46
Total pins	41 / 360 ( 11 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 19: Fitter summary of MAC-RX Control block

Slow 1200mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	252.14 MHz	250.0 MHz	MAC_RX_CLK

(a) Slow 1200mV 85C

Slow 1200mV OC Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	275.18 MHz	250.0 MHz	MAC_RX_CLK

(b) Slow 1200mV OC

Fast 1200mV OC Model Setup Summary			
<<Filter>>			
	Clock	Slack	End Point TNS
1	MA...LK	18.338	0.000

(c) Fast 1200mV OC

Figure 20: Time analysis of FCS block

## Testbench simulation

The simulation of the *MAC\_RX\_CONTROL* block was executed using the corresponding testbench in ModelSim, and the resulting waveform is analyzed below. The analysis begins with the initial section, followed by the final part. As previously discussed, this block is responsible for counting the bytes transmitted to the next block, which is essential for correctly identifying the various fields within the packet.

In the initial part, shown in figure 21, we can observe that when the signal `mac_rx_valid` is equal to 1, indicating the reception of the first byte of the packet, the signal `mac_rx_err` takes the value *01*. In this configuration, the first bit (*0*) signifies that the value is not valid yet, as this signal only carries a valid meaning at the end of packet analysis, as explained in the previous section. While, the second bit (*1*) indicates the presence of an error. This choice, as mentioned earlier, reflects a design strategy that assumes the worst-case scenario. Next, we observe that the signals `mac_rxd` and `data_fw` carry identical data, with `data_fw` being delayed by one clock cycle. Specifically, `mac_rxd` represents the incoming data stream, while `data_fw` delivers the corresponding output to the subsequent block. A similar shift applies to the signals `mac_rx_err` and `fw_out`, which transmit the error information, as well

as to the signals indicating the start and end of the packet.

We also note that the first byte received is *00000000*, corresponding to hexadecimal *00*, and the second is *00100000*, corresponding to *10*, exactly as specified in the testbench. Additionally, the `line_number` signal, which keeps track of the byte count within the packet, starts at *0000000000000000* and increments to *0000000000000001* upon sending the first byte, demonstrating the correct operation of the counter logic.

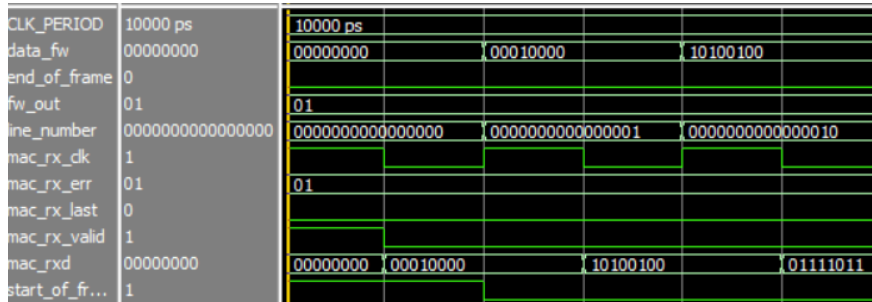


Figure 21: MAC-RX Control block: start of the wave

In the final part of the waveform, shown in figure 22, we observe that at the end of the packet, right after the signal `mac_rx_last` is set to 1 for the last byte, the signal `mac_rx_err` takes the value *10*. This indicates that the value is now valid and that no errors are present. As before, all signals, including the data path, the start and end of frame indicators, are delayed by one clock cycle between input and output. Finally, we observe that upon transmitting the last byte, *10110010* (hexadecimal *B2*), the `line_number` signal reaches the value *0000000000111111*, which corresponds to 63 in decimal. This value precisely matches the total number of bytes transmitted. Immediately afterward, the counter resets to its initial value *0000000000000000* in preparation for the next packet, once again confirming the correctness of the implementation.

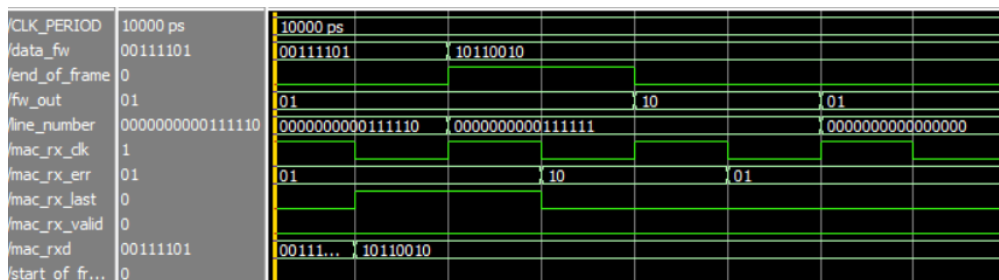


Figure 22: MAC-RX Control block: end of the wave

## 5.4 Packet Analyzer

### Design Compilation

To verify the correct implementation and resource usage of the `PacketAnalyzer` module, the design was compiled using Intel Quartus Prime Lite Edition v20.1.1. The target board is an **Intel MAX 10 FPGA (10M50DAF484C7G)**.

**Flow Summary** Figure 29 shows the Flow Summary after a successful compilation. As it can be seen from the image, the most relevant aspect is the very limited logic usage.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Fri May 02 11:26:00 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	PacketAnalyzer
Top-level Entity Name	PacketAnalyzer
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	140 / 49,760 ( < 1 % )
Total registers	120
Total pins	146 / 360 ( 41 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 23: Flow Summary after compilation

**Analysis and Synthesis** As shown in Figure 30, the Analysis and Synthesis stage completed successfully. The logic utilization and register count remain consistent with the flow summary, confirming that the synthesis tool correctly translated the HDL code into hardware logic without resource conflicts or overuse.

**Place and Route (Fitter)** The Place and Route phase was also completed without issues, as shown in Figure 31. Since the design is relatively small and simple, the routing process was fast and efficient. No timing or placement violations were detected.

**Timing Analysis** The Timing Analyzer results in Figures 32 and 33 report the maximum frequency ( $F_{\max}$ ) achievable by the design at two different temperature conditions:

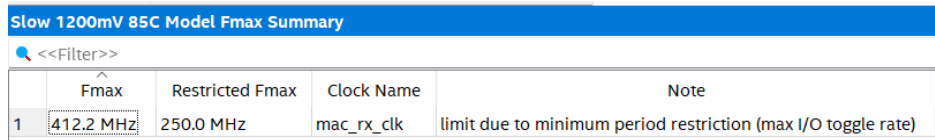
- 85 °C: 412.2 MHz

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Fri May 02 11:24:42 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	PacketAnalyzer
Top-level Entity Name	PacketAnalyzer
Family	MAX 10
Total logic elements	144
Total registers	120
Total pins	146
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 24: Analysis and Synthesis results

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Fri May 02 11:25:52 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	PacketAnalyzer
Top-level Entity Name	PacketAnalyzer
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	140 / 49,760 ( < 1 % )
Total registers	120
Total pins	146 / 360 ( 41 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 25: Place and Route results

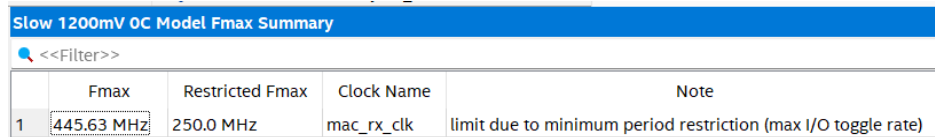


Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	412.2 MHz	250.0 MHz	mac_rx_clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 26: Timing Analyzer result at 85 °C



Slow 1200mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	445.63 MHz	250.0 MHz	mac_rx_clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 27: Timing Analyzer result at 0 °C

- 0 °C: 445.6 MHz

In the high temperature scenario (85 degrees Celsius), the maximum speed achieved was 412.2 MHz. This speed was limited to 250 MHz due to the minimum period restriction, specifically the maximum I/O toggle rate. In the low-temperature scenario (0 degrees Celsius), the maximum speed achieved was 445.63 MHz. Similar to the previous model, this speed was also limited to 250MHz due to the same minimum period restriction.

## Testbench Simulation

To verify the correctness of the `PacketAnalyzer` module, a functional simulation was carried out using a testbench. The test set as input an Ethernet frame containing an IPv4 packet. The objective is to evaluate the extraction of header fields and the correct behavior of control signals. Figure 28 shows the waveform of the main signals during the simulation.

The clock signal `mac_rx_clk` operates at 100 MHz. The input stream `data_fw` is driven sequentially with the bytes of the frame, and the `line_number` signal increments accordingly. The `start_of_frame` signal is high with the first valid byte of the frame, causing the FSM to transition from the `IDLE` state to the `CAPTURING` state. As visible in the waveform, the FSM remains in this state while parsing the packet, and returns to `IDLE` when all the relevant field have been extracted. The `end_of_frame` signal is asserted at the final byte, indicating the conclusion of the frame processing.

During the capture phase, the module correctly decodes the IPv4 source and destination addresses, which appear on `source_ip` and `dest_ip`, respectively. The `source_port`, `dest_port`, and `protocol` fields are also extracted and made available. The waveform shows that the source IP address is captured between byte index 26 and 29, while the destination address is captured between byte 30 and 33. At `line_number` = 23, the protocol field is extracted. The value present on the `protocol` signal is 00010001, corresponding to the decimal value 17, which identifies the UDP protocol. Similarly, the source port is acquired at bytes 34 and 35, and the destination port at bytes 36 and 37. Once all fields are parsed, the

`data_ready` signal is asserted, indicating that the extracted information is valid and ready for processing by the *CheckRules* block.

To replicate realistic timing behavior, particularly in relation to the FCS module, the test-bench sets the `fw_out` signal to "10" exactly one clock cycle after the last byte of the packet has been sent. This delay reflects the expected functionality of the FCS block, which requires the entire frame, including the CRC field, before it can determine the validity of the packet.

The output stream `fifo_data` forward the input `data_fw`, and the associated control signals `fifo_sof` and `fifo_eof` correctly identify the start and end of the frame for the FIFO block.

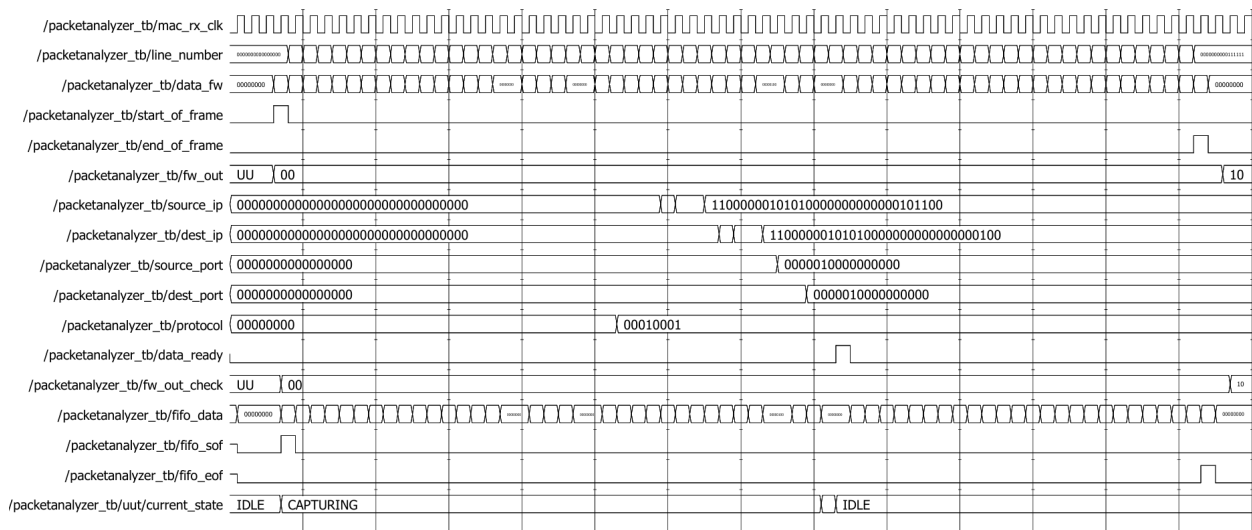


Figure 28: Simulation waveform of the PacketAnalyzer module.

## 5.5 Check Rules

### Design Compilation

The VHDL module `CheckRules` was compiled successfully using Quartus Prime Lite Edition 20.1.1 for the Intel MAX 10 FPGA (device 10M50DAF484C7G). The compilation process included analysis and synthesis, place and route, and timing analysis.

**Flow Summary** As shown in Figure 29, the compilation was completed without errors. The design uses only 48 logic elements out of the 49,760 available on the device, which is less than 1% of total capacity. This confirms that the block is very lightweight and does not require significant hardware resources.

**Analysis and Synthesis** The synthesis results (Figure 30) confirm the minimal use of internal resources: 47 logic elements and 4 registers. The design also uses 111 physical pins, equal to 31% of the total available. No memory blocks, multipliers, or special hardware modules are used.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sat May 03 15:21:11 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	CheckRules
Top-level Entity Name	CheckRules
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	48 / 49,760 ( < 1 % )
Total registers	4
Total pins	111 / 360 ( 31 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 29: Flow summary of the `CheckRules` module compilation.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Sat May 03 15:20:22 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	CheckRules
Top-level Entity Name	CheckRules
Family	MAX 10
Total logic elements	47
Total registers	4
Total pins	111
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 30: Analysis and synthesis report for the `CheckRules` block.

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Sat May 03 15:21:02 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	CheckRules
Top-level Entity Name	CheckRules
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	48 / 49,760 ( < 1 % )
Total registers	4
Total pins	111 / 360 ( 31 % )
Total virtual pins	0
Total memory bits	0 / 1,677,312 ( 0 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 31: Place and route results for the **CheckRules** block.

**Place and Route (Fitter)** The place and route stage (Figure 31) completed successfully. The resource usage remains consistent with the synthesis results, and no routing or fitting issues were detected.

**Timing Analysis** The Timing Analyzer results in Figures 32 and 33 report the maximum frequency ( $F_{\max}$ ) achievable by the design at two different temperature conditions:

- 85°C: 919.12 MHz
- 0°C: 1009.08 MHz

In both cases, the design supports clock frequencies above 900 MHz (Figures 32 and 33), while the actual clock constraint is set to 250 MHz. This ensures a large timing margin and stable operation.

Slow 1200mV 85C Model Fmax Summary				
<<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	919.12 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 32: Timing analysis result at 85°C.

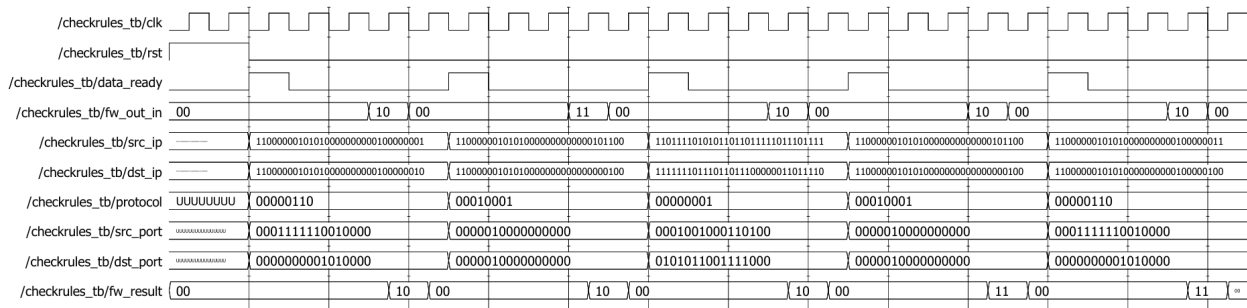


Slow 1200mV OC Model Fmax Summary				
	Fmax	Restricted Fmax	Clock Name	Note
1	1009.08 MHz	250.0 MHz	clk	limit due to minimum period restriction (max I/O toggle rate)

Figure 33: Timing analysis result at 0°C.

## Testbench Simulation

The simulation of the `CheckRules` module was performed to verify the correct behavior of the rule-matching logic under different network packet configurations. The testbench, shown in the appendix A, drives the main inputs of the unit under test (UUT), including IP addresses, ports, and protocol types, and provides timing control over the `clk`, `rst`, and `data_ready` signals. The simulation waveform is illustrated in Figure 34.

Figure 34: Waveform showing the simulation of the `CheckRules` module.

