Final Project Report

34349 - FPGA Design for Communication Systems

Authors:
Dimitrios Triantis (s220023)
Katherine Cardoso Petulante Fernandes (s184454)
Rohit Imandi (s202369)

May 23, 2022

Contents

1 Contribution

Since this was a group project, the design of the **Gigabit Ethernet Switch** had been split into three parts. Each one of the members of the group was assigned a different part of the switch. To be more exact, Rohit Imandi (s202369) worked on the input module, Katherine Cardoso Petulante Fernandes (s184454) on the MAC learning module and Dimitrios Triantis (s220023) on the output module.

For the report, these are the chapters that each one of us contributed:
Rohit - 3, 4, 5, 8, 12 Katherine - 4, 6, 9, 11 Dimitrios - 4, 7, 10,11

2 Abstract

The aim of our project is to design and simulate a basic 4-port gigabit Ethernet switch with FCS-check (CRC-32), MAC-learning, appropriate OSI layer-2 switching architecture comprising of a fair queuing/scheduling algorithm. The project is divided into three parts namely, input module, MAC learning and output module.

3 Introduction

Ethernet is a family of wired computer networking technologies used in LANs, MANs and WANs. It is one of the most commonly used protocols for private and corporate local area networks (LANs). Ethernet LANs are often used to share files in schools, universities, hospitals and corporate offices due to its speed, security and dependability.
Ethernet protocol falls under the physical/data link layer (layer 2) of the OSI model (refer Fig 1) and is standardized as IEEE 802.3.

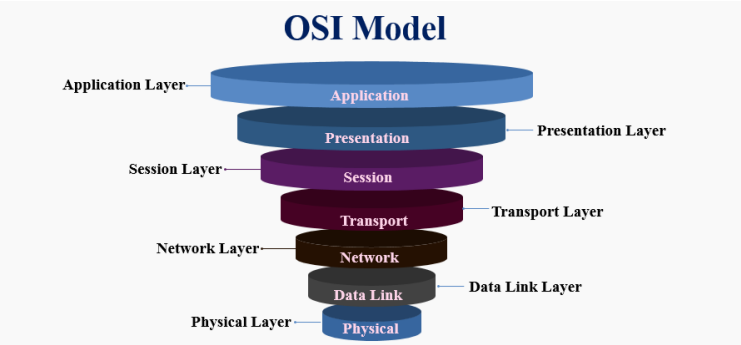


Figure 1: OSI Model

The frame format is shown below in Fig 2. The preamble of size 7 bytes and interframe gap of size 12 bytes after FCS (not shown in Fig 2) separates two ethernet frames. The start-of-frame delimiter of size 1 byte indicates the start of a particular ethernet frame. The destination and source addresses each of size 6 bytes are the physical MAC addresses of the source which generates the frame and the destination to which the frame is be transmitted. The data or payload has a size range of 46 - 1500 bytes and carries the data of the higher layers in OSI model. Finally, the frame check sequence (FCS) of size 4 bytes is an error detection mechanism to detect corrupted frames.

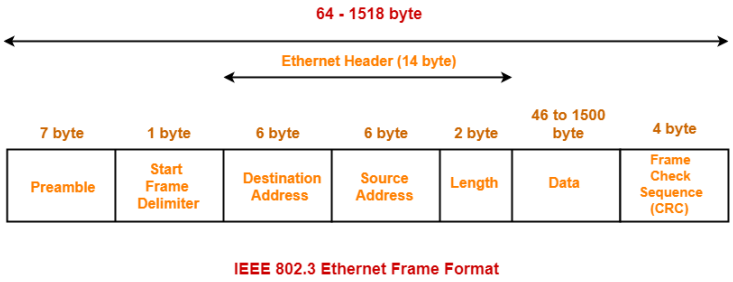


Figure 2: Ethernet Frame

The objective of our project is to design and simulate a basic 4-port gigabit ethernet switch for the standard ethernet protocol and frame format with FCS-check (CRC-32), appropriate layer-2 buffering architecture of OSI model comprising of a fair queuing/scheduling algorithm. The project is divided into three parts namely, input module, MAC learning and

output module. This report focuses on the input module of the switch which is responsible to first buffer the incoming frame and subsequently send it to the right output port based on the decision taken by the MAC learning module. Section 4 provides a brief overview of the entire switch describing each of the three parts, buffering architecture and scheduling algorithm used. Sections 5 6 and 7 describe in detail the design and functionality of various blocks used in the input module, MAC learning module and the output module of the switch. Sections 8, 9 and 10 describes the specifications the of various blocks used in the input module, MAC learning module and the output module of the switch. Section 11 discusses about test bench created, simulation and verification of the vhdl implementation of the design. The key performance parameters is like clock frequency and registers used are mentioned in section 12. Finally, the 13 provides a summary of the project, learnings and further improvements.

4 Overview of the Full System

As mentioned earlier, the switch comprises of three components namely, input module, MAC learning and output section. The switching and buffering architecture chosen is cross-point queuing as it mitigates the need for the control mechanism in the switching process. Although the memory required is of the order of N^2 , it is not an issue in this design. Fig 3 shows the various components as well as how they communicate with each other.

The I/O pins of the switch is mentioned below:

- 32 bit rx_data (input) indicating the data sent to the switch
- 4 bit rx_ctrl (input) indicating the port sending the data to the switch
- clk (input)
- reset (input)
- 32 bit tx_data (output) indicating the data to be sent to the right port
- 4 bit tx_ctrl (output) indicating the port which receives the data

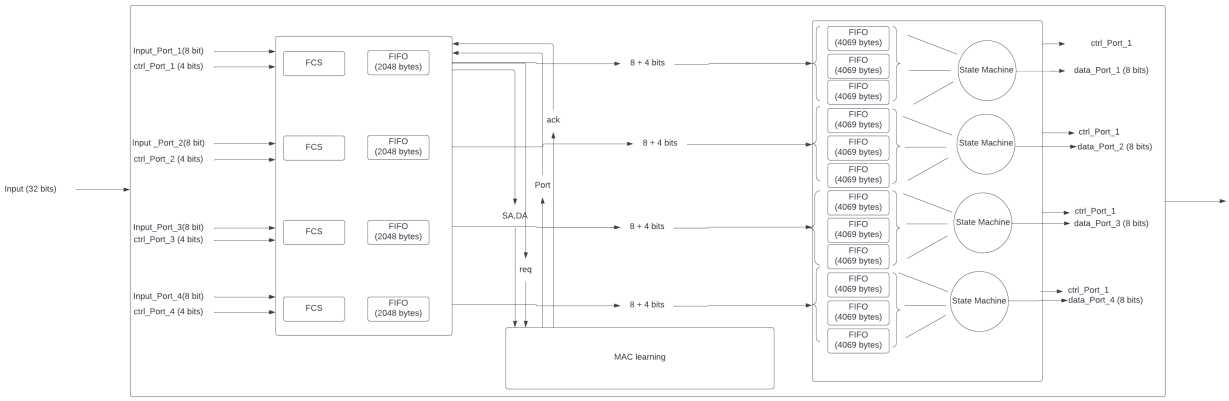


Figure 3: Components of the Switch

A brief overview of the three modules is mentioned below.

4.a Input Module

The input module of the switch comprising of a FIFO for each port continuously communicates with the MAC learning and output module. The received data is split into sets of 8-bits and sent to the corresponding buffers as well as an FCS block to check for any errors in the input module based on the rx_ctrl bit. The convention for rx_ctrl used in this design is:
0001 - data received from port one

0010 - data received from port two
0100 - data received from port three
1000 - data received from port four

After the data is stored, a request signal along with source and destination addresses from each port of the input module is sent to the MAC learning module. The MAC learning responds with an acknowledge signal as well as the output port for each request received. Thereafter, the data frame from each of FIFOs in the input module is sent to the output module along with the destination port and frame length.

4.b MAC Learning Module

The MAC learning module performs two different operations: the table lookup, in order to find the corresponding port address for the destination MAC address given in the packet and the MAC learning itself, which stores the source port and source MAC address into the table. To perform these operations, two inputs will be received in the block. The first one consists of a combination of the source and destination MAC addresses (48 + 48 bits) and the other is the request bit which indicates which port is requesting this operation to be performed. The table that stores the source MAC addresses along with their respective ports is a RAM memory. As the switch is designed to handle 8k MAC addresses, the size of the RAM is $2^{13} \times 52$ bits.
For the MAC learning operation, the source MAC address will be used. First, it will be hashed by using the CRC16 algorithm, excluding the last 3 bits since the table will only support 13 bit value for the addresses. The hashed value will correspond to the address in the table that both the MAC address is stored in the first 48 bits and the port number will be stored in the last 4 bits (52 bits).
For the table lookup operation, the destination MAC address will be used. It will be hashed as well with the same algorithm, therefore it will know in which address the corresponding port number for this MAC address is stored. The MAC address in the table is compared with the destination MAC address, and if it matches correctly, then the block will output the corresponding port number. If not, then the block will output the transmission control bit as broadcasting. The transmission control bit convention to transmit data is:
0001 - data sent to port one
0010 - data sent to port two
0100 - data sent to port three
1000 - data sent to port four
1111 - data broadcasted to all ports

4.c Output Module

The output module comprises of a demultiplexer and 3 FIFOs for each output port. There are only 3 FIFOs because the port at which the frame is received cannot be destination port. The cross-point switching operation is implemented using the demultiplexer and FIFOs. Fig 4 shows the architecture of a cross-point queue switch. As the frame arrives at the corresponding cross-point based on the port received from MAC learning, the demultiplexer selects the FIFO corresponding to the destination port. Finally, the 8-bits of the frame stored in the FIFO is multiplexed onto 32-bit output signal tx_data using round-robin scheduling algorithm. An example is mentioned below for better understanding.