The simulation involves testing the relevant fields, extracted by the Packet Analyzer block, from five different input packets. The objective is to verify whether the output aligns with the expected results based on the firewall table shown in 2. Each test case begins with the assertion of the `data_ready` signal for one clock cycle, during which the corresponding values for `src_ip`, `dst_ip`, `protocol`, `src_port`, and `dst_port` are set. After two clock cycles, the `fw_out_in` signal is assigned either 10 or 11 to indicate whether the packet has successfully passed the Frame Check Sequence and is ready for rule evaluation. The output signal `fw_result` reflects both the rule evaluation outcome and the FCS verification status, as detailed in the block description.

The following section provides a detailed analysis of each of the five cases individually.

**Blocked TCP Packet, Valid FCS:** The packet matches exactly a rule explicitly set to block TCP traffic from 192.168.1.1 to 192.168.1.2 on port 80. With valid FCS (`fw_out_in` = "10"), the result is correctly `fw_result` = "10"(BLOCK). It is important to note here that even though there is the general rule to allow TCP packets, the more specific blocking rule takes precedence, ensuring that the packet is denied (see Table 3).

Field	Value
Source IP	192.168.1.1
Destination IP	192.168.1.2
Protocol	TCP (06)
Source Port	8080
Destination Port	80

Table 3: Blocked TCP packet due to explicit deny rule

**Allowed UDP Packet, Invalid FCS:** An UDP packet matching a specific allow rule, but with an invalid FCS (`fw_out_in` = "11"). The output is `fw_result` = "10" (BLOCK), indicating that FCS validation resulted with an error has forced the module to drop the packet (see Table 4).

Field	Value
Source IP	192.168.0.44
Destination IP	192.168.0.4
Protocol	UDP (11)
Source Port	1024
Destination Port	1024

Table 4: Allowed UDP packet matching predefined rule

**No Matching Rule, Valid FCS:** A packet that does not match any explicitly defined rules is evaluated. Despite its valid FCS (`fw_out_in` = "10"), the firewall defaults to a DENY policy, resulting in the packet being blocked (`fw_result` = "10") (see Table 5).

Field	Value
Source IP	192.168.0.50
Destination IP	192.168.0.100
Protocol	UDP (11)
Source Port	5000
Destination Port	53

Table 5: Packet rejected due to absence of matching rule

**Allowed UDP Packet, Valid FCS:** This test case repeats the scenario of Test 2, but with a valid FCS. Since the packet matches an allow rule and its integrity is verified, the firewall correctly classifies it as allowed (`fw_result` = "10") (see Table 6).

Field	Value
Source IP	192.168.0.44
Destination IP	192.168.0.4
Protocol	UDP (11)
Source Port	1024
Destination Port	1024

Table 6: Allowed UDP packet with a valid rule

**TCP Packet Allowed, Valid FCS:** This case evaluates a general TCP packet, which is permitted under a rule that allows all TCP traffic unless a more restrictive rule explicitly blocks it. Since no conflicting block rules apply and the FCS is valid (`fw_out_in = "10"`), the result shows (`fw_result = "10"`) (ALLOW). This confirms that less specific allow rules function correctly when no higher-priority blocking rules exist (see Table 7).

Field	Value
Source IP	192.168.1.3
Destination IP	192.168.1.4
Protocol	TCP (06)
Source Port	8080
Destination Port	80

Table 7: Allowed TCP packet under general allow rule

In all five cases, the simulation demonstrates that the CheckRules block properly evaluates the rule set, respects priority order, checks FCS validity before finalizing the decision, and correctly outputs the firewall result. The use of `fw_out_in` ensures that even if a rule matches, the frame is not accepted unless FCS is confirmed valid.

## 5.6 FIFO

### Design Compilation

The VHDL design was compiled using Quartus Prime Lite Edition 20.1.1, targeting the Intel MAX 10 FPGA device (part number 10M50DAF484C7G). The compilation process completed successfully with zero errors. This process provided detailed reports regarding the analysis and synthesis, place-and-route (fitter), and timing analysis stages of the design.

**Flow Summary** Figure 35 presents the flow summary for the FIFO module. The design demonstrates efficient use of FPGA resources. Specifically, the total number of logic elements utilized is less than 1% of the available resources, 183 out of 49,760. Only 16% of the available pins are used, 56 out of 360. These values indicate that the design is highly resource-efficient.

and well within the capacity of the chosen FPGA device. The total memory bits that are used are 12.176 out of 1.677.312, which is less than 1 % of the available ones.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed May 07 10:38:14 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	FIFO
Top-level Entity Name	async_fifo
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	183 / 49,760 ( < 1 % )
Total registers	124
Total pins	56 / 360 ( 16 % )
Total virtual pins	0
Total memory bits	12,176 / 1,677,312 ( < 1 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 35: Flow summary of the FIFO module compilation.

**Analysis and Synthesis** The analysis and synthesis report, shown in Figure 36, confirms the same utilization figures for logic elements, registers, pins and total memory bits as reported in the flow summary. This consistency verifies that the synthesis stage accurately translated the VHDL code into hardware without any issues or resource constraints.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed May 07 10:37:41 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	FIFO
Top-level Entity Name	async_fifo
Family	MAX 10
Total logic elements	200
Total registers	124
Total pins	56
Total virtual pins	0
Total memory bits	12,176
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 36: Analysis and synthesis report of the FIFO module compilation.

**Place and Route (Fitter)** The place-and-route (fitter) stage successfully completed, as shown in Figure 37. All necessary connections to the I/O pins were correctly established, and no routing or placement issues were encountered. This indicates a reliable physical implementation of the design within the FPGA.

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Wed May 07 10:38:06 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	FIFO
Top-level Entity Name	async_fifo
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	183 / 49,760 ( < 1 % )
Total registers	124
Total pins	56 / 360 ( 16 % )
Total virtual pins	0
Total memory bits	12,176 / 1,677,312 ( < 1 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 37: Place and route report of the FIFO module compilation.

**Timing Analysis** The timing analysis assesses the maximum operating frequencies of the FIFO module under varying thermal conditions. This module features two distinct clock domains: the read clock (`rclk`) and the write clock (`wclk`). Figure 38 illustrates the maximum achievable frequencies for both clocks across different temperatures. The results indicate that at no point does the design exceed the specified  $F_{\max}$  constraint for either clock. At a junction temperature of 85°C, the maximum frequency of `wclk` is 149.79 MHz, while `rclk` reaches 177.02 MHz. When the temperature is reduced to 0°C, `wclk` can operate at up to 163.19 MHz, and `rclk` reaches 192.49 MHz. This demonstrates that the read clock domain is capable of operating at a higher frequency than the write clock domain. However, in the current simulation setup, both the write and read clocks are driven by the same clock source. Consequently, the maximum usable frequency is limited by the slower of the two, which is `wclk` at 149.79 MHz. This frequency is still well within acceptable limits, as the overall system is designed to operate at 125 MHz to support a 1 Gbit/s Ethernet line. Thus, the timing results confirm that the FIFO module can reliably meet the performance requirements of the target application.

Slow 1200mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	149.79 MHz	149.79 MHz	wclk
2	177.02 MHz	177.02 MHz	rclk

(a) Slow 1200mV 85 °C

Slow 1200mV 0C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	163.19 MHz	163.19 MHz	wclk
2	192.49 MHz	192.49 MHz	rclk

(b) Slow 1200mV 0 °C

Figure 38: Timing analysis of the FIFO module under different thermal conditions.

## Testbench Simulation

This section presents the simulation results of the FIFO module using ModelSim. To verify the correct functionality of the module, a dedicated testbench was developed. The testbench exercises all implemented features to ensure the block behaves as expected and can be reliably integrated into the complete system.

To verify the functionality of the FIFO module, a simulation was conducted in which two packets are transmitted sequentially. The simulation spans from 0 ns to 300 ns. For improved clarity and visualization in the waveform, both packets were kept shorter than standard Ethernet frames, each consisting of only 3 bytes. The FIFO depth was configured to 16 entries, which is sufficient for this test scenario, as the module is not required to buffer large amounts of data. Incoming data is stored in the FIFO until the EOP (End of Packet) flag is asserted. Figure 39 displays the simulation results. In the first half of the waveform, the first packet is correctly accepted and stored in the FIFO. This is indicated by the FW\_RESULT signal transitioning to 11. Afterward, the data is successfully read out via the read\_data\_out signal, confirming that the packet has been forwarded correctly through the module. The second packet is handled similarly in terms of storage, but it is subsequently sent to the dummy\_data\_out output. This behavior is determined by the FW\_RESULT signal being set to 10, which indicates that the packet is to be discarded. As a result, the FIFO is emptied once again after processing the two packets. Additionally, the simulation illustrates the behavior of key status signals, such as fifo\_empty, fifo\_occu\_in, and fifo\_occu\_out, which track the occupancy and availability status of the FIFO during the test.

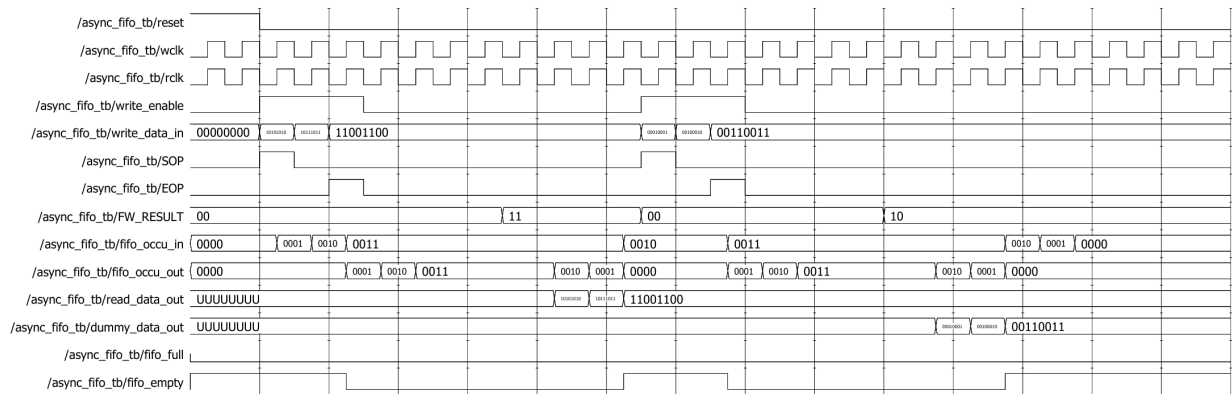


Figure 39: Waveform showing the simulation of FIFO module.

## 6 Firewall System

### 6.1 Design Compilation

The complete VHDL design was compiled using Quartus Prime Lite Edition 20.1.1, targeting the Intel MAX 10 FPGA device (part number 10M50DAF484C7G). The compilation completed successfully without any errors. As part of this process, Quartus generated detailed reports covering key stages of the implementation flow, including analysis and synthesis, placement and routing (fitter), and timing analysis. These reports provide essential insights into the design's resource utilization and overall performance.

#### Flow Summary

Figure 40 shows the flow summary of the complete **Firewall** system after compilation. The design makes highly efficient use of the available FPGA resources. Specifically, only 558 out of 49,760 logic elements are utilized, which corresponds to approximately 1% of the available capacity. In terms of I/O utilization, just 47 out of 360 pins are used, accounting for 13% of the total. The design also consumes 12,176 memory bits out of the 1,677,312 available, which is again less than 1%. Those are clearly from the FIFO module. These figures confirm that the design is well within the resource limits of the Intel MAX 10 FPGA and demonstrates a high level of resource efficiency.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Wed May 07 14:01:00 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Full_System
Top-level Entity Name	TopPackCheck
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	558 / 49,760 ( 1 % )
Total registers	407
Total pins	47 / 360 ( 13 % )
Total virtual pins	0
Total memory bits	12,176 / 1,677,312 ( < 1 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 40: Flow summary of the **Firewall** system compilation.



## Analysis and Synthesis

The analysis and synthesis report, shown in Figure 41, confirms the same resource usage figures as previously reported in the flow summary. This includes the utilization of logic elements, registers, I/O pins, and on-chip memory blocks. Specifically, the number of logic elements used remains at 558, with a consistent allocation of 47 pins and 12,176 memory bits. This alignment between the synthesis and flow summary results demonstrates that the VHDL description of the `Firewall system` was accurately interpreted and translated into a functional hardware design. The successful synthesis stage indicates that the design is structurally and functionally complete, with no syntax errors, missing definitions, or unsupported constructs. Furthermore, the synthesis process did not encounter any resource limitations or conflicts, confirming that the hardware description fits comfortably within the available resources of the Intel MAX 10 FPGA device. This step also verifies that all logic blocks were correctly inferred and optimized by the synthesis tool, allowing for efficient mapping onto the physical hardware. The consistent results further suggest that no unexpected behavior occurred during elaboration or optimization, which would otherwise introduce discrepancies in resource usage. Overall, the synthesis report provides strong evidence that the system is well-designed and ready for physical implementation.

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Wed May 07 14:00:35 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Full_System
Top-level Entity Name	TopPackCheck
Family	MAX 10
Total logic elements	631
Total registers	407
Total pins	47
Total virtual pins	0
Total memory bits	12,176
Embedded Multiplier 9-bit elements	0
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figure 41: Analysis and synthesis report of the `Firewall system` compilation.

## Place and Route (Fitter)

The place-and-route (fitter) process, illustrated in Figure 42, was completed successfully without any errors or routing issues. During this stage, the Quartus tool mapped the synthesized logic onto the physical resources of the FPGA and established all necessary interconnections between logic elements and I/O pins. The absence of routing congestion or placement warnings indicates that the design is not only functionally valid but also physically optimized for the selected device. All required signal paths to the I/O pins were correctly established, confirming that the pin assignments defined in the VHDL and constraint files were properly handled. Additionally, the routing engine was able to resolve all paths within the timing and layout constraints of the Intel MAX 10 FPGA. This implies that the physical layout of the **Firewall system** does not introduce any bottlenecks or delays that would interfere with proper operation. The successful completion of the fitter stage also ensures that the design complies with the device's electrical and structural rules. It validates that all components, including clock signals and data paths, are appropriately placed and connected in a way that supports robust and reliable system behavior. This confirms the system's readiness for further steps such as timing verification and hardware testing.

Fitter Summary	
🔍 <<Filter>>	
Fitter Status	Successful - Wed May 07 14:00:54 2025
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	Full_System
Top-level Entity Name	TopPackCheck
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	558 / 49,760 ( 1 % )
Total registers	407
Total pins	47 / 360 ( 13 % )
Total virtual pins	0
Total memory bits	12,176 / 1,677,312 ( < 1 % )
Embedded Multiplier 9-bit elements	0 / 288 ( 0 % )
Total PLLs	0 / 4 ( 0 % )
UFM blocks	0 / 1 ( 0 % )
ADC blocks	0 / 2 ( 0 % )

Figure 42: Place and route report of the **Firewall system** compilation.

## Timing Analysis

The timing analysis evaluates the maximum achievable operating frequency ( $F_{\max}$ ) of the `Firewall` system under different thermal conditions. The system operates with a single clock signal, `clk`, which governs the timing behavior of all synchronous components within the design.

As illustrated in Figure 43, the timing report provides estimates of the maximum clock frequency at two temperature extremes: 85 °C and 0 °C. At a junction temperature of 85 °C, the maximum frequency is 156.08 MHz, while at 0 °C, it reaches 169.12 MHz. These values represent the highest frequencies at which the design can reliably operate without violating setup or hold time constraints under the respective thermal conditions. Although the design does not meet the 250 MHz  $F_{\max}$  used in simulation benchmarks, this is not a limitation for the actual deployment. The intended target frequency for the system is 125 MHz, which is sufficient to support a 1 Gbit/s Ethernet data rate—the core performance requirement of the `Firewall` system.

Therefore, the results of the timing analysis confirm that the design comfortably meets its real-world operational requirements. The reported timing margins ensure that the system will function correctly within the expected temperature range and usage scenario, providing stable and reliable performance.

Slow 1200mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	156.08 MHz	156.08 MHz	clk

(a) Slow 1200mV 85 °C

Slow 1200mV 0C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	169.15 MHz	169.15 MHz	clk

(b) Slow 1200mV 0 °C

Figure 43: Timing analysis of the `Firewall` system under different thermal conditions.