Consider a frame stored in the FIFO of port 1 and the output port received from MAC learning is port 3 (destination port). This frame is sent to the port 3 where the demultiplexer then further sends it to the FIFO corresponding to port 1 (source port). The round-robin algorithm implemented as a state machine then multiplexes 8-bits from each the FIFO of each port onto tx_data.

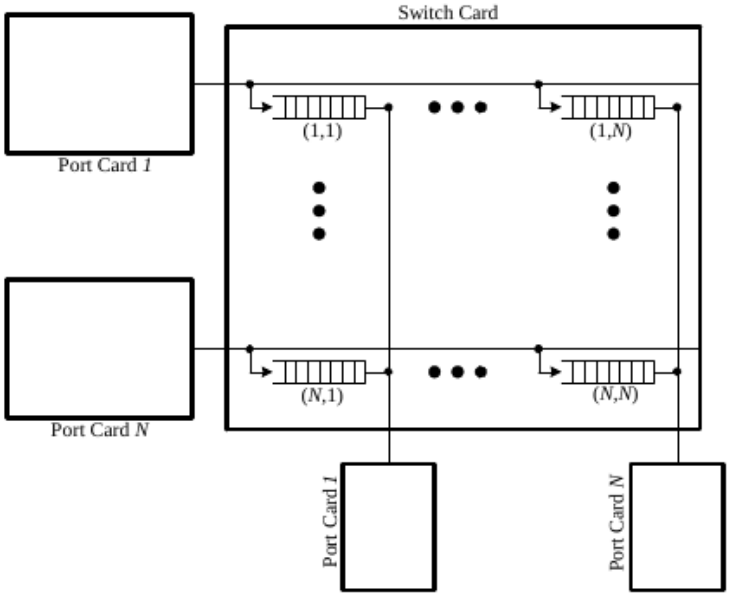


Figure 4: Cross Point Switch Node

5 Detailed Description of Input Module of the Switch

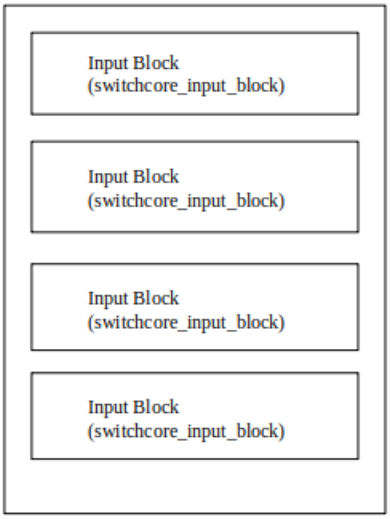


Figure 5: Input Module

The input module (shown in Fig 5) comprises of four individual blocks for each input port. Describing the functionality of one individual block explains the functionality of the entire input module as it is implemented in structural model thereby allowing reuse of the block for each port. Each block comprises of the following components:

1. FCS block
2. rxctrl_delay to generate start of frame and end of frame
3. A FIFO to store the arrived data frame – sync_fifo
4. Packet length counter to count the length of the arrived data frame
5. A memory block to store packet length – pkt_length_fifo
6. Address fifo wrapper block that fetches and stores the SRC and DEST address from the arrived data frame
7. A state machine to interact with the MAC Learning module and send data to output module

The depth of the FIFO in each of the above cases is always the required data size + one additional bit to store end-of-frame. This distinguishes between the current frame and the next frame. Fig 6 shows the various components of the individual block and how they interact with each other.

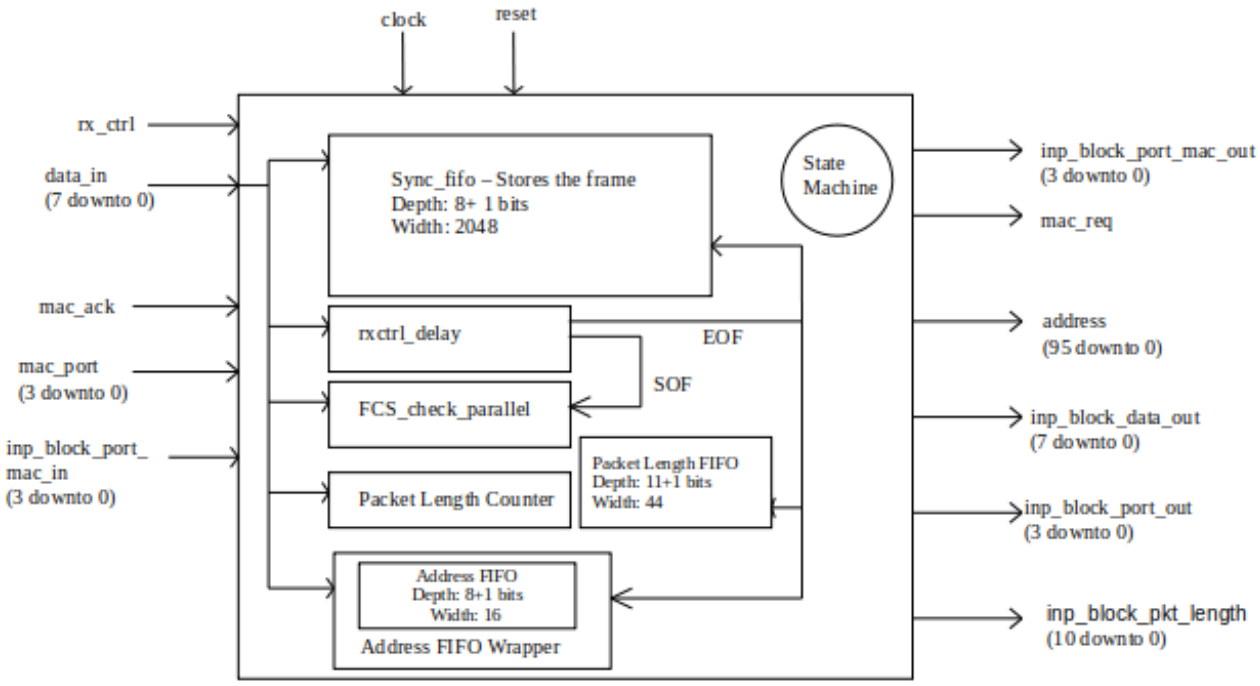


Figure 6: Input Block Components

When the data frame arrives into the individual block, the following processes are executed parallelly:

- Error detection using the FCS block
- Storing the frame in FIFO
- Counting the packet length and storing it in the dedicated memory block
- Identifying and storing the SRC and DEST addresses

5.a FCS Block

The FCS determines whether the received frame is corrupted during transmission. If any error is detected, the frame is discarded. In this design FCS is performed using parallel implementation of CRC-32 wherein the block receives 8-bits as input. The received byte is stored and shifted using a 32-bit linear feedback shift register (LFSR). The XOR operations executing the polynomial division in CRC-32 are obtained by generating a special matrix in matlab. CRC-32 has been implemented in this assignment with generator polynomial, $g(x) = x^32 + x^26 + x^23 + x^22 + x^16 + x^12 + x^11 + x^10 + x^8 + x^7 + x^4 + x^2 + x + 1$. According to the ethernet standard, both the first 32 bits and last 32 bits of the data need to be complemented. However the last 32 bits are not complemented as the end of frame signal generated by the rxctrl_delay block is delayed by one clock cycle. Instead the error-check condition in

the LFSR is changed to all 32 bits being ones from zeros.
Four signals namely *reg*, *shift_count*, *data* and *is_fcs_done* are declared and used. *reg* is a 32 downto 0 std_logic_vector used to implement a linear feedback shift register (LFSR) that shifts the inputs it receives as well as stores the final remainder. *shift_count* is used to count the number of shifts in the LFSR which in this case is $\frac{Total sizeof LFSR}{Numberofinputdatabits} = \frac{32}{8} = 4$. *data* is of std_logic_vector (7 downto 0) type because data 8 bits are passed into this entity in a single clock cycle. *is_fcs_done* is used to determine whether the fcs operation has been completed or not. Finally, the data is error free if all the 32 bits in *reg* is '1'.

5.b Delay Block

The rxctrl_delay block generates the start-of-frame (SOF) and end-of-frame (EOF) signals. The rxctrl signal is delayed by one clock cycle. The SOF signal is generated when *rxctrl* is '1' and *rxctrl_delayed* is '0'. Similarly, the EOF signal is generated when *rxctrl* is '0' and textit$rxctrl_delayed$ is '1'.

5.c Packet Length Counter

As the name suggests, this block counts the number of bytes in the packet which is then stored in a memory element implemented using a FIFO called pkt_length_fifo.

5.d Address FIFO Wrapper

This block fetches the source and destination addresses from the arriving data frame which is then stored in a memory element implemented using a FIFO. The destination address starts from the 9th byte and the source address ends at the 20th byte in the ethernet frame. A counter implemented in this block uses this information to fetch all bytes between 9th and 20th octet (inclusive of both 9 and 20) and subsequently writes to FIFO. It takes two additional clock cycles to read the addresses because counter is triggered every rising edge of the next clock cycle and the address fifo starts to read out one cycle after the read enable is set high.

5.e FIFO

The sync_fifo is used to store the frame. The read signal of this fifo is controlled by the state machine to send the data to the output module.

5.f State Machine

The state machine comprising of seven states controls the interaction of the block with the MAC learning module as well as the output module. Fig 18 shows the various states and the working of the entire state machine. Each of the seven states are described below:

1. ERROR_CHECK - This state is the equivalent of IDLE state in a typical state machine. It waits for the entire frame to be received and for the output from FCS block. If EOF is high the state variable is assigned to DECIDE_NEXT_STATE else the state variable continues to remain in this state.

2. DECIDE_NEXT_STATE - If the fcs_error_check signal from the FCS block is '0' then the state variable is set to GET_ADDR else it is set to DELETE_PKT. Additionally, the request signal to the address fifo wrapper block and the address_counter used in the GET_ADDR state are intitialized to '0' in this state.

3. DELETE_PKT - The read enable of sync_fifo is set high until the packet is removed from the FIFO.

4. GET_ADDR - The address_counter signal is used as it takes 12 + 2 additional clock cycles to read all the 12 bytes from the address fifo due to reasons mentioned in 5.d. The request signal to address fifo wrapper block is set high and the address_counter is incremented by 1. When an acknowledge signal is received from the address fifo wrapper block, the state variable is assigned to SEND_MAC_REQ.

5. SEND_MAC_REQ - The request signal to MAC learning block is set high. Additionally, the source and destination read in the previous state is sent to the MAC learning. The state variable is set to WAIT_FOR_MAC.

6. WAIT_FOR_MAC - When an acknowledge signal is received, the request signal to MAC learning is dropped low and the state is set to SEND_DATA_TO_OUTPUT else the state variable remains in this state.

7. SEND_DATA_TO_OUTPUT - The frame along with the frame length is sent to the output module and state variable remains in this state until frame length clock cycles. Subsequently, state variable is then set back to ERROR_CHECK.

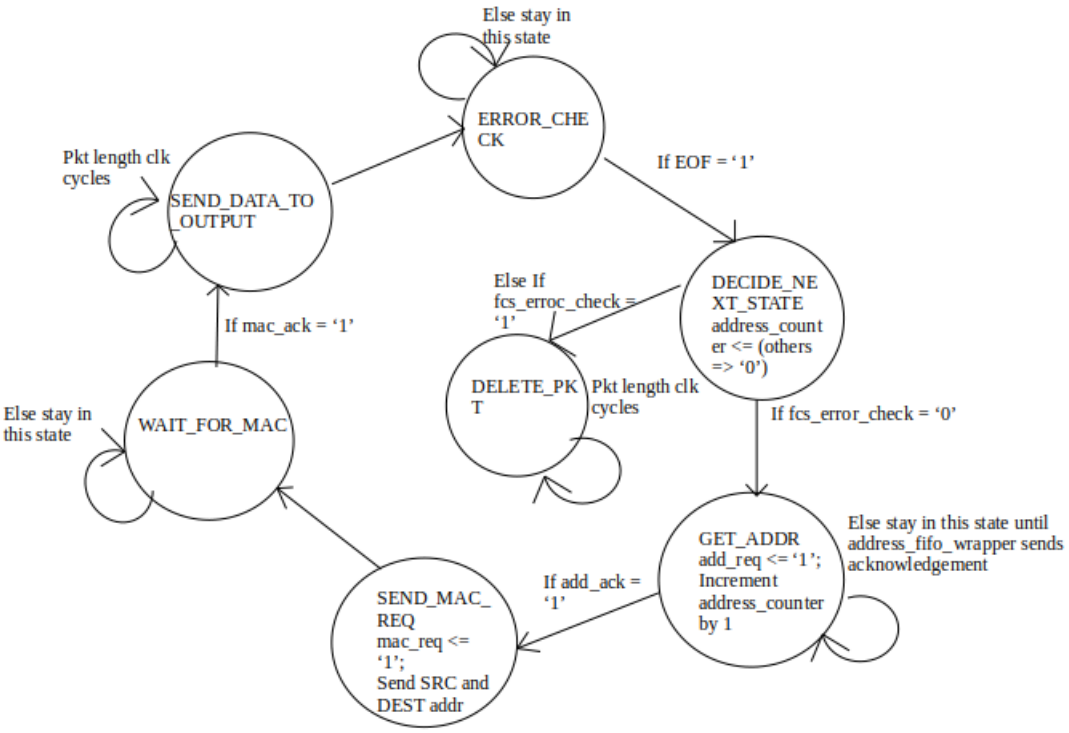


Figure 7: State Machine of Each Individual Block

6 Detailed Description of MAC Learning Module of the Switch

The MAC learning module consists of one block: the MAC learning architecture block which has the CRC16 and the MAC RAM components. Respectively, these two components aim to hash the MAC addresses and register MAC addresses along with their ports.

6.a CRC16 component

The CRC16 component exists in order to compute the hashing as said previously. This is done by using the algorithm CRC16-CCIT, which computes directly the 48-bit data word into an 16-bit output in one clock cycle. After the CRC is computed, the done flag will be 1 and the MAC learning architecture block will proceed with its operation (see section 6.c). It needs to be reset every time a new 48-bit word is sent, since the registers (crcOut - see section 9) will have to be emptied.

6.b MAC RAM component

The MAC RAM component is a file auto generated by Quartus. It is a RAM memory with a capacity of 8000 slots for 52-bit data, filling up the requirements of this project, which is a switch that should be able to handle 8000 MAC addresses.

6.c MAC learning architecture

This module has one state machine that has nine states in total. The diagram is represented below together with the description for each state.

- Check_p1, Check_p2, Check_p3, Check_p4 - Checks if the port has a request, if it has then it enters the hashing_src state. If not, then it checks the following port and keeps checking in loop until one of them receives a request. The acknowledgment and the destination port signals are being reset in this state.
- Hashing_src - It hashes the source MAC address using the CRC16 component. The hashed address must have only 13 bits, therefore, the last bit is discarded. It only proceeds to the next state once the done flag from the CRC16 component is up.
- Hashing_dest - It does the same as the state above, but with the destination MAC address instead.
- Table_processes - This states writes the source MAC address and the source port of the packet in the RAM using the hashed address computed in the hashing_src as the RAM address. It also reads the data written in the hashed destination MAC address as the RAM address. It takes three clock cycles, since it the amount of time that takes to read data from the RAM.

- Compare_MAC - This state compares the destination MAC provided by the packet with the one found in the RAM with the address as this destination MAC, but hashed. If the MAC addresses are the same, then the destination port will be the one stored in the RAM (the last 4-bit of the 52-bit stored), if not, then the packet will be broadcasted, and the destination port will be '1111'.

  This operation is essential in the code, since the 48-bit data is being reduced to a 13-bit data to find the address, which raises a lot the chances of getting collisions. With the absence of this state, if it happens that there is a collision and the destination MAC found in the RAM is not the same as the one provided by the packet, the packet will be sent to the wrong port.

- Send_ack - The last state has the send acknowledgment as its main purpose. It checks which port the packet comes from, and assigns the destination port to the correct signal that represents this port. It also raises the acknowledgment flag to '1' for this port. The acknowledgement and destination port flags are only raised for a duration of one clock cycle.

  The next state is based on which port the last request was sent from, therefore, for instance, if the data computed was received from port 1, then the next state is going to be check_p2, or if the data computed was received from port 2, then the next state is going to be check_p3, and so on.