## 6.2 Testbench Simulation

This section presents the functional simulation of the **Firewall system** using the ModelSim environment. In order to thoroughly validate the behavior of the complete module, a dedicated testbench was developed. The purpose of this testbench is to exercise all essential features and verify that each subsystem operates correctly and cohesively within the full design.

To test the system's behavior under different conditions, the simulation was structured into three distinct scenarios, each modifying either the configuration or the packet data. These scenarios were designed to reflect realistic use cases and edge cases, ensuring that both the firewall logic and the associated processing blocks behave as expected. In the first scenario, no forwarding rules are defined in the firewall's rule set. Consequently, when an Ethernet packet is introduced into the system, it is blocked by the firewall logic, demonstrating correct enforcement of default-deny behavior. In the second scenario, a rule is added that explicitly permits the previously blocked packet. As a result, the packet successfully traverses the firewall and continues through the system. This confirms that the rule-matching mechanism and packet forwarding logic are functioning correctly. In the final scenario, the packet contents are modified in such a way that the firewall module would still allow the packet to pass. However, the FCS module, responsible for verifying the integrity of incoming data, detects an error. Consequently, the packet is blocked and discarded at this later stage. This test confirms that the FCS module.

To provide a clearer overview, the simulation report focuses on the key time intervals that reflect the most critical moments in the operation of the **Firewall system**. Specifically, it highlights the start of the simulation (0-200 ns), the evaluation results of both the firewall and the FCS checks (700-900 ns), and the final phase (1500-1700 ns), which confirms whether the FIFO module successfully empties its contents as expected. These specific windows were chosen to reduce visual clutter and emphasize functional milestones. Notably, the timing remains consistent across all test cases, since the transmitted packets are of identical length and thus require the same amount of processing time by the system.

### Firewall blocks Packet

Figure 44 illustrates the Ethernet packet used to test the functionality of the **Firewall system**. This packet includes a valid Frame Check Sequence (FCS), ensuring that it is not rejected by the FCS module due to data integrity errors. To facilitate interpretation, the packet fields are clearly labeled, allowing for an easier visual breakdown of its structure. This particular packet is used in the first two simulation scenarios to evaluate the firewall's response both with and without defined forwarding rules.

```

-- Frame to test (72 bytes = 8 preamble + 60 dati + 4 CRC)
type frame_array is array (0 to 71) of std_logic_vector(7 downto 0);
constant test_frame : frame_array := (
    x"55", x"55", x"55", x"55", x"55", x"55", x"55", x"D5", --preamble
    x"00", x"10", x"A4", x"7B", x"EA", x"80", -- Dest MAC
    x"00", x"12", x"34", x"56", x"78", x"90", -- Src MAC
    x"08", x"00", -- EtherType: IPv4
    x"45", x"00", x"00", x"2E", x"B3", x"FE", x"00", x"00", x"80", -- IP header start
    x"11", -- Byte 23: Protocol (UDP = 17 = x"11")
    x"05", x"40", -- Checksum etc.
    x"C0", x"A8", x"00", x"2C", -- src ip:
    x"C0", x"A8", x"00", x"04", -- dst ip:
    x"04", x"00", -- src port:
    x"04", x"00", -- dst port:
    x"00", x"1A", x"2D", x"E8", -- Payload...
    x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
    x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
    x"10", x"11", -- More payload
    x"E6", x"C5", x"3D", x"B2" -- FCS
);

```

Figure 44: Test packet used in the `Firewall` system simulation, including valid FCS.

In the first simulation scenario, no firewall rules are configured. As a result, the packet is expected to be blocked by the `Firewall` system, demonstrating the correct default-deny behavior when no rule matches are found. Figure 45 presents the initial phase of this simulation, covering the time interval from 0 to 200 ns. At the beginning, the system is reset and remains idle for 20 ns. After the reset period, the `valid` signal is asserted, and packet transmission begins. The simulation proceeds with the transmission of the 7-byte preamble, followed by the Start Frame Delimiter (SFD). The ninth byte, which corresponds to the first byte of the destination MAC address, marks the beginning of the actual Ethernet frame. Each byte occupies a duration of 10 ns. During this phase, the `fifo_write_enable` signal is activated, enabling the FIFO to begin writing incoming data to memory. This activation is precisely aligned with the arrival of the first byte of the preamble. Other output signals remain inactive at this stage, as no decision has yet been made by the firewall or the downstream modules.

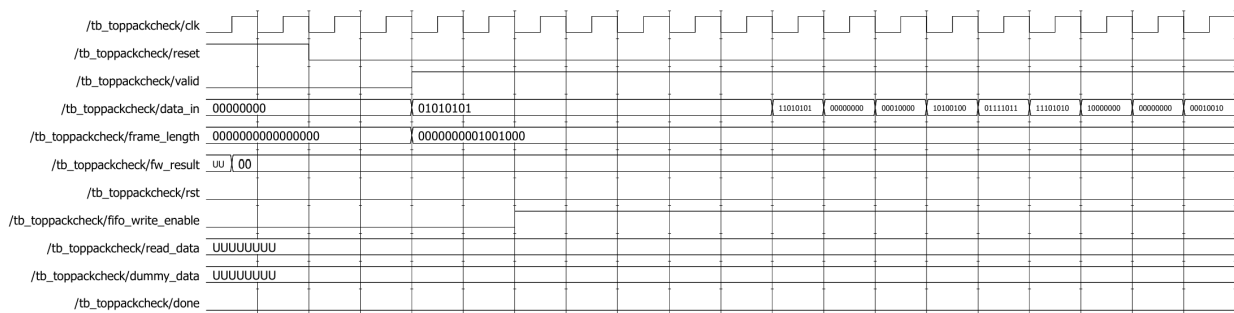


Figure 45: Simulation start with no firewall rules configured (0-200 ns).

Figure 46 displays the middle phase of the simulation, covering the time interval from 700 ns to 900 ns. During this segment, the decision of the `Check Rules` module becomes visible. The result clearly indicates that the packet does not match any active firewall rules and is therefore blocked by the `Firewall` system. Prior to this decision, the `fifo_write_enable` signal is deasserted, indicating that the entire packet has been written into the FIFO and the end of the Ethernet frame has been reached.

Once the blocking decision is made, the system begins redirecting the data to the `dummy_data` output interface. This behavior confirms that the packet is being discarded as intended, demonstrating that the system correctly enforces default-deny behavior in the absence of matching rules.

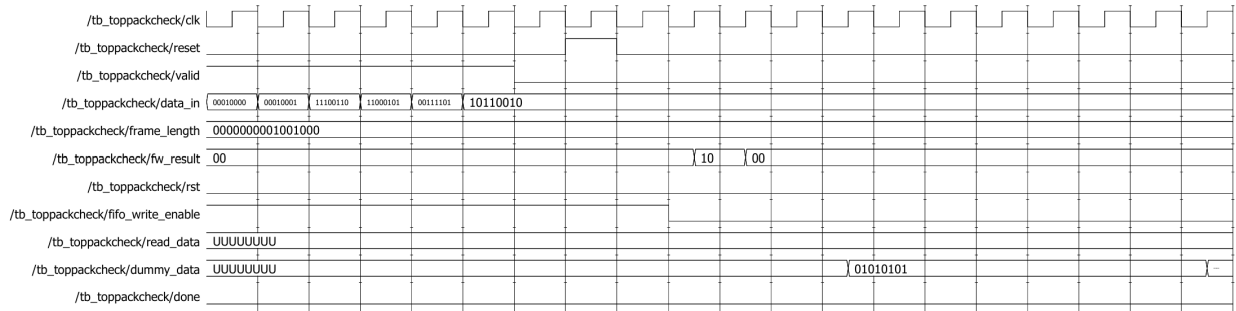


Figure 46: Mid-simulation: Packet evaluation and blocking with no firewall rules (700-900 ns).

Figure 47 captures the final stage of the simulation, spanning from 1500 ns to 1700 ns. During this time, the remaining bytes of the packet are processed, and the last four bytes are transmitted. Following the successful processing and discard of the complete packet, the `done` signal is asserted. This signal indicates the formal end of the simulation and confirms that the FIFO has emptied correctly, completing the system's handling of the packet.

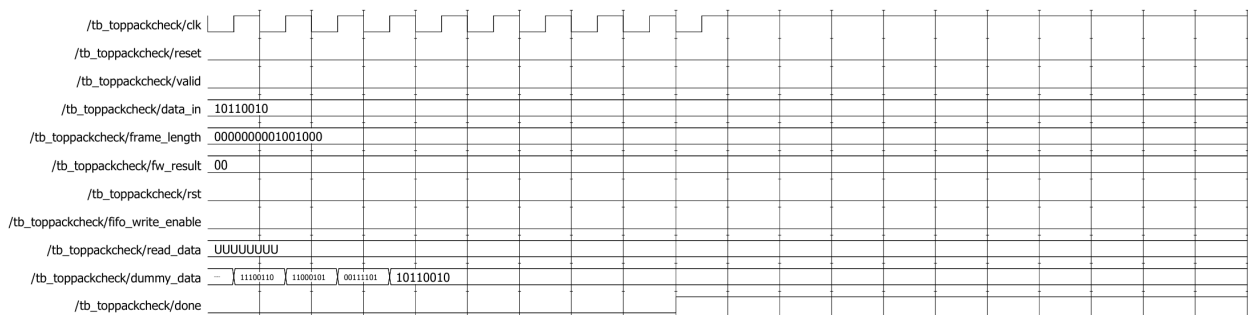


Figure 47: End of the simulation: Final bytes processed and simulation completion signal asserted, with no firewall rules (1500-1700 ns).

## Firewall allows packet

In this part of the simulation, a firewall rule is introduced to the `Firewall` system that explicitly allows the Ethernet packet used in the previous test. As noted earlier, both simulations use the same test packet, which contains a valid frame check sequence (FCS). Figure 48 shows the rule configuration used in this test. The rule permits traffic based on both the source and destination IP addresses present in the packet. Furthermore, it only allows UDP packets targeting specific ports, adding an additional layer of filtering.

```

-- Allow specific UDP packet
(x"COA8002C", x"COA80004", x"11", x"0400", x"0400",
 x"FFFFFFFF", x"FFFFFFFF", x"FF", x"FFFF", x"FFFF",
 '1'), -- ALLOW

```

Figure 48: Rule configured in the `Firewall` system to permit the test packet.

The beginning of the simulation, shown in Figure 49, remains essentially unchanged compared to the first scenario without firewall rules. This is expected, as the presence or absence of a rule does not affect how the packet is initially received and buffered. The data is still sequentially written into the `FIFO` module after the system reset. Since the packet format and timing are identical to the first test, no additional discussion is necessary for this phase.

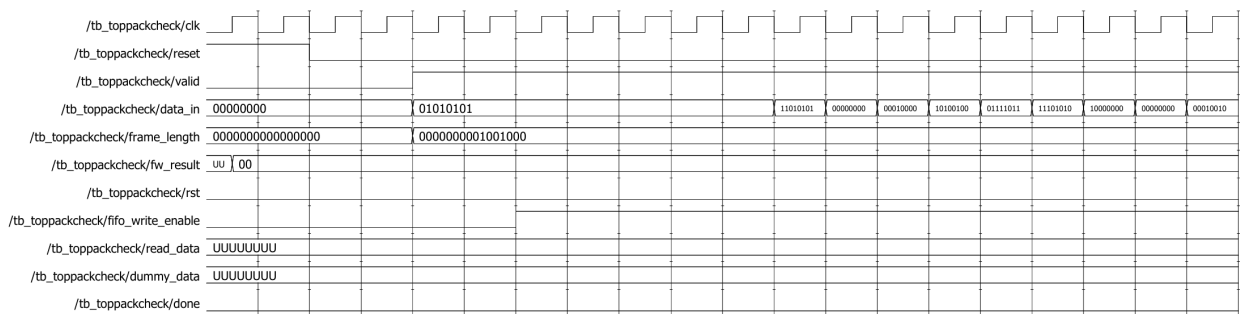


Figure 49: Start of the simulation with firewall rules configured (0-200 ns).

The main difference becomes apparent during the mid-simulation interval, shown in Figure 50, covering 700 ns to 900 ns. Here, the output of the `Check Rules` module is evaluated. The module correctly identifies the packet as matching the active rule, producing a positive result. As a result, the packet is forwarded through the `read_data` output line. This behavior confirms that the `Firewall` system correctly applies its rule set and permits valid traffic as configured.

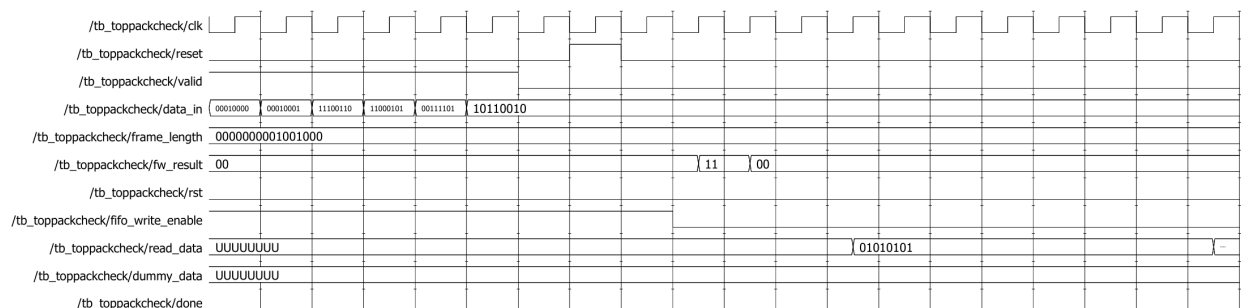


Figure 50: Mid-simulation: Packet allowed and forwarded due to matching firewall rules (700-900 ns).

The final part of the simulation is illustrated in Figure 51, which spans from 1500 ns to 1700 ns. Similar to the previous simulation, the last four bytes of the packet are transmitted, and the `done` flag is asserted to indicate the end of processing. The only notable difference is the output destination: in this case, the packet is successfully forwarded to the appropriate data path rather than being discarded. This confirms the correct and complete operation of the firewall when a rule is present that matches the incoming packet.

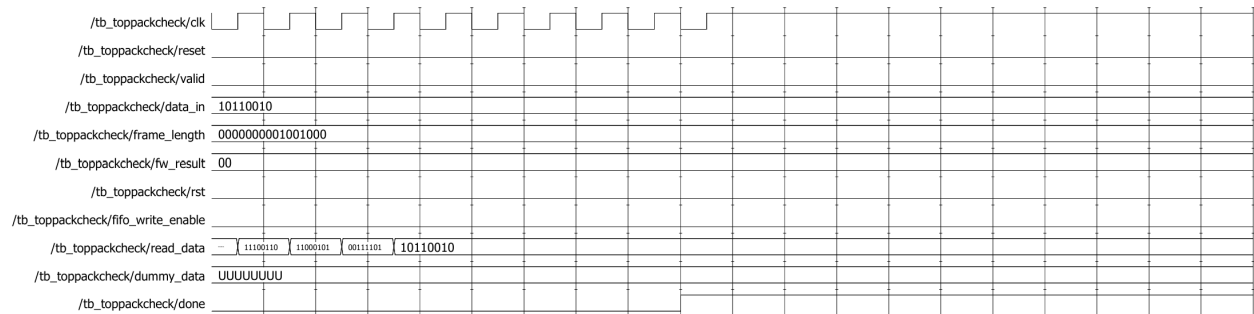


Figure 51: End of the simulation: Packet forwarded and simulation completed with firewall rules (1500-1700 ns).

## Packet Blocked Due to Invalid FCS

Figure 52 presents the Ethernet packet used to validate the behavior of the `Firewall` system when a frame with an invalid Frame Check Sequence (FCS) is received. This packet was intentionally crafted with a faulty FCS to trigger a rejection by the FCS module, which performs data integrity checks. As shown in the figure, all fields of the Ethernet packet are clearly annotated to aid visual understanding of its structure. In particular, the only modification made to the otherwise valid packet is a change in the first byte, altered from 0 to 1, which is sufficient to cause FCS verification to fail.

```
-- Frame to test (72 bytes = 8 preamble + 60 data + 4 CRC)
type frame_array is array (0 to 71) of std_logic_vector(7 downto 0);
constant test_frame : frame_array := (
    x"55", x"55", x"55", x"55", x"55", x"55", x"55", x"D5", --preamble
    x"01", x"10", x"A4", x"7B", x"EA", x"80", -- Dest MAC
    x"00", x"12", x"34", x"56", x"78", x"90", -- Src MAC
    x"08", x"00", -- EtherType: IPv4
    x"45", x"00", x"00", x"2E", x"B3", x"FE", x"00", x"00", x"80", -- IP header start
    x"11", -- Byte 23: Protocol (UDP = 17 = x"11")
    x"05", x"40", -- Checksum etc.
    x"C0", x"A8", x"00", x"2C", -- src ip:
    x"C0", x"A8", x"00", x"04", -- dst ip:
    x"04", x"00", -- src port:
    x"04", x"00", -- dst port:
    x"00", x"1A", x"2D", x"E8", -- Payload...
    x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
    x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
    x"10", x"11", -- More payload
    x"E6", x"C5", x"3D", x"B2" -- FCS
);
```

Figure 52: Test packet used in the `Firewall` system simulation, containing an invalid FCS.