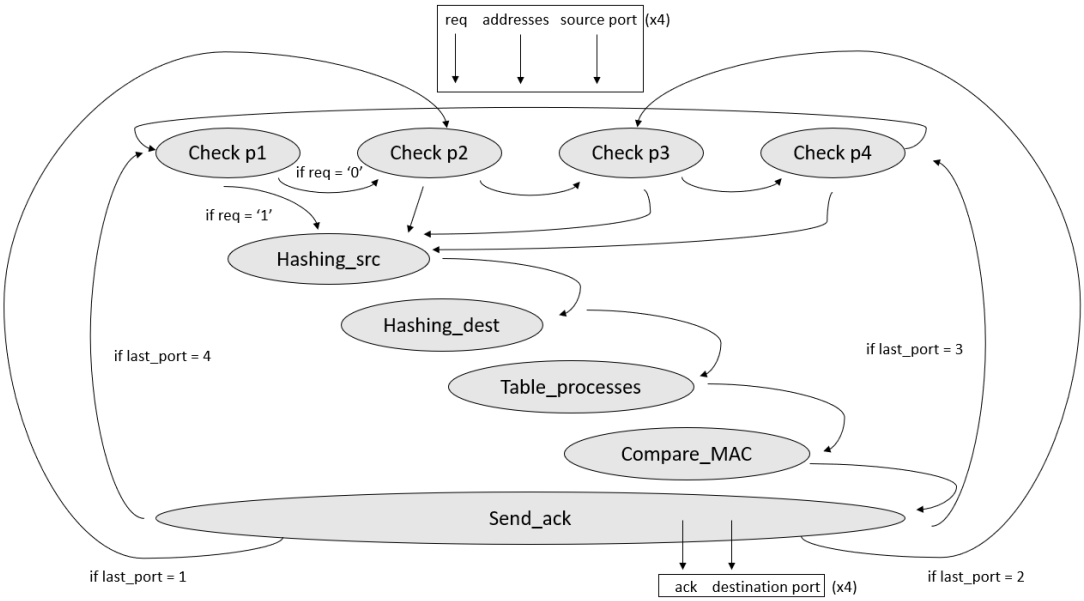


Figure 8: State Machine of the MAC learning block

7 Detailed Description of Output Module of the Switch

Switching and buffering is an important process that takes place in the output of the **Gigabit Ethernet Switch**. To be more exact, in this part of the **Switch** the incoming packets are scheduled to be sent to the corresponding output ports. Moreover, the switching of the packets is based on the **Deficit Round Robin** scheduling algorithm. Further, after a packet is sent out and completed an **interpacket gap** follows the packet. Interpacket gap (IPG) is idle time between packets. After a packet has been sent, transmitters are required to transmit a minimum of 96 bits (12 octets) of idle line state before transmitting the next packet. In our case, the interpacket gap will be a byte of 0s that is sent out for 12 clock cycles.

For a fundamental understanding of the operation of the **Scheduling and Buffering Block**, it is important to state the inputs and outputs of the block, as well as, some of the basic operations of this block. To begin with, in every clock cycle the block receives three types of input from every input port :

- The data of the packets and an extra bit indicating the last byte of the packet(8+1 bits),
- The destination port of the packet (4 bits),
- The packet length (11 bits).

Since there are four input ports the aforementioned data is multiplied by four, as well.

On the other hand, the output of the block in every clock cycle is a **32 bit field(tx_data)**, as well as, a **4 bit field(tx)**. The 32 bit field contains the data for every port, split into four different parts, each of one byte. So, the first byte of the 32 bit field contains data for port 1, the second byte contains data for the port 2 and so on. Furthermore, the 4 bit field refers to the state of the data included in the aforementioned 32 bit field. So, the first bit of the 4 bit field refers to the state of the data coming out from port 1, the second bit refers to the state of the data coming out from port 2 and so on.

Referring to the state of every port, it is important to understand that in some clock cycles the data coming out of the **Gigabit Ethernet Switch** is in an idle state meaning that some bytes of the 32 bit field do not contain information relevant to any packet but serve an important role in the trasmission . Such example is the interpacket gap between packets. In addition, in some cases one of the four output ports might not have any packet to send out, so the data coming out of this port does not come from a packet and consequently the transmitted bytes does not include information to be regarded as packet data. In order to be able to differentiate between the port idle state and the port actually transmitting packet data, every bit of the **tx** can take two values.

- '1' corresponding to packet data coming out of the port
- '0' corresponding to idle data coming out of the port.

7.a Data Destributers

7.a.1 Data Demultiplexers

For all the aforementioned processes to take place, the inputs of the block have to be sorted out and sent to the assigned output port of the block. In this part of the block the switching process takes place. The switch chosen for this design is a cross-point queuing switch as it is easy to implement since the packet arrives at the corresponding cross-point based on the port received from MAC learning. In order to be able to achieve this task data distributors such as Demultiplexers have to be implemented.

The inputs of these demultiplexers are the **packet data (plus one extra bit indicating the last byte of the packet)** and the **destination port**. The destination port field has a number of possible values it can take and these are presented below.

Table 1: Data Distribution

Destination Port Vector Values	Port Destination
0001	Port_1
0010	Port_2
0100	Port_3
1000	Port_4
1111	Broadcast

Consequently, the port destination field is used as the select signal in the demultiplexers which transfers the data to the corresponding output port. However, the above table is the general idea of how the data has to be distributed. In reality, every demultiplexer belonging to a specific input port has different configuration of the table. For example, the case "0100" does not exist for the input port 3. As a result, it is evident that for every input port a custom demultiplexer is created. Below the operation table of every demultiplexer of the block is presented.

Table 2: Demultiplexer 1

Destination Port Values	Port Destination
0010	Port_2
0100	Port_3
1000	Port_4
1111	Broadcast

Table 3: Demultiplexer 2

Destination Port Values	Port Destination
0001	Port_1
0100	Port_3
1000	Port_4
1111	Broadcast

Table 4: Demultiplexer 3

Destination Port Values	Port Destination
0001	Port_1
0010	Port_2
1000	Port_4
1111	Broadcast

Table 5: Demultiplexer 4

Destination Port Values	Port Destination
0001	Port_1
0010	Port_2
0100	Port_3
1111	Broadcast

7.a.2 Length Demultiplexers

The inputs of this block are port destination and length. As a result, the same process described before takes place for the packet lengths as well. Nevertheless, these demultiplexers differ from the data demultiplexers. The main difference is the input and output bit field. In the data case, the input is 9 bits that transfer an 8 bit as an output, in the length case the input is 11 bits that are being distributed whole in the corresponding FIFO.

7.b FIFO Buffers

7.b.1 Data FIFOs

After the packet data and the packet length have been sorted out to their respective ports, they have to be stored in FIFOs first. Due to the complexity of the design three FIFOs have been assigned to every output port. In each output port there can be three packets arriving from the input ports. For example, output port 1 can have data sent in from port 2, port 3 and port 4. As a consequence, each port is assigned with three FIFOs. For a better understanding each FIFO has been named with a specific name that corresponds to the port of destination and the port of origin. The table below illustrates every FIFO assigned to store the packet data.

Table 6: Data Fifos

Data Origin	Data Destination	FIFO Name
Port_2	Port_1	FIFO 1.2
Port_3	Port_1	FIFO 1.3
Port_4	Port_1	FIFO 1.4
Port_1	Port_2	FIFO 2.1
Port_3	Port_2	FIFO 2.3
Port_4	Port_2	FIFO 2.4
Port_1	Port_3	FIFO 3.1
Port_2	Port_3	FIFO 3.2
Port_4	Port_3	FIFO 3.4
Port_1	Port_4	FIFO 4.1
Port_2	Port_4	FIFO 4.2
Port_3	Port_4	FIFO 4.3

From the table above it is profound that twelve FIFOs have been used for storing the data of every packet.

7.b.2 Length FIFOs

Equivalently, the same FIFOs configuration described in the above-mentioned section takes place for the length of the packets. The most important difference between the fifos for data and the FIFOs for length is that the length FIFOs have a **fifo length** of 11 bits. The table below illustrates every FIFO assigned to store the packet length.

Table 7: Data Length Fifos

Data Origin	Data Destination	FIFO Name
Port_2	Port_1	FIFO 1.2_Length
Port_3	Port_1	FIFO 1.3_Length
Port_4	Port_1	FIFO 1.4_Length
Port_1	Port_2	FIFO 2.1_Length
Port_3	Port_2	FIFO 2.3_Length
Port_4	Port_2	FIFO 2.4_Length
Port_1	Port_3	FIFO 3.1_Length
Port_2	Port_3	FIFO 3.2_Length
Port_4	Port_3	FIFO 3.4_Length
Port_1	Port_4	FIFO 4.1_Length
Port_2	Port_4	FIFO 4.2_Length
Port_3	Port_4	FIFO 4.3_Length

7.c Scheduling Implementation

7.c.1 Deficit Round Robin

Deficit Round Robin is a variation of Weighted Round Robin (WRR) that allows flows with variable packet lengths to share the link bandwidth fairly. Each flow is characterized by a quantum of **Q** bits, which measures the quantity of packets that flow should ideally

transmit during a round, and by a deficit variable **Deficit Counter** . When a backlogged flow is serviced, a burst of packets is allowed to be transmitted of an overall length not exceeding **Q + Deficit Counter** . When a flow is not able to send a packet in a round because the packet is too large, the number of bits which could not be used for transmission in the current round is saved into the flow's deficit variable, and are therefore made available to the same flow in the next round.

More specifically, the **Deficit Counter** is managed as follows

- Reset to zero when the flow is not backlogged,
- Increased by Q when the flow is selected for service during a round,
- Decreased by the packet length when a packet is transmitted.

For the design of the **Gigabit Ethernet Switch** the Deficit Round Robin was implemented. The DRR algorithm is used to schedule the packet transmission from every port. Since, there are 3 FIFOs assigned for every port, the DRR algorithm is used to provide fair queuing between the packets inside these FIFOs. Moreover, for the design of the **Gigabit Ethernet Switch** the quantum value Q is set at 1550 which represents the maximum amount of bytes an Ethernet packet can have and since the outputs are one byte per clock cycle the **Deficit Counter** increments per byte . Consequently, in one round of the DRR an Ethernet packet with the maximum length can be served for each one of the FIFOs. Below a flowchart of the DRR algorithm for the **Gigabit Ethernet Switch** is presented. It has to be noted that the following example corresponds to one of the four output ports of the block and in our case it is port 1. As a result, the FIFOs used in this example are **FIFO_1.2**, **FIFO_1.3** and **FIFO_1.4**.