The beginning of the simulation is shown in Figure 53 and appears largely identical to the previous two test scenarios. This is expected, as the packet structure and timing remain



consistent. However, closer inspection reveals a difference in the ninth byte, which contains the modified value responsible for the FCS error. Other than this change, the behavior of the control signals—including the `valid` and `fifo_write_enable` flags—remains the same as in the earlier simulations, and the packet continues to be stored in the `FIFO`.

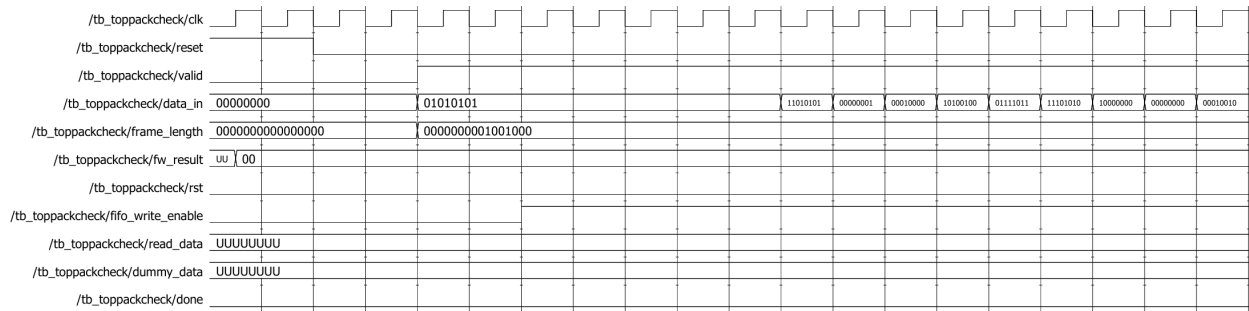


Figure 53: Simulation start with firewall rules configured, but invalid FCS (0200 ns).

The impact of the invalid FCS becomes evident in the mid-simulation stage, illustrated in Figure 54, which covers the 700 ns to 900 ns window. The `Check Rules` module detects the data integrity failure and issues a negative result, indicating that the packet should not be forwarded. Consequently, the system redirects the packet to the `dummy_data` output, effectively discarding it. This confirms that the firewall correctly handles packets with corrupted or manipulated FCS values. Other control and status signals maintain the same behavior as in the previous tests.

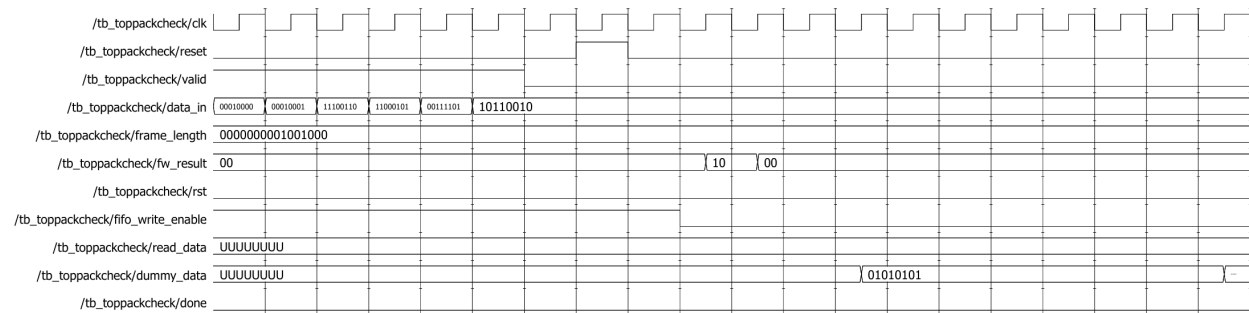


Figure 54: Mid-simulation: Packet evaluation with firewall rules active, but invalid FCS detected (700-900 ns).

The final stage of the simulation is shown in Figure 55, covering the interval from 1500 ns to 1700 ns. As in the previous scenarios, the last four bytes of the packet are processed, and the `done` flag is asserted to mark the end of the simulation. Signal behavior during this phase mirrors the earlier tests, reaffirming that the overall processing flow remains stable even when a packet is rejected due to FCS errors.

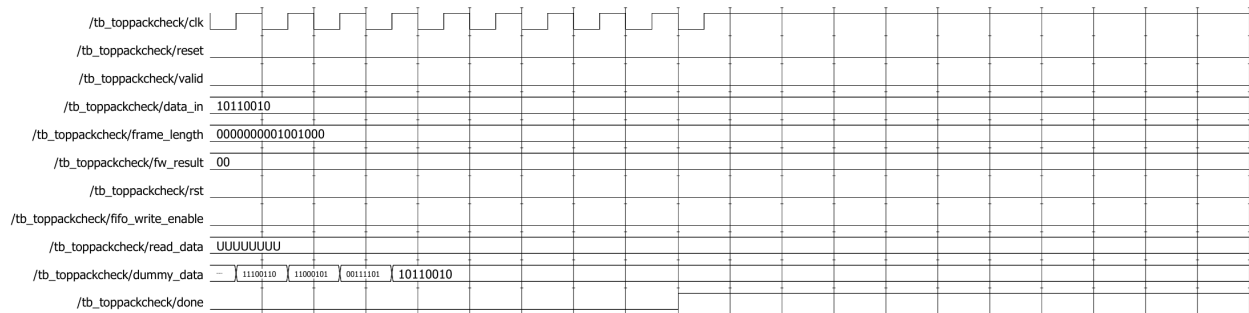


Figure 55: End of the simulation: Final bytes processed and simulation completion signal asserted with firewall rules active and invalid FCS (1500-1700 ns).

## Conclusion of Simulation Tests

The simulation results confirm that the **Firewall system** performs as intended under a variety of conditions. In the first test, with no firewall rules defined, packets were correctly blocked by default. The second test demonstrated that a valid forwarding rule enables the packet to pass through the system as expected. Finally, the third simulation showed that even with a valid rule in place, a packet with a corrupted FCS is effectively detected and discarded by the FCS module.

Across all scenarios, the **Firewall system** showed consistent and deterministic timing behavior due to the identical packet structure, confirming that its performance is stable and predictable. The correct interaction between subsystems—such as rule checking, FCS verification, and FIFO buffering—further validates the reliability of the design. These simulations collectively verify that the system enforces rule-based filtering and data integrity checks, meeting the functional requirements for Ethernet packet inspection and forwarding.

## 7 Conclusions

For this project, we were asked to create a stateless firewall using FPGA technology and simulating it through Quartus and ModelSim. Thanks to our logic design and implementation, we were able to successfully complete this task. We chose the approach that, in our view, offered the best balance between functionality and feasibility, even though we acknowledge it introduces some delays.

As discussed in detail earlier, our implementation includes an FCS check to verify the integrity of incoming packets. However, due to logical constraints, the outcome of this check can only be determined in the clock cycle following the complete reception of the packet. For this reason, we chose to allow the packet to continue through the system and to perform the FCS validation not immediately after the CRC block, but later in the **Check Rules** block. This decision was driven by two main considerations.

On one hand, in a realistic scenario, it is reasonable to assume that the majority of packets will be correct. On the other, since the firewall's purpose is to evaluate whether a packet is authorized to proceed, we would ultimately have had to halt the packet's progress in that block either way. Therefore, it seemed acceptable to adopt an approach where the packet is temporarily accepted and stored in the FIFO, and only then is it either validated or discarded based on both the firewall rule check and the FCS result.

We are aware that if the number of incorrect packets were to exceed the correct ones, this method would become inefficient and waste resources. However, that scenario did not reflect the assumptions under which we were operating.

In the early stages of design, we considered sending data to the FIFO in parallel while other blocks were still processing the packet, in an attempt to save clock cycles. However, we later realized that since the FCS validation is only possible after the entire packet has been received, parallel transmission would not have actually reduced processing time. In any case, we would still have needed to wait for the full packet to perform validation. For this reason, we decided to adopt the more straightforward approach we described above, which offered similar timing performance with lower complexity.

Later on, we refined this strategy further: the packet is always sent to memory, where it is held while awaiting confirmation for forwarding. In parallel, the firewall checks the relevant fields of the packet according to the defined rules. This avoids introducing additional delays. Although these processes run concurrently, we know the firewall only examines the first 37 bytes (after the preamble and SFD), while the FCS block requires the entire packet. Even in the worst case, the last 4 bytes required for the FCS check, but not by the firewall, provide a sufficient safety margin to ensure that the FCS result is never needed before the firewall decision is complete. This prevents any timing conflicts between the two processes.

Overall, we believe the project was a success, as it functions correctly and performs all the intended checks. While the logic itself did not present major issues, we did encounter challenges during code implementation and especially in integrating the various parts. After many hours of work and debugging, however, everything came together.

As for the design choices, we are satisfied with the results, although we recognize opportunities for improvement, especially in terms of processing time and computational efficiency. One area for optimization is rule management. In our current implementation, rules are hardcoded, which works but is not scalable or efficient. In more complex systems, where many rules are needed, this method would be limiting. A potential improvement could involve implementing hashing or techniques that allow blocking entire ranges of IP or MAC addresses, rather than specifying them one by one.

Another possible alternative could have been to send the packet and the FCS result directly to the FIFO in parallel with the rest of the operations. However, in our evaluation, the additional code complexity introduced by the resulting asynchronicity was not justified by the marginal improvements in performance or memory usage.

That said, considering the project goals and the results obtained in simulation we are confident that our implementation met the specifications and successfully delivered a functional stateless firewall.

## A Code

### A.1 GMII to MAC: mac\_rx.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity MAC_RX is
6      Port (
7          -- GMII Interface
8          GMII_RX_CLK      : in std_logic;
9          GMII_RXD          : in std_logic_vector(7 downto 0);
10         GMII_RX_DV        : in std_logic;
11         GMII_RX_RESET     : in std_logic;
12         TOTAL_LENGTH      : in std_logic_vector(15 downto 0); -- Total
                                packet length input at runtime
13
14         -- MAC RX Interface
15         MAC_RX_CLK        : out std_logic;
16         MAC_RXD            : out std_logic_vector(7 downto 0);
17         MAC_RX_VALID      : out std_logic;
18         MAC_RX_FIRST      : out std_logic;
19         MAC_RX_LAST       : out std_logic;
20         MAC_RX_RESET      : out std_logic
21     );
22 end MAC_RX;
23
24 architecture Behavioral of MAC_RX is
25
26     signal byte_count      : integer range 0 to 65535 := 0;
27     signal preamble_done   : std_logic := '0';
28
29 begin
30
31     -- Forward GMII clock and reset to MAC
32     MAC_RX_CLK    <= GMII_RX_CLK;
33     MAC_RX_RESET <= GMII_RX_RESET;
34
35     process(GMII_RX_CLK)
36         variable total_len : integer := 0;
37     begin
38         if rising_edge(GMII_RX_CLK) then
39
40             total_len := to_integer(unsigned(TOTAL_LENGTH));
41
42             if GMII_RX_DV = '1' then
43
44                 -- Handle preamble end detection (first 8 bytes)
45                 if byte_count = 6 then

```

```

47         preamble_done <= '1';
48     end if;
49
50     -- Set MAC_RX_FIRST at first byte after preamble
51     if byte_count = 8 then
52         MAC_RX_FIRST <= '1';
53     else
54         MAC_RX_FIRST <= '0';
55     end if;
56
57     -- Set MAC_RX_VALID for bytes after preamble until last
byte
58     if byte_count > total_len - 5 then
59         MAC_RX_VALID <= '1';
60     end if;
61
62     -- Increment byte counter or reset after last byte
63     if byte_count = total_len - 1 then
64         byte_count <= 0;
65         preamble_done <= '0';
66     else
67         byte_count <= byte_count + 1;
68     end if;
69     MAC_RXD <= GMII_RXD;
70
71     -- Set MAC_RX_LAST only for the final byte
72     if byte_count = total_len - 1 then
73         MAC_RX_LAST <= '1';
74     else
75         MAC_RX_LAST <= '0';
76     end if;
77 else
78     -- No valid data; reset signals
79     MAC_RX_VALID <= '0';
80     MAC_RX_FIRST <= '0';
81     MAC_RX_LAST <= '0';
82     byte_count <= 0;
83     preamble_done <= '0';
84 end if;
85
86     end if;
87 end process;
88
89 end Behavioral;

```

## A.2 GMII to MAC testbench: mac\_rx\_tb.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4

```

```

5  entity tb_MAC_RX is
6  end tb_MAC_RX;
7
8  architecture sim of tb_MAC_RX is
9
10     -- Component declaration
11     component MAC_RX
12         Port (
13             GMII_RX_CLK      : in std_logic;
14             GMII_RXD         : in std_logic_vector(7 downto 0);
15             GMII_RX_DV       : in std_logic;
16             GMII_RX_RESET    : in std_logic;
17             TOTAL_LENGTH     : in std_logic_vector(15 downto 0);
18
19             MAC_RX_CLK       : out std_logic;
20             MAC_RXD          : out std_logic_vector(7 downto 0);
21             MAC_RX_VALID     : out std_logic;
22             MAC_RX_FIRST     : out std_logic;
23             MAC_RX_LAST      : out std_logic;
24             MAC_RX_RESET     : out std_logic
25         );
26     end component;
27
28     -- Signals
29     signal clk                : std_logic := '0';
30     signal rst                : std_logic := '0';
31     signal rx_dv              : std_logic := '0';
32     signal rx_d               : std_logic_vector(7 downto 0) := (others => '0');
33     signal total_len          : std_logic_vector(15 downto 0) := (others => '0');
34
35     signal mac_clk            : std_logic;
36     signal mac_d              : std_logic_vector(7 downto 0);
37     signal mac_valid          : std_logic;
38     signal mac_first          : std_logic;
39     signal mac_last           : std_logic;
40     signal mac_reset          : std_logic;
41
42     -- Clock generation (125 MHz typical for GMII)
43     constant clk_period : time := 8 ns;
44
45 begin
46
47     -- Instantiate DUT
48     uut: MAC_RX
49         Port map (
50             GMII_RX_CLK => clk,
51             GMII_RXD    => rx_d,
52             GMII_RX_DV  => rx_dv,
53             GMII_RX_RESET => rst,
54             TOTAL_LENGTH => total_len,
55

```

```
56         MAC_RX_CLK      => mac_clk,
57         MAC_RXD          => mac_d,
58         MAC_RX_VALID     => mac_valid,
59         MAC_RX_FIRST     => mac_first,
60         MAC_RX_LAST      => mac_last,
61         MAC_RX_RESET     => mac_reset
62     );
63
64     -- Clock process
65     clk_process : process
66     begin
67         while true loop
68             clk <= '0';
69             wait for clk_period / 2;
70             clk <= '1';
71             wait for clk_period / 2;
72         end loop;
73     end process;
74
75     -- Stimulus process
76     stimulus : process
77         procedure send_packet(packet_len : integer) is
78         begin
79             total_len <= std_logic_vector(to_unsigned(packet_len, 16));
80             rx_dv <= '1';
81             for i in 0 to packet_len - 1 loop
82                 rx_d <= std_logic_vector(to_unsigned(i mod 256, 8));
83                 wait for clk_period;
84             end loop;
85             rx_dv <= '0';
86             wait for clk_period * 5; -- Idle period between packets
87         end procedure;
88     begin
89         -- Reset
90         rst <= '1';
91         wait for clk_period * 2;
92         rst <= '0';
93
94         -- Send 3 packets: 72, 64, and 80 bytes long
95         wait for clk_period * 5;
96         send_packet(72);
97         send_packet(64);
98         send_packet(80);
99
100        -- Finish
101        wait for clk_period * 20;
102        assert false report "Simulation finished" severity failure;
103    end process;
104
105 end sim;
```