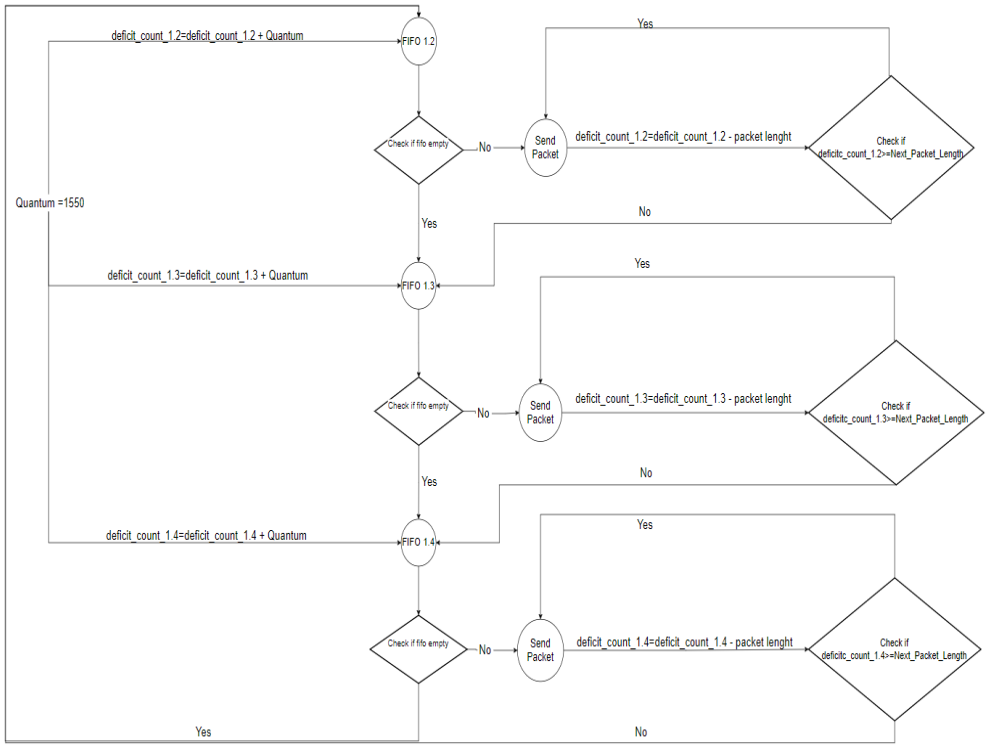


Figure 9: Deficit Round Robin Scheduling for port 1

7.c.2 State Machine

7.c.3 Theory of the State Machine Implementation

In the previous section a brief explanation of the Scheduling process was introduced. However, when implementing this scheduling algorithm to the **Switch and Output Buffering Block** the complexity of the process increases greatly. First of all, in order to implement this logic to Field Programmable Gate Arrays (FPGAs) a state machine has to be created. A state machine is a digital device that traverses through a predetermined sequence of states in an orderly fashion. A state is a set of values measured at different parts of the circuit. More specifically, in this case these values are the **Deficit Counter** set for every FIFO of the output port and the states are the FIFOs themselves. As a consequence, the state machine changes states based on the deficit counter value of every FIFO. In principle, the statement that decides in which state (FIFO) the state machine should stay is **If deficit_counter>=Packet_Length**. Since there are three FIFOs assigned for every output port there will be four state machines trans versing between three different states that correspond to the FIFOs.

It is evident now that the complexity of the Deficit Round Robin scheduling process has increased. In addition, as mentioned previously in the idle state the output of the **Switch** ought to be a byte of 0s and the **tx** should be 0. However, the state machines presented here do not have that capability. They can provide a **tx** flag with the value of '0'. Another state machine that controls the output of the four state machines will provide a byte of 0s in that

case. These state machine will be named **Control State Machine** and will be presented in detail later on the report.

Additionally, some specifications of these state machines have to be made. First of all, it was mentioned that the state machine will have three states corresponding to the FIFOs of every port. In reality, an additional forth state was created which will be named as **State_S0**. This state represents the case when every data and length FIFO is empty. By implementing this state in the DDR scheduling algorithm, the state machine needs less clock cycles to identify that state. And as a consequence there is no delay in the first clock cycles that the **Switch** is starting to operate. In other words, for the first clock cycles that the input needs to perform the FCS check of the incoming packets, and after that the MAC learning takes place the Data and Length FIFOs are empty. As a result, the output of the **Switch** ought to be a 32 bit field completed with 0s (**tx_data**) and a 4 bit field completed with all 0s (**tx**), as well .Which will happen immediately for the reason that the **State_S0** will be the starting point of the DRR algorithm and as a result the **Switch** will have output the same time as input is being inserted.

Below a flowchart of the operations happening inside the **State_S0** is presented.

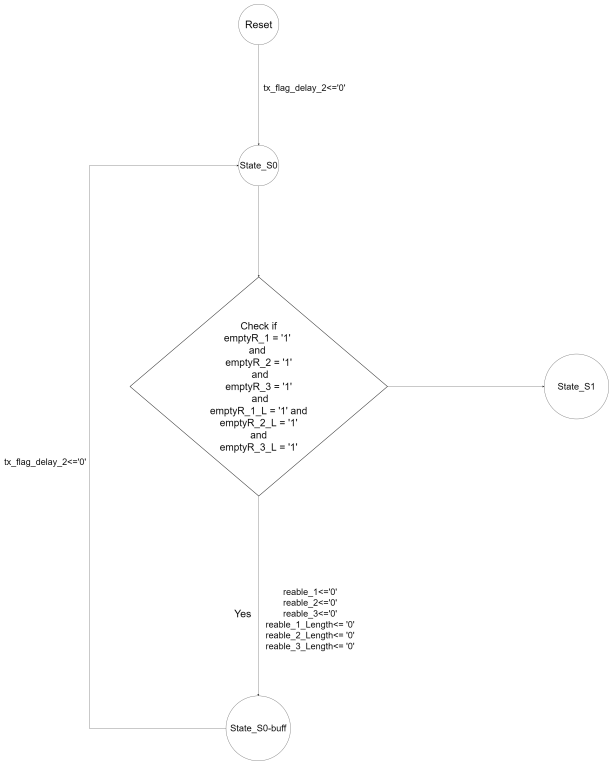


Figure 10: State Machine Port 1

From the flow chart above, it is profound that there is only one logical statement taking place. Nonetheless, the **State_S0** consists of another sub-state named **State_S0_buff**. This state was created for the reason that the data needs two clock cycles to be transferred from the FIFO to the output of the state machine. As a result we had to create this sub-state to stall

the idle data for one clock cycle. However, it could be logical to create two sub-states for that process, nevertheless the aforementioned **State Machine Controller** needs one more clock cycle to implement a byte of 0s. And as a consequence, there is no delay between the packet data and the idle data. On top of that, we need to delay the tx flag of that state machine for two clock cycles, as well. That is the reason why the **tx_flag_delay_2** has be named that way.

Moving on, when the FIFOs are not empty but contain data the state machine moves to the next state. In our case that is **State_S1**. Based on the previous example of DRR , **State_S1** corresponds to FIFO 1.2, in the same way **State_S2** corresponds to FIFO 1.3 and **State_S3** to FIFO 1.4. In that state a number of checks and processes take place. The figure below illustrates these checks and processes.

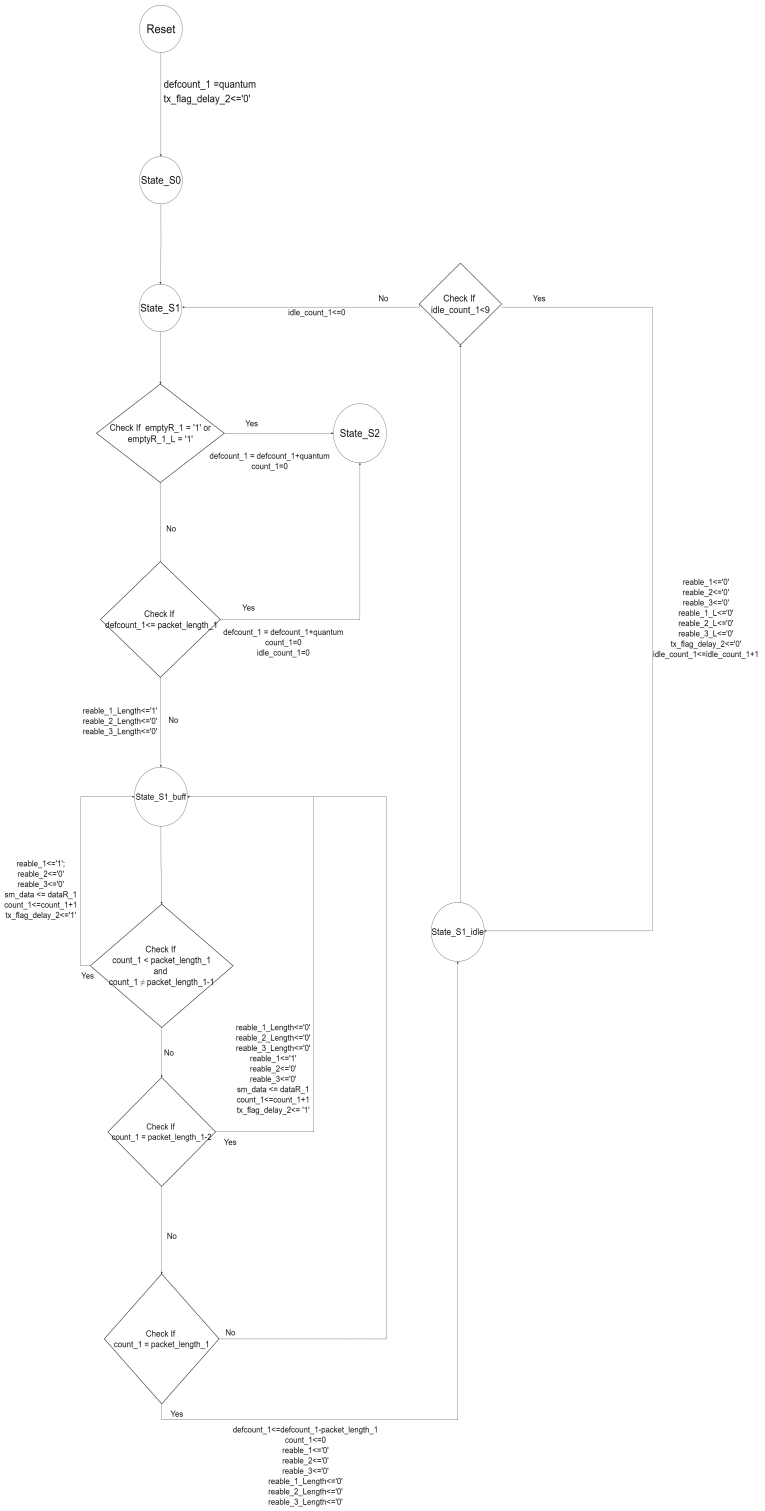


Figure 11: State Machine Port 1

This figure is a little bit confusing, so for a better understanding of the reader a brief explanation of the processes taking place inside the **State_S1** is presented.

1. First, the state machine moves to **State_S1** from **State_S0**.
2. A check on the FIFO capacity is performed and if the FIFO is empty the state machine moves to the next state **State_S2**(next FIFO).
3. Since the FIFO contains both packet and packet length data the state machine moves to the next check. This check is the **If deficit_counter>=Packet_Length** check in which the actual DRR check is performed to determine if this FIFO can serve one more packet. In the first run of the state, there is no packet length data acquired yet so the packet length is 0 as predefined from the reset button while the deficit quantum value is 1550 as s predefined from the reset button in the same way. So this statement is true and the state machine moves to the next sub state which is **State_S1_buff**. But before that transverse the **Read_Length_Enable** of the first FIFO is raised to '1' so that the state machine can acquire length data for the packet.
4. In the **State_S1_buff** sub state the first check is performed.
5. If the case **If count_1<(Packet_Length_1) and count_1≠(Packet_Length_1)-1** is true than that means the state machine can start to read out data from the FIFO and transfer it to the next entity. In addition, a number of process take place if that statement is true. The corresponding **Read_Enable** flag is raised to '1' so that the state machine can read out data from the FIFO. Moreover, the **tx** flag is raised to '1' indicating that the state machine is providing packet data as an output. It is crucial to mention the counter that is presented in the if case. This counter named **counter_1** increments per clock cycle when data is being transferred from the FIFO. In other words, this counter counts the number of bytes that are being transferred from the state machine to the output. As a consequence it is essential to count the number of bytes being transferred to the output and compare it to the length of the packet, so that it is guaranteed that the right number of bytes will be read out of the FIFO. Moreover, since both the length of the packet is in bytes and the output of the state machine is bytes per clock cycles, the counter increments per byte every clock cycle.
6. In the previous case there were two sub cases, both the counter had to be less that the packet length but also different from the length packet minus 1. As we proceed to the next case there is a specific statement checking if the the counter is equal to the packet length minus 1. This case exists for the solely reason that the state machine has already read out the packet length once before starting to read out data from the Data FIFO. Consequently, in order not to read out the packet length value for the next packet that might be stored in the Length FIFO, the **Read_Length_Enable** flag has to be dropped to '0' one clock cycle earlier from the **Read_Enable** flag of the data. Nevertheless, the rest of the processes remains the same with the previous case.
7. Following, the next case is if the aforementioned counter 1 equals to the packet length. In other words, if this case is true then the whole packet has been read out from

the FIFO. As a result, the state machine moves to the next sub state which is the **State_S1_idle**. But, before moving on to the next state every read flag for both of the FIFOs in switched to '0', the counter as well resets to 0, the **tx** flag also drops indicating the end of the packet and the value of the deficit counter is being updated to a new value. This new value is the previous deficit counter value minus the packet length of the packet that was just served.

8. **State_S1_idle** represents the interpacket gap (IPG) state. In this state, the state machine must remain in the idle state for a minimum of 12 clock cycles. To facilitate this idle state for the wanted amount of cycles another counter is implemented(**idle_count_1**). This counter increments up until 9 before allowing the state machine to move to the next state which in reality is again **State_S1**. In theory the idle counter should increment until 12. However, due to the complexity of the system three extra clock cycles are needed for the state machine to switch from **State_S1_buff** to **State_S1_idle** and then to **State_S1**. So in total there are 12 clock cycles, where the **tx** flag is '0' and nothing is being read out from the FIFOs.

9. Once again in the **State_S1** the same comparisons are being performed with the only difference being the value of the deficit counter decreased by the served packet length. If the deficit counter is great enough to serve the next packet than the same process is performed. If not, then the state machine moves to the next state **State_S2**(FIFO 1.3).

In general, the logic behind this implementation is to create loops. Inside these loops different processes are taking place either to read out packet data or to stay in the idle state for a number of clock cycles. The only way to break out of these loops is when the counters inside these loops have reached a certain number allowing the state machine to move to next states. Nevertheless, inside these loops there are processes that control the reading procedure from FIFOs, but also giving out correct indicators about the content of the data (tx flag). In order to make this processes synchronized sub states are to be made.

7.d State Machine Controller

As stated previously, the four aforementioned state machines are connected to an other entity called **State Machine Controller**. The inputs of this entity are the outputs of the state machines. To be more exact: The operations performed in this entity are simple. To begin with, each **tx_flag** value is checked. The check is related to whether the **tx_flag** value is '1' or '0'. In case of an '1', the data provided from the corresponding input is passed to the output of the **State Machine Controller**. On the contrary, when the **tx_flag** value is '0' the controller creates an 8 bit vector filled with 0s, and passes it to the output. This process of creating the final data values is similar the process of the state machine passing data to this entity. As a consequence, there is a delay between the data output and the flag indicator. In other words, there is delay between the **tx_data** and **tx** signals. For that reason, the four

flag input signals are delayed for two clock cycles each. The logic behind the control state machine is depicted in the flow chart below.

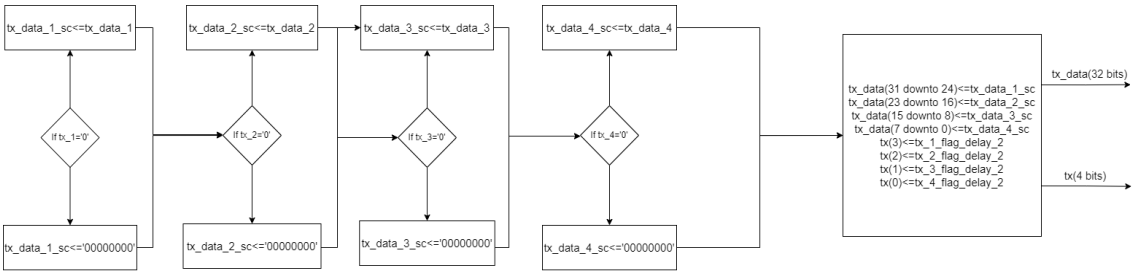


Figure 12: State Machine Controller

From the figure above, the final check process is depicted. All the incoming flag values are being checked every clock cycle and depending on their value, the output value is formed.