### A.3 FCS: fcs.vhd

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5  entity fcs is
6      port (
7          clk          : in  std_logic;
8          reset        : in  std_logic;
9          valid        : in  std_logic;
10         start_of_frame : in  std_logic;
11         end_of_frame   : in  std_logic;
12         data_in        : in  std_logic_vector(7 downto 0);
13         last_of_frame  : out std_logic;
14         first_of_frame : out std_logic;
15         data_out       : out std_logic_vector(7 downto 0);
16         fcs_error      : out std_logic_vector(1 downto 0)
17     );
18 end fcs;
19
20 architecture behavioral of fcs is
21
22     signal R          : std_logic_vector(31 downto 0);
23     signal data       : std_logic_vector(7 downto 0);
24     signal fcs_done   : std_logic;
25     signal one_more   : std_logic;
26     signal actived    : std_logic;
27
28     signal byte_count : unsigned(2 downto 0);
29
30
31
32 begin
33
34     --Complementation of first 4 and last 4 bytes
35     process (byte_count, start_of_frame, valid, data_in)
36     begin
37         if (byte_count < 4) or (start_of_frame = '1') or (valid = '1') then
38             data <= not data_in;
39         else
40             data <= data_in;
41         end if;
42     end process;
43
44
45     --Main CRC computation
46     process (clk, reset)
47     begin
48
49         if reset = '1' then
50             R          <= (others => '0'); -- Initial value: 0xFFFFFFFF
```

```

51         fcs_error    <= "01";
52         fcs_done     <= '0';
53         actived      <= '0';
54
55         elsif rising_edge(clk) then
56             last_of_frame <= end_of_frame;
57             first_of_frame <= start_of_frame;
58             data_out <= data_in;
59
60             if start_of_frame = '1' then
61                 R        <= (others => '0'); -- Reset CRC on new frame
62                 fcs_done <= '0';
63                 one_more <= '0';
64                 fcs_error <= "01";
65                 actived  <= '1';
66             end if;
67
68
69             if (start_of_frame = '1') or (valid = '1') then
70                 byte_count <= (others => '0');
71             elsif byte_count < 4 then
72                 byte_count <= byte_count + 1;
73             end if;
74
75             if actived = '1' then
76                 -- CRC matrix calculation
77                 R(0) <= data(0) xor R(24) xor R(30);
78                 R(1) <= data(1) xor R(24) xor R(25) xor R(30) xor R(31);
79                 R(2) <= data(2) xor R(24) xor R(25) xor R(26) xor R(30) xor R(31);
80                 R(3) <= data(3) xor R(25) xor R(26) xor R(27) xor R(31);
81                 R(4) <= data(4) xor R(24) xor R(26) xor R(27) xor R(28) xor R(30);
82                 R(5) <= data(5) xor R(24) xor R(25) xor R(27) xor R(28) xor R(29) xor R(30) xor
83                     R(31);
84                 R(6) <= data(6) xor R(25) xor R(26) xor R(28) xor R(29) xor R(30) xor R(31);
85                 R(7) <= data(7) xor R(24) xor R(26) xor R(27) xor R(29) xor R(31);
86                 R(8) <= R(0) xor R(24) xor R(25) xor R(27) xor R(28);
87                 R(9) <= R(1) xor R(25) xor R(26) xor R(28) xor R(29);
88                 R(10) <= R(2) xor R(24) xor R(26) xor R(27) xor R(29);
89                 R(11) <= R(3) xor R(24) xor R(25) xor R(27) xor R(28);
90                 R(12) <= R(4) xor R(24) xor R(25) xor R(26) xor R(28) xor R(29) xor R(30)
91                     ;
92                 R(13) <= R(5) xor R(25) xor R(26) xor R(27) xor R(29) xor R(30) xor R(31)
93                     ;
94                 R(14) <= R(6) xor R(26) xor R(27) xor R(28) xor R(30) xor R(31);
95                 R(15) <= R(7) xor R(27) xor R(28) xor R(29) xor R(31);
96                 R(16) <= R(8) xor R(24) xor R(28) xor R(29);
97                 R(17) <= R(9) xor R(25) xor R(29) xor R(30);
98                 R(18) <= R(10) xor R(26) xor R(30) xor R(31);
99                 R(19) <= R(11) xor R(27) xor R(31);
100                R(20) <= R(12) xor R(28);
101                R(21) <= R(13) xor R(29);
102                R(22) <= R(14) xor R(24);
103                R(23) <= R(15) xor R(24) xor R(25) xor R(30);

```

```

101  R(24) <=R(16) xor R(25) xor R(26) xor R(31);
102  R(25) <=R(17) xor R(26) xor R(27);
103  R(26) <=R(18) xor R(24) xor R(27) xor R(28) xor R(30);
104  R(27) <=R(19) xor R(25) xor R(28) xor R(29) xor R(31);
105  R(28) <=R(20) xor R(26) xor R(29) xor R(30);
106  R(29) <=R(21) xor R(27) xor R(30) xor R(31);
107  R(30) <=R(22) xor R(28) xor R(31);
108  R(31) <=R(23) xor R(29);
109      end if;
110
111  if end_of_frame = '1' then
112      one_more <= '1';
113  end if;
114  if one_more = '1' then
115      fcs_done <= '1';
116  end if;
117
118  end if;
119
120      -- Check CRC result (with final XOR 0xFFFFFFFF)
121      if (R = x"00000000") and (fcs_done = '1') then
122          fcs_error <= "10"; -- No error
123      fcs_done <= '0';
124      one_more <= '0';
125      elsif (R /= x"00000000") and (fcs_done = '1') then
126          fcs_error <= "11";
127      fcs_done <= '0';
128      one_more <= '0';
129      else
130          fcs_error <= "01"; -- Error
131      end if;
132
133
134  end process;
135
136 end behavioral;

```

## A.4 FCS testbench: tb\_fcs.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4
5
6  entity tb_fcs is
7  end tb_fcs;
8
9  architecture sim of tb_fcs is
10
11      -- Component declaration
12      component fcs

```

```

13     port (
14         clk          : in  std_logic;
15         reset        : in  std_logic;
16         valid        : in  std_logic;
17         start_of_frame : in  std_logic;
18         end_of_frame  : in  std_logic;
19         data_in       : in  std_logic_vector(7 downto 0);
20         last_of_frame  : out std_logic;
21         first_of_frame : out std_logic;
22         data_out       : out std_logic_vector(7 downto 0);
23         fcs_error      : out std_logic_vector(1 downto 0)
24     );
25 end component;
26
27 signal clk          : std_logic := '0';
28 signal reset        : std_logic := '1';
29 signal valid        : std_logic := '0';
30 signal start_of_frame : std_logic := '0';
31 signal end_of_frame  : std_logic := '0';
32 signal data_in       : std_logic_vector(7 downto 0) := (others => '0');
33 signal fcs_error      : std_logic_vector(1 downto 0) ;
34
35 -- Clock generation
36 constant clk_period : time := 10 ns;
37 signal done : boolean := false;
38
39 -- Frame to test (64 bytes = 60 dati + 4 CRC)
40 type frame_array is array (0 to 63) of std_logic_vector(7 downto 0);
41 constant test_frame : frame_array := (
42     x"00", x"10", x"A4", x"7B", x"EA", x"80", x"00", x"12",
43     x"34", x"56", x"78", x"90", x"08", x"00", x"45", x"00",
44     x"00", x"2E", x"B3", x"FE", x"00", x"00", x"80", x"11",
45     x"05", x"40", x"C0", x"A8", x"00", x"2C", x"C0", x"A8",
46     x"00", x"04", x"04", x"00", x"04", x"00", x"00", x"1A",
47     x"2D", x"E8", x"00", x"01", x"02", x"03", x"04", x"05",
48     x"06", x"07", x"08", x"09", x"0A", x"0B", x"0C", x"0D",
49     x"0E", x"0F", x"10", x"11", -- dati
50     x"E6", x"C5", x"35", x"11"  -- FCS 3D B2
51 );
52
53
54 begin
55
56     -- Instantiate the fcs module
57     uut: fcs
58     port map (
59         clk          => clk,
60         reset        => reset,
61         valid        => valid,
62         start_of_frame => start_of_frame,
63         end_of_frame  => end_of_frame,
64         data_in       => data_in,
65         fcs_error      => fcs_error

```

```
66     );
67
68     -- Clock generation process
69     clk_process : process
70     begin
71         while not done loop
72             clk <= '0';
73             wait for clk_period / 2;
74             clk <= '1';
75             wait for clk_period / 2;
76         end loop;
77         wait;
78     end process;
79
80     -- Stimulus process
81     stimulus : process
82     begin
83         wait for 20 ns;
84         reset <= '0';
85         wait for 20 ns;
86
87         start_of_frame <= '1';
88
89         for i in 0 to 63 loop
90             data_in <= test_frame(i);
91             wait for clk_period;
92             if i = 0 then
93                 wait for clk_period;
94                 start_of_frame <= '0';
95             end if;
96             if i >= 59 then
97                 valid <= '1';
98                 if i = 63 then
99                     end_of_frame <= '1';
100                 end if;
101             end if;
102         end loop;
103         wait for clk_period;
104         end_of_frame <= '0';
105         reset <= '1';
106         wait for clk_period;
107
108         reset <= '0';
109
110         -- Let the simulation run a bit
111         wait for 50 ns;
112         done <= true;
113         wait;
114     end process;
115
116 end sim;
```

## A.5 MAC-RX Control: MAC\_RX\_CONTROL.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_ARITH.ALL;
4  use IEEE.STD_LOGIC_UNSIGNED.ALL;
5
6
7  entity MAC_RX_CONTROL is
8      Port (
9          MAC_RX_CLK      : in  STD_LOGIC;
10         MAC_RXD          : in  STD_LOGIC_VECTOR(7 downto 0);
11         MAC_RX_VALID     : in  STD_LOGIC;  -- 1 when the first CRC byte is
            sent
12         MAC_RX_LAST      : in  STD_LOGIC;
13         MAC_RX_ERR        : in  STD_LOGIC_VECTOR(1 downto 0);
14         reset            : in  STD_LOGIC;
15         LINE_NUMBER       : out STD_LOGIC_VECTOR(15 downto 0);
16         DATA             : out STD_LOGIC_VECTOR(7 downto 0);
17         FW_OUT            : out STD_LOGIC_VECTOR(1 downto 0);
18         START_OF_FRAME    : out STD_LOGIC;
19         END_OF_FRAME      : out STD_LOGIC
20     );
21 end MAC_RX_CONTROL;
22
23 architecture Behavioral of MAC_RX_CONTROL is
24     signal line_counter : STD_LOGIC_VECTOR(15 downto 0) := (others => '0')
25     ;
26     signal in_packet    : STD_LOGIC := '0';  -- internal state
27     signal sop          : STD_LOGIC;
28 begin
29     process (MAC_RX_CLK)
30     begin
31         if rising_edge(MAC_RX_CLK) then
32             -- Default outputs
33             START_OF_FRAME <= MAC_RX_VALID;
34             END_OF_FRAME   <= MAC_RX_LAST;
35             FW_OUT         <= MAC_RX_ERR;
36             DATA          <= MAC_RXD;
37             if in_packet = '0' or reset = '1' then
38                 line_counter <= (others => '0');
39             end if;
40
41             if sop = '1' then
42                 LINE_NUMBER <= line_counter + 1;
43             end if;
44
45             if MAC_RX_VALID = '1' then
46                 sop <= '1';
47                 line_counter <= (others => '0');
48             end if;
49         end if;
50     end process;
51 end Behavioral;

```

```

49         in_packet    <= '1';
50     elsif in_packet = '1' then
51         line_counter <= line_counter + 1;
52
53         if MAC_RX_LAST = '1' then
54             in_packet    <= '0';
55         sop <= '0';
56         end if;
57     end if;
58 if MAC_RX_VALID = '1' then
59     LINE_NUMBER <= line_counter;
60 end if;
61 end if;
62 end process;
63
64 end Behavioral;

```

## A.6 MAC-RX Control testbench: MAC\_RX\_CONTROL\_tb.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity MAC_RX_CONTROL_tb is
6  end MAC_RX_CONTROL_tb;
7
8  architecture tb of MAC_RX_CONTROL_tb is
9      -- Inputs to DUT
10     signal mac_rx_clk      : STD_LOGIC := '0';
11     signal mac_rxd         : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
12     signal mac_rx_valid    : STD_LOGIC := '0';
13     signal mac_rx_last     : STD_LOGIC := '0';
14     signal mac_rx_err      : STD_LOGIC := '0';
15     signal reset           : STD_LOGIC := '0';
16
17     -- Outputs from DUT
18     signal line_number     : STD_LOGIC_VECTOR(15 downto 0);
19     signal data_fw         : STD_LOGIC_VECTOR(7 downto 0);
20     signal fw_out          : STD_LOGIC;
21     signal start_of_frame : STD_LOGIC;
22     signal end_of_frame   : STD_LOGIC;
23
24     component MAC_RX_CONTROL is
25         Port (
26             MAC_RX_CLK      : in  STD_LOGIC;
27             MAC_RXD         : in  STD_LOGIC_VECTOR(7 downto 0);
28             MAC_RX_VALID    : in  STD_LOGIC;
29             MAC_RX_LAST     : in  STD_LOGIC;
30             MAC_RX_ERR      : in  STD_LOGIC;
31             reset           : in  STD_LOGIC;
32             LINE_NUMBER     : out STD_LOGIC_VECTOR(15 downto 0);

```

```

33         DATA          : out STD_LOGIC_VECTOR(7 downto 0);
34         FW_OUT          : out STD_LOGIC;
35         START_OF_FRAME : out STD_LOGIC;
36         END_OF_FRAME   : out STD_LOGIC
37     );
38 end component;
39
40 constant CLK_PERIOD : time := 10 ns;
41
42 -- Example packet
43 type byte_array is array (natural range <>) of STD_LOGIC_VECTOR(7
downto 0);
44 constant test_packet : byte_array := (
45     x"00", x"10", x"A4", x"7B", x"EA", x"80", -- Dest MAC
46     x"00", x"12", x"34", x"56", x"78", x"90", -- Src MAC
47     x"08", x"00", -- EtherType: IPv4
48     x"45", x"00", x"00", x"2E", x"B3", x"FE", x"00", x"00", x"80", --
IP header start
49     x"11", -- Byte 23: Protocol (
UDP = 17 = x"11")
50     x"05", x"40", -- Checksum etc.
51     x"C0", x"A8", x"00", x"2C", -- Source IP:
192.168.0.44
52     x"C0", x"A8", x"00", x"04", -- Dest IP:
192.168.0.4
53     x"04", x"00", -- Source Port: 1024
54     x"04", x"00", -- Dest Port: 1024
55     x"00", x"1A", x"2D", x"E8", -- Payload...
56     x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
57     x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
58     x"10", x"11", -- More payload
59     x"E6", x"C5", x"3D", x"B2" -- FCS
60 );
61
62 begin
63     -- DUT instantiation
64     uut: MAC_RX_CONTROL
65     port map (
66         MAC_RX_CLK   => mac_rx_clk,
67         MAC_RXD      => mac_rxd,
68         MAC_RX_VALID => mac_rx_valid,
69         MAC_RX_LAST  => mac_rx_last,
70         MAC_RX_ERR   => mac_rx_err,
71         LINE_NUMBER  => line_number,
72         DATA        => data_fw,
73         FW_OUT       => fw_out,
74         reset        => reset,
75         START_OF_FRAME => start_of_frame,
76         END_OF_FRAME  => end_of_frame
77     );
78
79     -- Clock generation
80     clk_process : process

```



```

81     begin
82         while true loop
83             mac_rx_clk <= '0';
84             wait for CLK_PERIOD / 2;
85             mac_rx_clk <= '1';
86             wait for CLK_PERIOD / 2;
87         end loop;
88     end process;
89
90     -- Stimulus
91     stim_proc : process
92     begin
93         wait for 20 ns;
94         mac_rx_valid <= '1';
95         mac_rx_err <= '0';
96     reset <= '0';
97     for i in 0 to test_packet'length - 1 loop
98         mac_rxd <= test_packet(i);
99         if i = 0 then
100             start_of_frame <= '1';
101         else
102             start_of_frame <= '0';
103         end if;
104
105         if i = test_packet'length - 1 then
106             mac_rx_last <= '1';
107             end_of_frame <= '1';
108         else
109             mac_rx_last <= '0';
110             end_of_frame <= '0';
111         end if;
112
113         wait for CLK_PERIOD;
114     end loop;
115
116     mac_rx_valid <= '0';
117     mac_rx_last <= '0';
118     end_of_frame <= '0';
119     wait;
120 end stim_proc;
121
122 end tb;

```

## A.7 Packet Analyzer: PacketAnalyzer.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.STD_LOGIC_UNSIGNED.ALL;
4
5  entity PacketAnalyzer is
6      Port (

```

```

7      mac_rx_clk      : in  STD_LOGIC;
8      rst             : in  STD_LOGIC;
9      line_number     : in  STD_LOGIC_VECTOR(15 downto 0);
10     data_fw         : in  STD_LOGIC_VECTOR(7 downto 0);
11     start_of_frame  : in  STD_LOGIC;
12     end_of_frame    : in  STD_LOGIC;
13     fw_out          : in  STD_LOGIC_VECTOR(1 downto 0);
14
15     -- Output verso CheckRules
16     source_ip       : out STD_LOGIC_VECTOR(31 downto 0);
17     dest_ip        : out STD_LOGIC_VECTOR(31 downto 0);
18     source_port     : out STD_LOGIC_VECTOR(15 downto 0);
19     dest_port      : out STD_LOGIC_VECTOR(15 downto 0);
20     protocol        : out STD_LOGIC_VECTOR(7 downto 0);
21     data_ready      : out STD_LOGIC;
22     fw_out_check    : out STD_LOGIC_VECTOR(1 downto 0);
23
24     -- Output verso FIFO
25     fifo_data       : out STD_LOGIC_VECTOR(7 downto 0);
26     fifo_sof        : out STD_LOGIC;
27     fifo_eof        : out STD_LOGIC
28 );
29 end PacketAnalyzer;
30
31 architecture Behavioral of PacketAnalyzer is
32     type state_type is (IDLE, CAPTURING, READY);
33     signal current_state : state_type := IDLE;
34
35     signal src_ip_reg      : STD_LOGIC_VECTOR(31 downto 0) := (others => '0'
36 );
37     signal dst_ip_reg      : STD_LOGIC_VECTOR(31 downto 0) := (others => '0'
38 );
39     signal src_port_reg    : STD_LOGIC_VECTOR(15 downto 0) := (others => '0'
40 );
41     signal dst_port_reg    : STD_LOGIC_VECTOR(15 downto 0) := (others => '0'
42 );
43     signal protocol_reg    : STD_LOGIC_VECTOR(7 downto 0) := (others => '0'
44 );
45     signal ready_reg       : STD_LOGIC := '0';
46 begin
47
48     process(mac_rx_clk)
49     begin
50         if rising_edge(mac_rx_clk) then
51             if rst = '1' then
52                 current_state <= IDLE;
53                 src_ip_reg    <= (others => '0');
54                 dst_ip_reg    <= (others => '0');
55                 src_port_reg  <= (others => '0');
56                 dst_port_reg  <= (others => '0');
57                 protocol_reg  <= (others => '0');
58                 ready_reg     <= '0';
59                 fifo_data     <= (others => '0');

```

```

55         fifo_sof      <= '0';
56         fifo_eof      <= '0';
57         fw_out_check  <= (others => '0');
58     else
59         -- Forward dati al FIFO
60         fifo_data      <= data_fw;
61         fifo_sof      <= start_of_frame;
62         fifo_eof      <= end_of_frame;
63
64         -- Inoltra fw_out a CheckRules
65         fw_out_check  <= fw_out;
66
67         -- FSM per catturare i campi del pacchetto
68         case current_state is
69             when IDLE =>
70                 ready_reg <= '0';
71                 if start_of_frame = '1' then
72                     current_state <= CAPTURING;
73                 end if;
74
75             when CAPTURING =>
76                 case line_number is
77                     when X"0017" => protocol_reg <= data_fw;
78                     when X"001A" => src_ip_reg(31 downto 24) <=
data_fw;
79                     when X"001B" => src_ip_reg(23 downto 16) <=
data_fw;
80                     when X"001C" => src_ip_reg(15 downto 8) <=
data_fw;
81                     when X"001D" => src_ip_reg(7 downto 0) <=
data_fw;
82                     when X"001E" => dst_ip_reg(31 downto 24) <=
data_fw;
83                     when X"001F" => dst_ip_reg(23 downto 16) <=
data_fw;
84                     when X"0020" => dst_ip_reg(15 downto 8) <=
data_fw;
85                     when X"0021" => dst_ip_reg(7 downto 0) <=
data_fw;
86                     when X"0022" => src_port_reg(15 downto 8) <=
data_fw;
87                     when X"0023" => src_port_reg(7 downto 0) <=
data_fw;
88                     when X"0024" => dst_port_reg(15 downto 8) <=
data_fw;
89                     when X"0025" => dst_port_reg(7 downto 0) <=
data_fw;
90                     when others => null;
91                 end case;
92
93                 if line_number = X"0025" then
94                     current_state <= READY;
95                 end if;

```

```

96         when READY =>
97             ready_reg <= '1';
98             current_state <= IDLE;
99         end case;
100     end if;
101 end if;
102 end if;
103 end process;
104
105 source_ip    <= src_ip_reg;
106 dest_ip      <= dst_ip_reg;
107 source_port  <= src_port_reg;
108 dest_port    <= dst_port_reg;
109 protocol     <= protocol_reg;
110 data_ready   <= ready_reg;
111
112 end Behavioral;

```

## A.8 Packet Analyzer testbench: PacketAnalyzer\_tb.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity PacketAnalyzer_tb is
6  end PacketAnalyzer_tb;
7
8  architecture tb of PacketAnalyzer_tb is
9      signal mac_rx_clk      : STD_LOGIC := '0';
10     signal rst              : STD_LOGIC := '0';
11     signal line_number      : STD_LOGIC_VECTOR(15 downto 0) := (others => '0');
12     signal data_fw          : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
13     signal start_of_frame   : STD_LOGIC := '0';
14     signal end_of_frame     : STD_LOGIC := '0';
15     signal fw_out           : STD_LOGIC_VECTOR(1 downto 0);
16
17     signal source_ip        : STD_LOGIC_VECTOR(31 downto 0);
18     signal dest_ip          : STD_LOGIC_VECTOR(31 downto 0);
19     signal source_port      : STD_LOGIC_VECTOR(15 downto 0);
20     signal dest_port        : STD_LOGIC_VECTOR(15 downto 0);
21     signal protocol         : STD_LOGIC_VECTOR(7 downto 0);
22     signal data_ready       : STD_LOGIC;
23     signal fw_out_check     : STD_LOGIC_VECTOR(1 downto 0);
24
25
26     signal fifo_data        : STD_LOGIC_VECTOR(7 downto 0);
27     signal fifo_sof         : STD_LOGIC;
28     signal fifo_eof         : STD_LOGIC;
29

```

```

30     component PacketAnalyzer is
31         Port (
32             mac_rx_clk      : in  STD_LOGIC;
33             rst              : in  STD_LOGIC;
34             line_number     : in  STD_LOGIC_VECTOR(15 downto 0);
35             data_fw          : in  STD_LOGIC_VECTOR(7 downto 0);
36             start_of_frame  : in  STD_LOGIC;
37             end_of_frame    : in  STD_LOGIC;
38             fw_out          : in  STD_LOGIC_VECTOR(1 downto 0);
39
40             source_ip       : out STD_LOGIC_VECTOR(31 downto 0);
41             dest_ip         : out STD_LOGIC_VECTOR(31 downto 0);
42             source_port     : out STD_LOGIC_VECTOR(15 downto 0);
43             dest_port       : out STD_LOGIC_VECTOR(15 downto 0);
44             protocol        : out STD_LOGIC_VECTOR(7 downto 0);
45             data_ready      : out STD_LOGIC;
46             fw_out_check   : out STD_LOGIC_VECTOR(1 downto 0);
47
48             fifo_data       : out STD_LOGIC_VECTOR(7 downto 0);
49             fifo_sof        : out STD_LOGIC;
50             fifo_eof        : out STD_LOGIC
51         );
52     end component;
53
54     -- Example packet (including only relevant bytes for brevity)
55     type byte_array is array (natural range <>) of STD_LOGIC_VECTOR(7
downto 0);
56     constant test_packet : byte_array := (
57         x"01", x"10", x"A4", x"7B", x"EA", x"80", -- Dest MAC
58         x"00", x"12", x"34", x"56", x"78", x"90", -- Src MAC
59         x"08", x"00", -- EtherType: IPv4
60         x"45", x"00", x"00", x"2E", x"B3", x"FE", x"00", x"00", x"80", --
IP header start
61         x"11", -- Byte 23: Protocol (
UDP = 17 = x"11")
62         x"05", x"40", -- Checksum etc.
63         x"C0", x"A8", x"00", x"2C", -- Source IP:
192.168.0.44
64         x"C0", x"A8", x"00", x"04", -- Dest IP:
192.168.0.4
65         x"04", x"00", -- Source Port: 1024
66         x"04", x"00", -- Dest Port: 1024
67         x"00", x"1A", x"2D", x"E8", -- Payload...
68         x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
69         x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
70         x"10", x"11", -- More payload
71         x"E6", x"C5", x"3D", x"B2" -- FCS
72     );
73
74     constant CLK_PERIOD : time := 10 ns;
75
76 begin
77     -- DUT Instantiation

```

```

78     uut: PacketAnalyzer port map (
79         mac_rx_clk      => mac_rx_clk,
80         rst             => rst,
81         line_number     => line_number,
82         data_fw         => data_fw,
83         start_of_frame  => start_of_frame,
84         end_of_frame    => end_of_frame,
85         fw_out          => fw_out,
86
87         source_ip       => source_ip,
88         dest_ip         => dest_ip,
89         source_port     => source_port,
90         dest_port       => dest_port,
91         protocol        => protocol,
92         data_ready      => data_ready,
93         fw_out_check    => fw_out_check,
94
95         fifo_data       => fifo_data,
96         fifo_sof        => fifo_sof,
97         fifo_eof        => fifo_eof
98     );
99
100    -- Clock generation
101    clk_process : process
102    begin
103        while true loop
104            mac_rx_clk <= '0';
105            wait for CLK_PERIOD / 2;
106            mac_rx_clk <= '1';
107            wait for CLK_PERIOD / 2;
108        end loop;
109    end process;
110
111    -- Stimulus process
112    stim_proc: process
113    begin
114        wait for 3 * CLK_PERIOD;
115
116        -- Start of tx
117        for i in 0 to test_packet'length - 1 loop
118            data_fw <= test_packet(i);
119            line_number <= std_logic_vector(to_unsigned(i, 16));
120            fw_out <= "00"; -- inizialmente 00 durante la ricezione
121
122            -- Setting start of frame
123            if i = 0 then
124                start_of_frame <= '1';
125            else
126                start_of_frame <= '0';
127            end if;
128
129            -- Setting end of frame
130            if i = test_packet'length - 1 then

```

```

131         end_of_frame <= '1';
132     else
133         end_of_frame <= '0';
134     end if;
135
136     wait for CLK_PERIOD;
137 end loop;
138
139 -- Send FCS
140 start_of_frame <= '0';
141 end_of_frame <= '0';
142 data_fw <= (others => '0');
143 wait for CLK_PERIOD;
144 fw_out <= "10"; -- FCS segnala che il pacchetto   valido
145
146 wait for 100 ns;
147
148 wait;
149 end process;
150
151
152 end tb;

```

## A.9 Check Rules: CheckRules.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity CheckRules is
6     Port (
7         clk          : in  STD_LOGIC;
8         rst          : in  STD_LOGIC;
9         data_ready   : in  STD_LOGIC;
10        fw_out_in    : in  STD_LOGIC_VECTOR(1 downto 0);
11        src_ip       : in  STD_LOGIC_VECTOR(31 downto 0);
12        dst_ip       : in  STD_LOGIC_VECTOR(31 downto 0);
13        protocol     : in  STD_LOGIC_VECTOR(7 downto 0);
14        src_port     : in  STD_LOGIC_VECTOR(15 downto 0);
15        dst_port     : in  STD_LOGIC_VECTOR(15 downto 0);
16        fw_result    : out STD_LOGIC_VECTOR(1 downto 0)
17    );
18 end CheckRules;
19
20 architecture Behavioral of CheckRules is
21
22     type rule_t is record
23         src_ip       : STD_LOGIC_VECTOR(31 downto 0);
24         dst_ip       : STD_LOGIC_VECTOR(31 downto 0);
25         protocol     : STD_LOGIC_VECTOR(7 downto 0);
26         src_port     : STD_LOGIC_VECTOR(15 downto 0);

```

```

27     dst_port      : STD_LOGIC_VECTOR(15 downto 0);
28     src_ip_mask   : STD_LOGIC_VECTOR(31 downto 0);
29     dst_ip_mask   : STD_LOGIC_VECTOR(31 downto 0);
30     protocol_mask : STD_LOGIC_VECTOR(7  downto 0);
31     src_port_mask : STD_LOGIC_VECTOR(15 downto 0);
32     dst_port_mask : STD_LOGIC_VECTOR(15 downto 0);
33     allow         : STD_LOGIC;
34 end record;
35
36 type rule_array_t is array (0 to 3) of rule_t;
37
38 -- Rule list
39 constant rules : rule_array_t := (
40
41     -- Block UDP from 192.168.1.1 to 192.168.1.2
42     (x"COA80101", x"COA80102", x"11", x"1F90", x"0050",
43      x"FFFFFFFF", x"FFFFFFFF", x"FF", x"FFFF", x"FFFF",
44      '0'),
45
46     -- Allow specific UDP packet
47     (x"COA8002C", x"COA80004", x"11", x"0400", x"0400",
48      x"FFFFFFFF", x"FFFFFFFF", x"FF", x"FFFF", x"FFFF",
49      '1'), -- ALLOW
50
51     -- Block TCP from 192.168.1.1 to 192.168.1.2
52     (x"COA80101", x"COA80102", x"06", x"1F90", x"0050",
53      x"FFFFFFFF", x"FFFFFFFF", x"FF", x"FFFF", x"FFFF",
54      '0'),
55
56     -- Default allow for TCP (example of lower-priority ALLOW)
57     (x"00000000", x"00000000", x"06", x"0000", x"0000",
58      x"00000000", x"00000000", x"FF", x"0000", x"0000",
59      '1')
60 );
61
62 signal rule_result_ready : STD_LOGIC := '0';
63 signal rule_result_value : STD_LOGIC := '0';
64
65 begin
66
67     process(clk, rst)
68         variable matched_allow : boolean := false;
69         variable matched_block : boolean := false;
70         variable allow_match_value : STD_LOGIC := '0';
71     begin
72         if rst = '1' then
73             rule_result_ready <= '0';
74             rule_result_value <= '0';
75             fw_result        <= (others => '0');
76         elsif rising_edge(clk) then
77
78             -- Check rules only when data is ready
79             if data_ready = '1' then

```



```

80         matched_allow := false;
81         matched_block := false;
82
83         for i in 0 to rules'high loop
84             if (src_ip and rules(i).src_ip_mask) = (rules(i).
src_ip and rules(i).src_ip_mask) and
85                 (dst_ip and rules(i).dst_ip_mask) = (rules(i).
dst_ip and rules(i).dst_ip_mask) and
86                 (protocol and rules(i).protocol_mask) = (rules(i).
protocol and rules(i).protocol_mask) and
87                 (src_port and rules(i).src_port_mask) = (rules(i).
src_port and rules(i).src_port_mask) and
88                 (dst_port and rules(i).dst_port_mask) = (rules(i).
dst_port and rules(i).dst_port_mask) then
89
90                 if rules(i).allow = '0' then
91                     matched_block := true;
92                     exit; -- No need to check more rules; block
wins
93
94                     elsif not matched_allow then
95                         matched_allow := true;
96                         allow_match_value := '1'; -- Save allow
decision
97
98                     end if;
99
100                 end if;
101             end loop;
102
103             if matched_block then
104                 rule_result_value <= '0'; -- BLOCK
105             elsif matched_allow then
106                 rule_result_value <= allow_match_value; -- ALLOW
107             else
108                 rule_result_value <= '0'; -- Default deny
109             end if;
110
111             rule_result_ready <= '1';
112         end if;
113
114         -- Output result only when FCS is ready
115         if fw_out_in(1) = '1' and rule_result_ready = '1' then
116             fw_result(1) <= '1'; -- Result ready
117             fw_result(0) <= rule_result_value and not fw_out_in(0); --
Apply FCS check
118             rule_result_ready <= '0'; -- Reset for next packet
119         else
120             fw_result(1) <= '0'; -- No result ready
121             fw_result(0) <= '0';
122         end if;
123     end if;
end process;
end Behavioral;

```

## A.10 Check Rules testbench: tb\_CheckRules.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity CheckRules_tb is
6  end CheckRules_tb;
7
8  architecture Behavioral of CheckRules_tb is
9
10     -- Component declaration
11     component CheckRules
12         Port (
13             clk          : in  STD_LOGIC;
14             rst          : in  STD_LOGIC;
15             data_ready   : in  STD_LOGIC;
16             fw_out_in    : in  STD_LOGIC_VECTOR(1 downto 0);
17             src_ip       : in  STD_LOGIC_VECTOR(31 downto 0);
18             dst_ip       : in  STD_LOGIC_VECTOR(31 downto 0);
19             protocol     : in  STD_LOGIC_VECTOR(7 downto 0);
20             src_port     : in  STD_LOGIC_VECTOR(15 downto 0);
21             dst_port     : in  STD_LOGIC_VECTOR(15 downto 0);
22             fw_result    : out STD_LOGIC_VECTOR(1 downto 0)
23         );
24     end component;
25
26     -- Signals for testing
27     signal clk          : STD_LOGIC := '0';
28     signal rst          : STD_LOGIC := '0';
29     signal data_ready   : STD_LOGIC := '0';
30     signal fw_out_in    : STD_LOGIC_VECTOR(1 downto 0) := (others => '0');
31
32     signal src_ip       : STD_LOGIC_VECTOR(31 downto 0);
33     signal dst_ip       : STD_LOGIC_VECTOR(31 downto 0);
34     signal protocol     : STD_LOGIC_VECTOR(7 downto 0);
35     signal src_port     : STD_LOGIC_VECTOR(15 downto 0);
36     signal dst_port     : STD_LOGIC_VECTOR(15 downto 0);
37     signal fw_result    : STD_LOGIC_VECTOR(1 downto 0);
38
39     constant clk_period : time := 10 ns;
40
41 begin
42
43     -- Instantiate the unit under test
44     uut: CheckRules
45         port map (
46             clk          => clk,
47             rst          => rst,
48             data_ready   => data_ready,
49             fw_out_in    => fw_out_in,
50             src_ip       => src_ip,

```

```

50         dst_ip      => dst_ip,
51         protocol    => protocol,
52         src_port    => src_port,
53         dst_port    => dst_port,
54         fw_result   => fw_result
55     );
56
57 -- Clock process
58 clk_process : process
59 begin
60     while now < 400 ns loop
61         clk <= '0';
62         wait for clk_period / 2;
63         clk <= '1';
64         wait for clk_period / 2;
65     end loop;
66     wait;
67 end process;
68
69 -- Stimulus process
70 stim_proc: process
71 begin
72     -- Reset
73     rst <= '1';
74     wait for 20 ns;
75     rst <= '0';
76
77     -- Test 1: Match rule 0 (BLOCK) + valid FCS
78     src_ip      <= x"C0A80101";
79     dst_ip      <= x"C0A80102";
80     protocol    <= x"06"; -- tcp specific block
81     src_port    <= x"1F90";
82     dst_port    <= x"0050";
83     data_ready  <= '1';
84     wait for clk_period;
85     data_ready  <= '0';
86
87     -- Simulate FCS ready and valid after a few cycles
88     wait for 2 * clk_period;
89     fw_out_in <= "10"; --(FCS OK)
90     wait for clk_period;
91     fw_out_in <= "00";
92
93     -- Test 2: Match rule 1 (ALLOW) + invalid FCS
94     wait for clk_period;
95     src_ip      <= x"C0A8002C";
96     dst_ip      <= x"C0A80004";
97     protocol    <= x"11"; --udp specific allow
98     src_port    <= x"0400";
99     dst_port    <= x"0400";
100    data_ready  <= '1';
101    wait for clk_period;
102    data_ready  <= '0';

```

```
103
104     wait for 2 * clk_period;
105     fw_out_in <= "11";  -- (FCS ERROR)
106     wait for clk_period;
107     fw_out_in <= "00";
108
109     -- Test 3: No match (default DENY) + valid FCS
110     wait for clk_period;
111     src_ip      <= x"C0A80032";  -- 192.168.0.50
112     dst_ip      <= x"C0A80064";  -- 192.168.0.100
113     protocol    <= x"11";       -- UDP (11)
114     src_port    <= x"1388";     -- 5000
115     dst_port    <= x"0035";     -- 53
116     data_ready  <= '1';
117     wait for clk_period;
118     data_ready  <= '0';
119
120     wait for 2 * clk_period;
121     fw_out_in <= "10";  -- FCS OK
122     wait for clk_period;
123     fw_out_in <= "00";
124
125     -- Test 4: Match rule 2 (ALLOW) + valid FCS
126     wait for clk_period;
127     src_ip      <= x"C0A8002C";
128     dst_ip      <= x"C0A80004";  -- udp specific allow
129     protocol    <= x"11";
130     src_port    <= x"0400";
131     dst_port    <= x"0400";
132     data_ready  <= '1';
133     wait for clk_period;
134     data_ready  <= '0';
135
136     wait for 2 * clk_period;
137     fw_out_in <= "10";  -- FCS OK
138     wait for clk_period;
139     fw_out_in <= "00";
140
141     -- Test 5: Allow TCP general packet + valid FCS
142     wait for clk_period;
143     src_ip      <= x"C0A80103";
144     dst_ip      <= x"C0A80104";
145     protocol    <= x"06";  -- tcp general allow
146     src_port    <= x"1F90";
147     dst_port    <= x"0050";
148     data_ready  <= '1';
149     wait for clk_period;
150     data_ready  <= '0';
151
152     wait for 2 * clk_period;
153     fw_out_in <= "10";  -- FCS OK
154     wait for clk_period;
155     fw_out_in <= "00";
```

```

156         wait;
157     end process;
158
159 end Behavioral;
160

```

## A.11 FIFO: async\_fifo.vhd

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  entity async_fifo is
6      generic (
7          FIFO_DEPTH : integer := 1522;
8          PTR_WIDTH   : integer := 11
9      );
10     port (
11         reset          : in std_logic;
12         wclk            : in std_logic;
13         rclk            : in std_logic;
14         write_enable    : in std_logic;
15         write_data_in   : in std_logic_vector(7 downto 0);
16         SOP             : in std_logic;
17         EOP             : in std_logic;
18         FW_RESULT       : in std_logic_vector(1 downto 0); -- (0) = send to
19         real, (1) = valid
20         fifo_occu_in    : out std_logic_vector(PTR_WIDTH - 1 downto 0);
21         fifo_occu_out   : out std_logic_vector(PTR_WIDTH - 1 downto 0);
22         read_data_out   : out std_logic_vector(7 downto 0);
23         dummy_data_out  : out std_logic_vector(7 downto 0);
24         fifo_full       : out std_logic;
25         fifo_empty      : out std_logic
26     );
27 end async_fifo;
28
29 architecture Behavioral of async_fifo is
30
31     type memory_array is array (0 to FIFO_DEPTH - 1) of std_logic_vector(7
32     downto 0);
33     signal mem : memory_array := (others => (others => '0'));
34
35     -- Pointers
36     signal wptr_bin : std_logic_vector(PTR_WIDTH - 1 downto 0) := (others
37     => '0');
38     signal rptr_bin : std_logic_vector(PTR_WIDTH - 1 downto 0) := (others
39     => '0');
40     signal wptr_gray : std_logic_vector(PTR_WIDTH - 1 downto 0) := (others
41     => '0');
42     signal rptr_gray : std_logic_vector(PTR_WIDTH - 1 downto 0) := (others
43     => '0');
44

```

```

38
39 -- Sync
40 signal rp_ptr_gray_sync1, rp_ptr_gray_sync2 : std_logic_vector(PTR_WIDTH -
41 1 downto 0) := (others => '0');
42
43 signal wp_ptr_gray_sync1, wp_ptr_gray_sync2 : std_logic_vector(PTR_WIDTH -
44 1 downto 0) := (others => '0');
45
46 -- Flags
47 signal full_flag, empty_flag : std_logic := '0';
48
49 -- Packet state
50 signal packet_ready      : std_logic := '0';
51 signal packet_result     : std_logic := '0'; -- 1: real output, 0: dummy
52 signal packet_end_ptr    : std_logic_vector(PTR_WIDTH - 1 downto 0) := (
53 others => '0');
54
55 signal reading           : std_logic := '0';
56
57 -- Functions
58 function bin2gray(bin : std_logic_vector) return std_logic_vector is
59     variable gray : std_logic_vector(bin'range);
60 begin
61     gray(bin'high) := bin(bin'high);
62     for i in bin'high - 1 downto bin'low loop
63         gray(i) := bin(i+1) xor bin(i);
64     end loop;
65     return gray;
66 end function;
67
68 function gray2bin(gray : std_logic_vector) return std_logic_vector is
69     variable bin : std_logic_vector(gray'range);
70 begin
71     bin(bin'high) := gray(bin'high);
72     for i in bin'high - 1 downto bin'low loop
73         bin(i) := bin(i+1) xor gray(i);
74     end loop;
75     return bin;
76 end function;
77
78 begin
79
80 -- WRITE
81 process(wclk, reset)
82 begin
83     if reset = '1' then
84         wp_ptr_bin <= (others => '0');
85         wp_ptr_gray <= (others => '0');
86     elsif rising_edge(wclk) then
87         -- Handle end of packet (EOP)
88         if EOP = '1' then
89             packet_end_ptr <= wp_ptr_bin;
90         end if;

```

```

88      -- Write operation occurs when SOP = '1' and write_enable is
also '1'
89      if write_enable = '1' and full_flag = '0' then
90          mem(to_integer(unsigned(wptra_bin))) <= write_data_in;
91          wptra_bin <= std_logic_vector(unsigned(wptra_bin) + 1);
92          wptra_gray <= bin2gray(std_logic_vector(unsigned(wptra_bin)
+ 1));
93      end if;
94  end if;
95  end process;
96
97  -- READ
98  process(rclk, reset)
99  begin
100      if reset = '1' then
101          rptra_bin <= (others => '0');
102          rptra_gray <= (others => '0');
103          packet_ready <= '0';
104          packet_result <= '0';
105          reading <= '0';
106      elsif rising_edge(rclk) then
107
108          -- Accept FW_RESULT if a packet is waiting
109          if packet_ready = '0' and FW_RESULT(1) = '1' then
110              packet_ready <= '1';
111              packet_result <= FW_RESULT(0);
112              reading <= '1';
113          end if;
114
115          -- Read if allowed
116          if reading = '1' and empty_flag = '0' then
117              if packet_result = '1' then
118                  read_data_out <= mem(to_integer(unsigned(rptra_bin)));
119              else
120                  dummy_data_out <= mem(to_integer(unsigned(rptra_bin)));
121              end if;
122
123              rptra_bin <= std_logic_vector(unsigned(rptra_bin) + 1);
124              rptra_gray <= bin2gray(std_logic_vector(unsigned(rptra_bin)
+ 1));
125
126          -- End of packet reading
127          if rptra_bin = packet_end_ptr then
128              reading <= '0';
129              packet_ready <= '0';
130          end if;
131      end if;
132  end if;
133  end process;
134
135  -- SYNC pointers
136  process(wclk)
137  begin

```

```

138         if rising_edge(wclk) then
139             rp_ptr_gray_sync1 <= rp_ptr_gray;
140             rp_ptr_gray_sync2 <= rp_ptr_gray_sync1;
141         end if;
142     end process;
143
144     process(rclk)
145     begin
146         if rising_edge(rclk) then
147             wp_ptr_gray_sync1 <= wp_ptr_gray;
148             wp_ptr_gray_sync2 <= wp_ptr_gray_sync1;
149         end if;
150     end process;
151
152     -- STATUS flags
153     process(wp_ptr_gray, rp_ptr_gray_sync2, wp_ptr_bin)
154     begin
155         if (bin2gray(std_logic_vector(unsigned(wp_ptr_bin) + 1)) =
156 rp_ptr_gray_sync2) then
157             full_flag <= '1';
158         else
159             full_flag <= '0';
160         end if;
161     end process;
162
163     process(rp_ptr_gray, wp_ptr_gray_sync2)
164     begin
165         if rp_ptr_gray = wp_ptr_gray_sync2 then
166             empty_flag <= '1';
167         else
168             empty_flag <= '0';
169         end if;
170     end process;
171
172     -- OUTPUTS
173     fifo_occu_in <= std_logic_vector(to_unsigned(to_integer(unsigned(
174 wp_ptr_bin)) - to_integer(unsigned(gray2bin(rp_ptr_gray_sync2))), PTR_WIDTH
175 ));
176     fifo_occu_out <= std_logic_vector(to_unsigned(to_integer(unsigned(
177 gray2bin(wp_ptr_gray_sync2))) - to_integer(unsigned(rp_ptr_bin)), PTR_WIDTH
178 ));
179     fifo_full <= full_flag;
180     fifo_empty <= empty_flag;
181
182 end Behavioral;

```

## A.12 FIFO testbench: async\_fifo\_tb.vhd

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;

```



```
4
5 entity async_fifo_tb is
6 end async_fifo_tb;
7
8 architecture behavior of async_fifo_tb is
9
10     constant CLK_PERIOD : time := 10 ns;
11
12     -- Component declaration
13     component async_fifo is
14         generic (
15             FIFO_DEPTH : integer := 16;
16             PTR_WIDTH   : integer := 4
17         );
18         port (
19             reset          : in std_logic;
20             wclk            : in std_logic;
21             rclk            : in std_logic;
22             write_enable    : in std_logic;
23             write_data_in   : in std_logic_vector(7 downto 0);
24             SOP             : in std_logic;
25             EOP             : in std_logic;
26             FW_RESULT       : in std_logic_vector(1 downto 0);
27             fifo_occu_in    : out std_logic_vector(PTR_WIDTH - 1 downto 0);
28             fifo_occu_out   : out std_logic_vector(PTR_WIDTH - 1 downto 0);
29             read_data_out   : out std_logic_vector(7 downto 0);
30             dummy_data_out  : out std_logic_vector(7 downto 0);
31             fifo_full       : out std_logic;
32             fifo_empty      : out std_logic
33         );
34     end component;
35
36     -- Signals
37     signal reset          : std_logic := '1';
38     signal wclk, rclk     : std_logic := '0';
39     signal write_enable    : std_logic := '0';
40     signal write_data_in   : std_logic_vector(7 downto 0) := (others => '0');
41 );
42     signal SOP            : std_logic := '0';
43     signal EOP            : std_logic := '0';
44     signal FW_RESULT      : std_logic_vector(1 downto 0) := (others => '0');
45 );
46     signal fifo_occu_in   : std_logic_vector(3 downto 0);
47     signal fifo_occu_out  : std_logic_vector(3 downto 0);
48     signal read_data_out  : std_logic_vector(7 downto 0);
49     signal dummy_data_out : std_logic_vector(7 downto 0);
50     signal fifo_full, fifo_empty : std_logic;
51
52 begin
53     -- Clock generation
54     wclk_proc : process
55         begin
```

```

55         while true loop
56             wclk <= '0';
57             wait for CLK_PERIOD/2;
58             wclk <= '1';
59             wait for CLK_PERIOD/2;
60         end loop;
61     end process;
62
63     rclk_proc : process
64     begin
65         while true loop
66             rclk <= '0';
67             wait for CLK_PERIOD/2;
68             rclk <= '1';
69             wait for CLK_PERIOD/2;
70         end loop;
71     end process;
72
73     -- DUT
74     uut: async_fifo
75         generic map (
76             FIFO_DEPTH => 16,
77             PTR_WIDTH  => 4
78         )
79         port map (
80             reset          => reset,
81             wclk            => wclk,
82             rclk            => rclk,
83             write_enable    => write_enable,
84             write_data_in   => write_data_in,
85             SOP             => SOP,
86             EOP             => EOP,
87             FW_RESULT       => FW_RESULT,
88             fifo_occu_in    => fifo_occu_in,
89             fifo_occu_out   => fifo_occu_out,
90             read_data_out   => read_data_out,
91             dummy_data_out  => dummy_data_out,
92             fifo_full       => fifo_full,
93             fifo_empty      => fifo_empty
94         );
95
96     -- Stimulus
97     stim_proc : process
98     begin
99         wait for 20 ns;
100         reset <= '0';
101
102         -- First packet: valid, goes to real output
103         SOP <= '1';
104         write_enable <= '1';
105         write_data_in <= x"AA";
106         wait for CLK_PERIOD;
107

```

```
108     SOP <= '0';
109     write_data_in <= x"BB";
110     wait for CLK_PERIOD;
111
112     write_data_in <= x"CC";
113     EOP <= '1';
114     wait for CLK_PERIOD;
115
116     EOP <= '0';
117     write_enable <= '0';
118
119     wait for 40 ns;
120
121     -- Firewall result: valid + accepted
122     FW_RESULT <= "11";
123     wait for 40 ns;
124
125     -- Clear FW_RESULT
126     FW_RESULT <= "00";
127
128     -- Second packet: invalid, goes to dummy output
129     SOP <= '1';
130     write_enable <= '1';
131     write_data_in <= x"11";
132     wait for CLK_PERIOD;
133
134     SOP <= '0';
135     write_data_in <= x"22";
136     wait for CLK_PERIOD;
137
138     write_data_in <= x"33";
139     EOP <= '1';
140     wait for CLK_PERIOD;
141
142     EOP <= '0';
143     write_enable <= '0';
144
145     wait for 40 ns;
146
147     -- Firewall result: valid + rejected
148     FW_RESULT <= "10";
149
150     wait for 100 ns;
151     wait;
152 end process;
153
154 end behavior;
```

## A.13 Full System: TopPackCheck.vhd

```

1  --Top Level Entity including: MAC_RX_CONTROL, Packet Analyzer, CheckRules,
   FIFO
2
3  library IEEE;
4  use IEEE.STD_LOGIC_1164.ALL;
5  use IEEE.NUMERIC_STD.ALL;
6
7  entity TopPackCheck is
8      Port (
9          rst          : in  STD_LOGIC;
10         clk          : in  std_logic;
11         reset        : in  std_logic;
12         valid        : in  std_logic;
13         data_in       : in  std_logic_vector(7 downto 0);
14         frame_length  : in  std_logic_vector(15 downto 0);
15         fifo_write_enable : in  STD_LOGIC;
16         fw_result     : out STD_LOGIC_VECTOR(1 downto 0);
17         read_data     : out STD_LOGIC_VECTOR(7 downto 0);
18         dummy_data    : out STD_LOGIC_VECTOR(7 downto 0)
19     );
20 end TopPackCheck;
21
22 architecture Behavioral of TopPackCheck is
23
24
25     --signals between MAC_RX and FCS
26     signal mac_rx_reset : std_logic;
27     signal mac_rx_clk : std_logic;
28     signal mac_rx_valid_1 : std_logic;
29     signal mac_rx_start_of_frame : std_logic;
30     signal mac_rx_end_of_frame : std_logic;
31     signal mac_rx_data_out : std_logic_vector(7 downto 0);
32
33     -- signals between FCS and MAC_RX_CONTROL
34     signal fcs_error_signal : std_logic_vector(1 downto 0);
35     --signal reset          : std_logic;
36     --signal valid          : std_logic;
37     signal fcs_last        : std_logic;
38     signal fcs_first       : std_logic;
39     signal fcs_data_out    : std_logic_vector(7 downto 0);
40
41     -- signals between MAC_RX_CONTROL and PacketAnalyzer
42     signal line_number     : STD_LOGIC_VECTOR(15 downto 0);
43     signal fw_out          : STD_LOGIC_VECTOR(1 downto 0);
44     signal data_in_1       : STD_LOGIC_VECTOR(7 downto 0);
45     signal mac_end         : std_logic;
46     signal mac_start       : std_logic;
47     signal mac_rxd         : STD_LOGIC_VECTOR(7 downto 0);
48     signal mac_rx_valid    : std_logic;
49     signal mac_rx_last     : std_logic;
50     signal mac_rx_err      : std_logic_vector(1 downto 0);
51

```

```

52  -- PacketAnalyzer -> CheckRules
53  signal src_ip_sig      : STD_LOGIC_VECTOR(31 downto 0);
54  signal dst_ip_sig      : STD_LOGIC_VECTOR(31 downto 0);
55  signal protocol_sig    : STD_LOGIC_VECTOR(7 downto 0);
56  signal src_port_sig    : STD_LOGIC_VECTOR(15 downto 0);
57  signal dst_port_sig    : STD_LOGIC_VECTOR(15 downto 0);
58  signal data_ready_sig  : STD_LOGIC;
59  signal fw_out_sig      : STD_LOGIC_VECTOR(1 downto 0);  -- Output from
PacketAnalyzer
60
61  -- PacketAnalyzer -> FIFO
62  signal fifo_data       : STD_LOGIC_VECTOR(7 downto 0);
63  signal fifo_sof        : STD_LOGIC;
64  signal fifo_eof        : STD_LOGIC;
65
66  -- FIFO -> Top level
67  signal fifo_read_data_out : STD_LOGIC_VECTOR(7 downto 0);
68  signal fifo_dummy_data_out : STD_LOGIC_VECTOR(7 downto 0);
69
70  -- Internal firewall result signal
71  signal fw_result_sig    : STD_LOGIC_VECTOR(1 downto 0);
72
73  begin
74
75  mac_rx : entity work.MAC_RX
76  port map (
77      GMII_RX_CLK      => clk,
78      GMII_RXD         => data_in,
79      GMII_RX_DV       => valid,
80      GMII_RX_RESET    => reset,
81      TOTAL_LENGTH     => frame_length,
82
83
84      MAC_RX_RESET     => mac_rx_reset,
85      MAC_RX_CLK       => mac_rx_clk,
86      MAC_RX_VALID     => mac_rx_valid_1,
87      MAC_RX_FIRST     => mac_rx_start_of_frame,
88      MAC_RX_LAST      => mac_rx_end_of_frame,
89      MAC_RXD          => mac_rx_data_out
90  );
91
92  fcs_inst : entity work.fcs
93  port map (
94      clk              => mac_rx_clk,
95      reset            => mac_rx_reset,
96      valid            => mac_rx_valid_1,
97      start_of_frame   => mac_rx_start_of_frame,
98      end_of_frame     => mac_rx_end_of_frame,
99      data_in          => mac_rx_data_out,
100
101      fcs_error        => fcs_error_signal,
102      last_of_frame    => fcs_last,
103      first_of_frame   => fcs_first,

```

```

104         data_out          => fcs_data_out
105     );
106
107     -- MAC_RX_CONTROL instance
108     MAC_RX_CONTROL_inst : entity work.MAC_RX_CONTROL
109     port map (
110         MAC_RX_CLK        => clk,
111         MAC_RXD            => fcs_data_out,
112         MAC_RX_VALID      => fcs_first,
113         MAC_RX_LAST       => fcs_last,
114         MAC_RX_ERR        => fcs_error_signal,
115
116         LINE_NUMBER       => line_number,
117         DATA              => data_in_1,
118         FW_OUT             => fw_out,                -- now 2-bit
119         START_OF_FRAME    => mac_start,
120         END_OF_FRAME      => mac_end
121     );
122
123     -- PacketAnalyzer instance
124     PacketAnalyzer_inst : entity work.PacketAnalyzer
125     port map (
126         mac_rx_clk        => clk,
127         line_number       => line_number,
128         data_fw           => data_in_1,
129         start_of_frame    => mac_start,
130         end_of_frame      => mac_end,
131         fw_out            => fw_out,                -- 2-bit input from
132     MAC_RX_CONTROL
133
134         source_ip         => src_ip_sig,
135         dest_ip           => dst_ip_sig,
136         source_port       => src_port_sig,
137         dest_port         => dst_port_sig,
138         protocol          => protocol_sig,
139         data_ready        => data_ready_sig,
140         fw_out_check      => fw_out_sig,            -- 2-bit output to
141     CheckRules
142
143         fifo_data         => fifo_data,
144         fifo_sof          => fifo_sof,
145         fifo_eof          => fifo_eof
146     );
147
148     -- CheckRules instance
149     CheckRules_inst : entity work.CheckRules
150     port map (
151         clk                => clk,
152         rst                => rst,
153         data_ready        => data_ready_sig,
154         src_ip            => src_ip_sig,
155         dst_ip            => dst_ip_sig,
156         protocol          => protocol_sig,
157         src_port          => src_port_sig,

```

```

155         dst_port      => dst_port_sig,
156         fw_out_in     => fw_out_sig,           -- input from
PacketAnalyzer
157         fw_result     => fw_result_sig
158     );
159
160     -- FIFO instance
161     FIFO_inst : entity work.async_fifo
162     port map (
163         reset          => rst,
164         wclk           => clk,
165         rclk           => clk,
166         write_enable   => fifo_write_enable,
167         write_data_in  => fifo_data,
168         SOP            => fifo_sof,
169         EOP            => fifo_eof,
170         FW_RESULT      => fw_result_sig,
171         fifo_occu_in   => open,
172         fifo_occu_out  => open,
173         read_data_out  => fifo_read_data_out,
174         dummy_data_out => fifo_dummy_data_out,
175         fifo_full      => open,
176         fifo_empty     => open
177     );
178
179     -- Top-level outputs
180     read_data  <= fifo_read_data_out;
181     dummy_data <= fifo_dummy_data_out;
182     fw_result  <= fw_result_sig;
183
184 end Behavioral;

```

## A.14 Full System testbench: tb\_TopPackCheck.vhd

```

1  -- Testbench for the Top Level Entity: sending a packet with the rule "
    allow" and FCS correct --> fw_result should output: 11
2
3  -- Firewall table in CheckRules block is:
4
5      -- (x"COA80101", x"COA80102", x"06", x"1F90", x"0050", '0'), -- block
6      -- (x"0A000001", x"0A000002", x"11", x"04D2", x"0035", '1'), -- allow
7      -- (x"COA8002C", x"COA80004", x"11", x"0400", x"0400", '1') -- allow
    --> THIS IS THE PACKET WE ARE TESTING IN THIS TESTBENCH
8
9  -- To test different packets, change the fields of src ip, dest ip,
    protocol, src port, dst port in the packet
10
11
12 library IEEE;
13 use IEEE.STD_LOGIC_1164.ALL;
14 use IEEE.NUMERIC_STD.ALL;

```

```

15
16 entity tb_TopPackCheck is
17 end tb_TopPackCheck;
18
19 architecture Behavioral of tb_TopPackCheck is
20
21     component TopPackCheck
22     Port (
23         rst          : in  STD_LOGIC;
24         clk          : in  std_logic;
25         reset        : in  std_logic;
26         valid        : in  std_logic;
27         data_in      : in  std_logic_vector(7 downto 0);
28         frame_length : in  std_logic_vector(15 downto 0);
29         fifo_write_enable : in std_logic;
30         fw_result    : out STD_LOGIC_VECTOR(1 downto 0);
31         read_data    : out STD_LOGIC_VECTOR(7 downto 0);
32         dummy_data   : out STD_LOGIC_VECTOR(7 downto 0)
33     );
34     end component;
35
36     signal clk          : std_logic := '0';
37     signal reset        : std_logic := '1';
38     signal valid        : std_logic := '0';
39     signal data_in      : std_logic_vector(7 downto 0) := (others => '0');
40     signal frame_length : std_logic_vector(15 downto 0) := (others =>
41         '0');
42     signal fw_result    : STD_LOGIC_VECTOR(1 downto 0);
43     signal rst          : std_logic := '0';
44     signal fifo_write_enable : STD_LOGIC := '0';
45     signal read_data    : STD_LOGIC_VECTOR(7 downto 0);
46     signal dummy_data   : STD_LOGIC_VECTOR(7 downto 0);
47     --signal fcs_error_signal : std_logic_vector(1 downto 0) ;
48
49     -- Clock generation
50     constant clk_period : time := 10 ns;
51     signal done : boolean := false;
52
53     -- Frame to test (72 bytes = 8 preamble + 60 dati + 4 CRC)
54     type frame_array is array (0 to 71) of std_logic_vector(7 downto 0);
55     constant test_frame : frame_array := (
56         x"55", x"55", x"55", x"55", x"55", x"55", x"55", x"D5", --preamble
57         x"00", x"10", x"A4", x"7B", x"EA", x"80", -- Dest MAC
58         x"00", x"12", x"34", x"56", x"78", x"90", -- Src MAC
59         x"08", x"00", -- EtherType: IPv4
60         x"45", x"00", x"00", x"2E", x"B3", x"FE", x"00", x"00", x"80", --
61         IP header start
62         x"11", -- Byte 23: Protocol (
63         UDP = 17 = x"11")
64         x"05", x"40", -- Checksum etc.
65         x"C0", x"A8", x"00", x"2C", -- src ip:
66         x"C0", x"A8", x"00", x"04", -- dst ip:
67         x"04", x"00", -- src port:

```



```

65         x"04", x"00",                                -- dst port:
66         x"00", x"1A", x"2D", x"E8",                  -- Payload...
67         x"00", x"01", x"02", x"03", x"04", x"05", x"06", x"07",
68         x"08", x"09", x"0A", x"0B", x"0C", x"0D", x"0E", x"0F",
69         x"10", x"11",                                -- More payload
70         x"E6", x"C5", x"3D", x"B2"                  -- FCS
71     );
72
73 begin
74
75     -- Instantiate the TopPackCheck module
76     uut: TopPackCheck
77     port map (
78         rst          => rst,
79         clk          => clk,
80         reset        => reset,
81         valid        => valid,
82         data_in      => data_in,
83         frame_length => frame_length,
84         fifo_write_enable => fifo_write_enable,
85         fw_result    => fw_result,
86         read_data    => read_data,
87         dummy_data   => dummy_data
88         --fcs_error_signal    => fcs_error_signal
89     );
90
91     -- Clock generation process
92     clk_process : process
93     begin
94         while not done loop
95             clk <= '0';
96             wait for clk_period / 2;
97             clk <= '1';
98             wait for clk_period / 2;
99         end loop;
100         wait;
101     end process;
102
103     -- Stimulus process
104     stimulus : process
105     begin
106         wait for 20 ns;
107         reset <= '0';
108         rst <= '0';
109         wait for 20 ns;
110         frame_length <= std_logic_vector(to_unsigned(test_frame'length, 16));
111         valid <= '1';
112         for i in 0 to (test_frame'length - 1) loop
113             data_in <= test_frame(i);
114             wait for clk_period;
115             if i = 1 then
116                 fifo_write_enable <= '1';
117             end if;

```

```
118   if i = (test_frame'length - 1) then
119       valid <= '0';
120   end if;
121
122   end loop;
123   wait for clk_period;
124   reset <= '1';
125   wait for clk_period;
126
127   reset <= '0';
128   wait for clk_period;
129   fifo_write_enable <= '0';
130   wait for 800 ns;
131   done <= true;
132   wait;
133   end process;
134
135 end Behavioral;
```

## B Contributions

Section	Name
Introduction 1	Gea Staropoli
Overview 2	Edoardo Santucci
Block specification 3	Everyone
Block description 4	Everyone
Simulation and verification 5	Everyone
Firewall System 6	Felix Hell
Conclusions 7	Gea Staropoli

Table 8: Contributions

For the sections titled "Block Specification" 3, "Block Description" 4, and "Simulation and verification" 5, each team member has contributed by writing and working on a specific block. The division of these blocks is detailed below.

Block	Name
GMII to MAC (3.1,4.1, 5.1)	Felix Hell
FCS (3.2,4.2, 5.2)	Gea Staropoli
MAC RX Control (3.3,4.3, 5.3)	Gea Staropoli
Packet Analyzer (3.4,4.4, 5.4)	Edoardo Santucci
Check Rules (3.5,4.5, 5.5)	Edoardo Santucci
FIFO (3.6,4.6, 5.6)	Felix Hell

Table 9: Block division