8 Detailed Specification of Input Module of the Switch

The I/O ports of an individual block is shown in Fig 13. The size of the FIFO inside the block used to store the frame is 8+1 $\times$ 2048 which is at least twice the size of Ethernet frame (1550 bytes). This ensures that the next frame is not missed by the switch. The I/O ports of the block are described below:

- clock
- reset
- data_in (7 downto 0) - receives input data
- mac_ack - receives acknowledge signal from MAC learning module
- mac_port (3 downto 0) - the destination port received from MAC learning module
- inp_block_port_mac_in (3 downto 0) - port corresponding to the individual block
- inp_block_port_mac_out (3 downto 0) - destination port received from MAC learning to be sent to output module
- mac_req - request signal sent to MAC learning module
- address (95 downto 0) - source and destination addresses sent to MAC learning module
- inp_block_port_out (3 downto 0) - source port corresponding to the individual block to be sent to MAC learning module
- inp_block_data_out (7 downto 0) - data to be sent to output module
- inp_block_pkt_length (10 downto 0) - frame length to be sent to output module

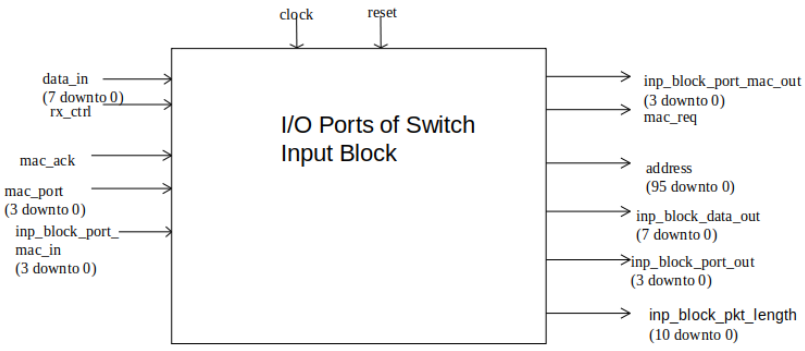


Figure 13: Individual Block I/O Ports

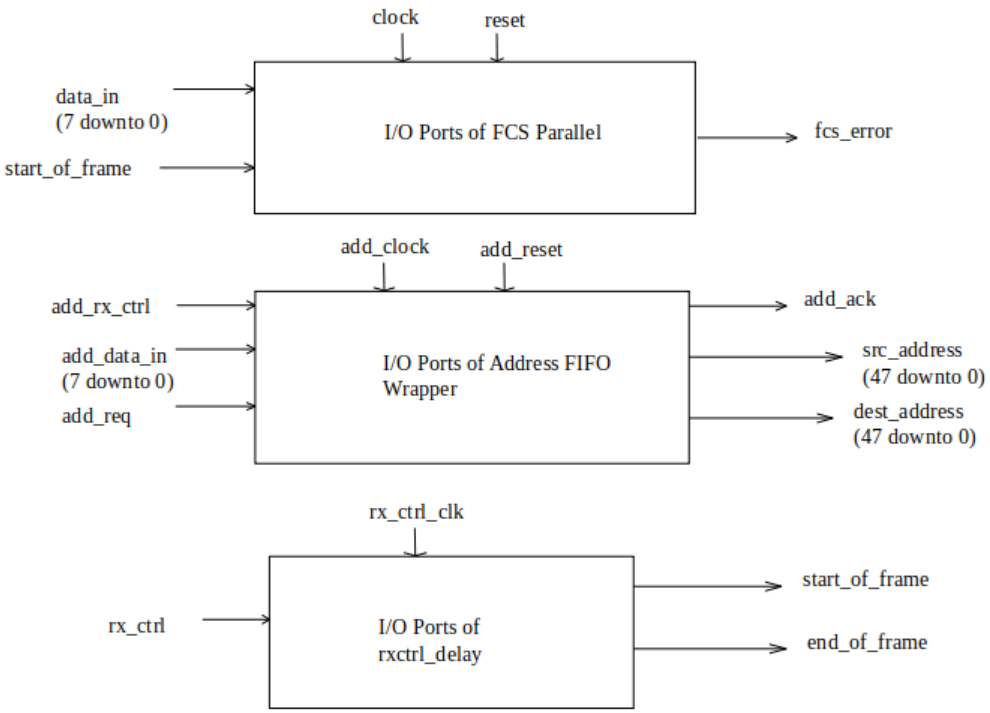


Figure 14: Individual Block Components I/O Ports

The I/O ports of the FCS block (shown in Fig 14) are mentioned below:

- clock
- reset
- data_in (7 downto 0) - receives input data for FCS
- start_of_frame - SOF signal required by the block to complement the first 32 bits (refer section 5.a)
- fcs_error - output signal indicating whether the frame is error-free or not. The frame is error free when fcs_error is '0'

The address fifo consists of a fifo of size $8 + 1 \times 16$ bits where the additional bit stores the EOF. As mentioned earlier, storing the EOF helps in separating the current frame from the next frame. The I/O ports of the address fifo wrapper block (shown in Fig 14) are mentioned below:

- add_clock
- add_reset
- add_rx_ctrl - control signal is high as long as the block receives data

- add_data_in (7 downto 0) - receives input data from which source and destination addresses are retrieved
- add_req - input request signal to read source and destination address from address fifo.
- add_ack - output acknowledge signal to indicate that the reading operation from fifo is complete
- src_address (47 downto 0) - source address read from fifo to be sent to MAC learning module
- dest_address (47 downto 0) - destination address read from fifo to be sent to MAC learning module

The I/O ports of the rxctrl_delay block (shown in Fig 14) are mentioned below:

- rx_ctrl_clk
- rx_ctrl - the control signal to be delayed
- start_of_frame - generated SOF signal
- end_of_frame - generated EOF signal to be stored in sync_fifo

Finally, the I/O ports of input module is shown in Fig 15.

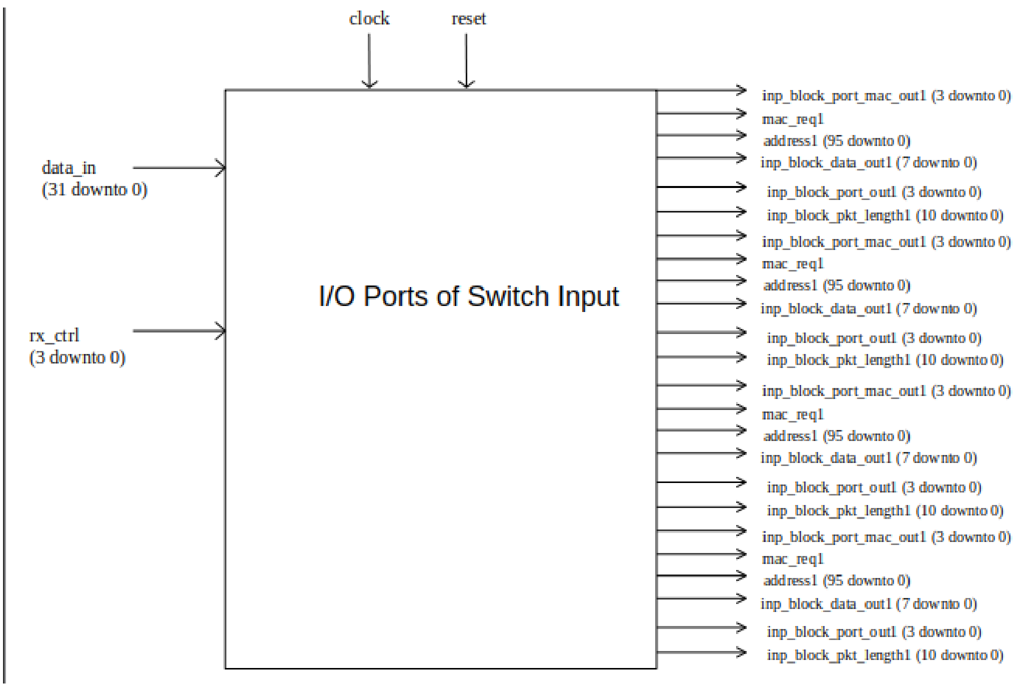


Figure 15: I/O ports of switchcore_input

9 Detailed Specification of MAC Learning Module of the Switch

The I/O ports for the MAC learning module are:

- clock
- reset

The following ports are unique for each port. That means there are 20 (5 x 4) ports.

- addresses (95 downto 0) - receives both source and destination addresses of the packet from the input module
- s_port (3 downto 0) - source port corresponding to the addresses from the input module
- req - receives request signal from the input module
- ack - sends acknowledge to the input module from a respective port
- d_port (3 downto 0) - the destination port of the packet

The image below shows a representation of the I/O ports in the main file of this module.

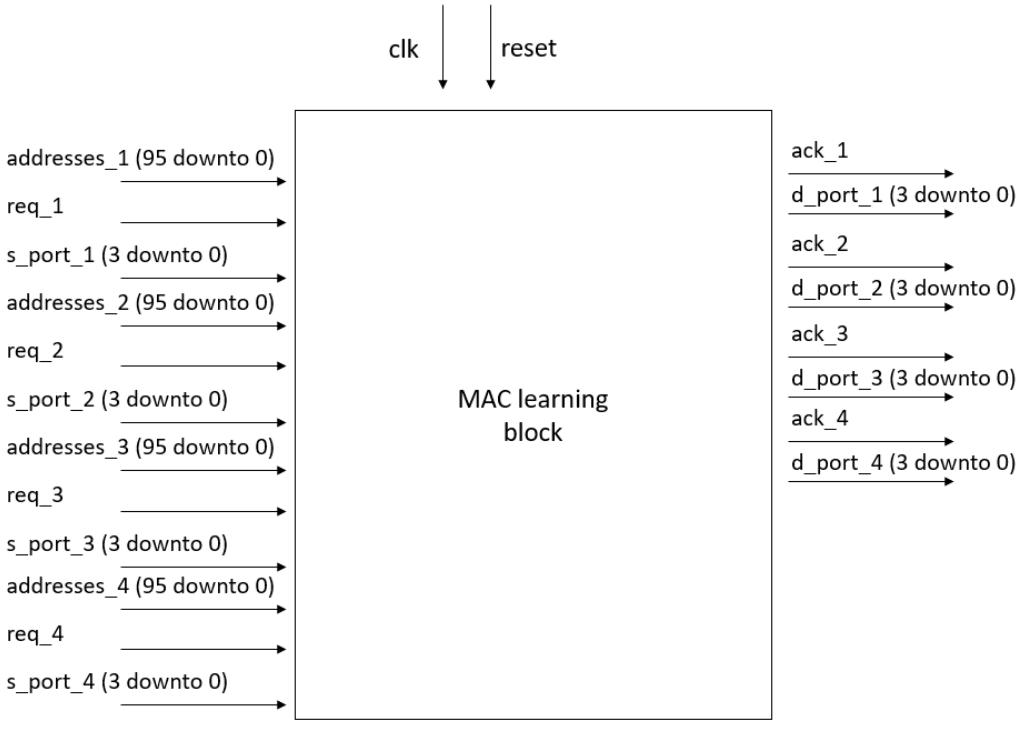


Figure 16: I/O ports for the MAC learning block

For the hashing block, these are the I/O ports:

- clock
- reset
- dataIn (47 downto 0) - receives the data to be hashed
- done - sends acknowledgment once the hashing is done
- crcOut (15 downto 0) - sends the hashed data

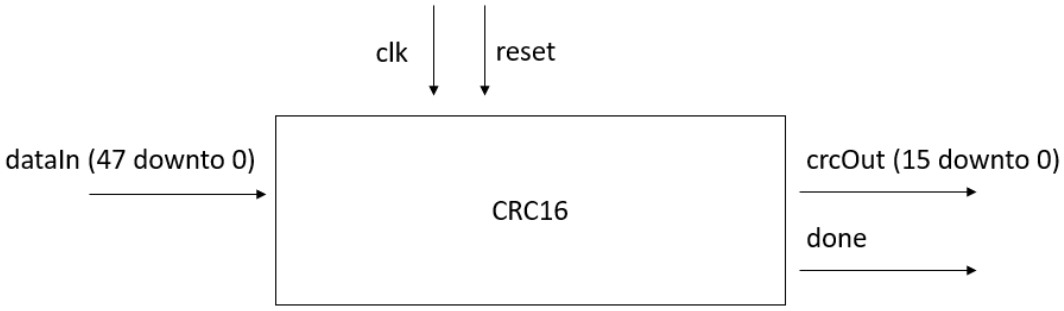


Figure 17: I/O ports for the CRC16 block

Finally, the I/O ports for the RAM block are:

- clock
- reset
- data (51 downto 0) - data to be written
- rdaddress (12 downto 0) - address to read the data from
- wraddress (12 downto 0) - address to write the data to
- wren - enable writing in the RAM
- q (51 downto 0) - data read

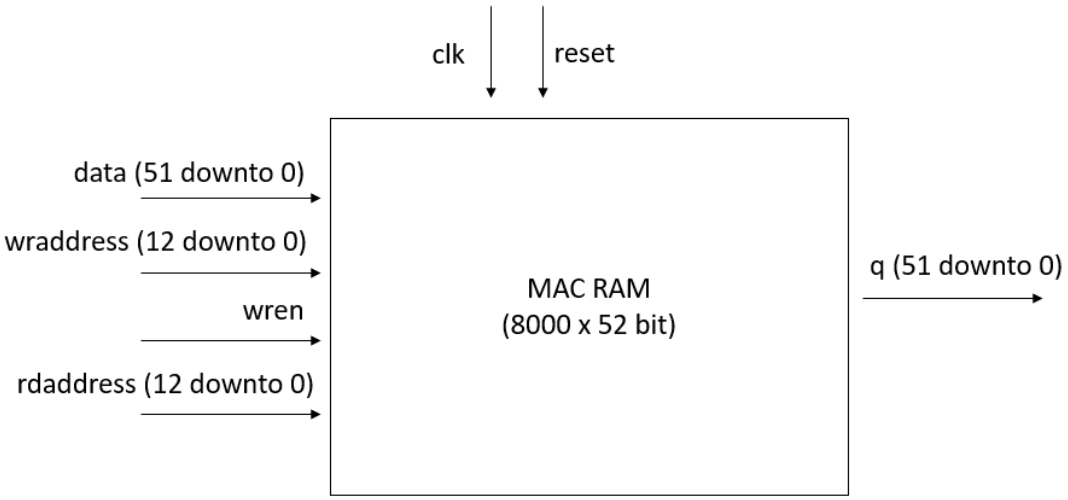


Figure 18: I/O ports for the RAM

10 Detailed Specification of Output Module of the Switch

The detailed description of every component of **Output Module of the Switch** has been made. For a brief summary of the aforementioned entities, the figure below illustrates all the components inside the block.

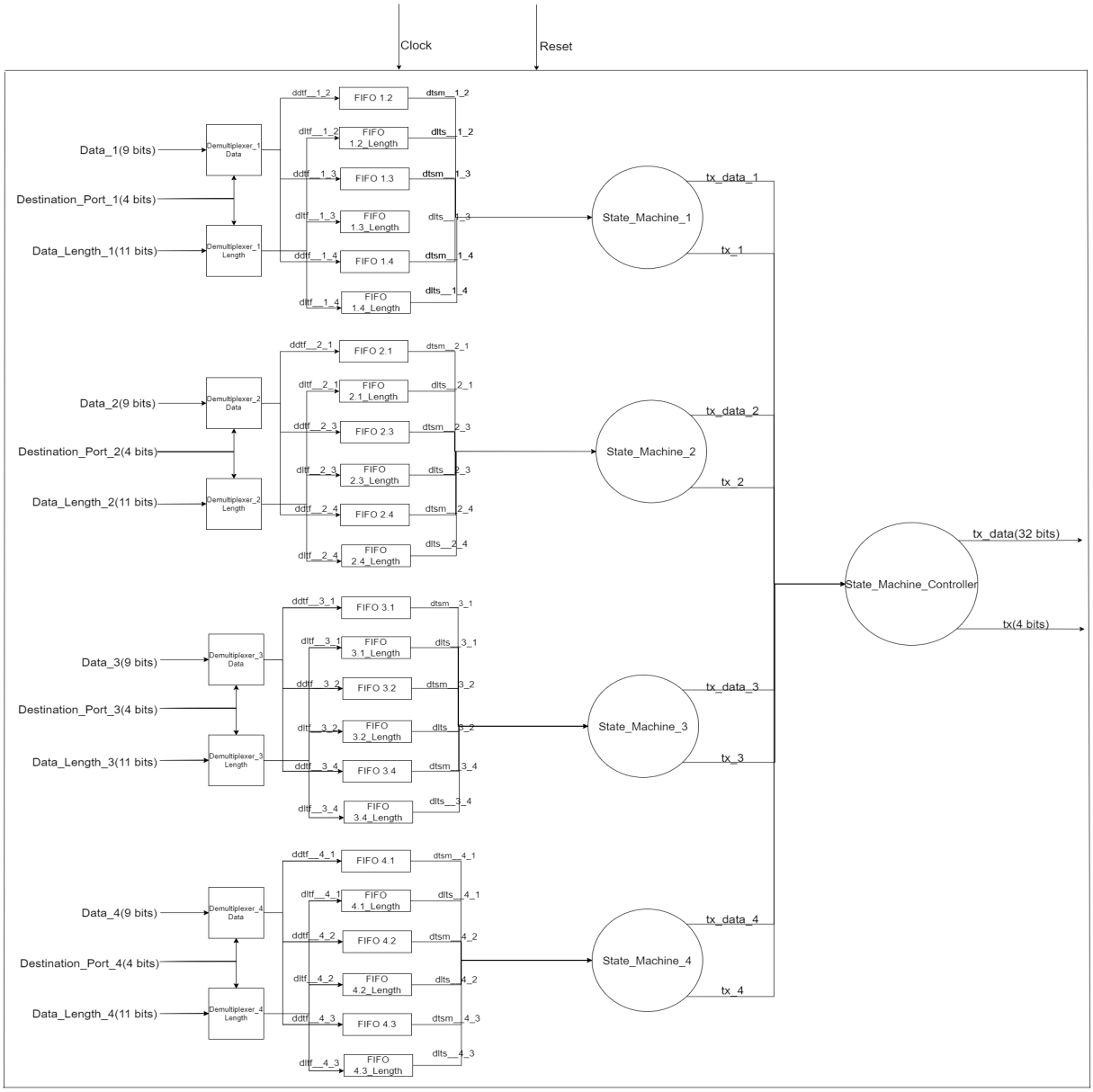


Figure 19: Switch and Output Buffering Block

In this part of the report the I/O ports of every entity inside this module is described. In general the I/O of the **output module** are

- clock
- reset

- data_1, data_2, data_3, data_4(8 downto 0)-input packet data from port 1,2,3 and 4
- destination_port_1, destination_port_2, destination_port_2 destination_port_2(3 downto 0)-input packet destination port from port 1, 2, 3 and 4
- data_length_1, data_length_2, data_length_3, data_length_4(10 downto 0)-input packet length from port 1,2 3 and 4
- tx_data(31 downto 0)-output data of the module
- tx(3 downto 0)-output flag of the module

More specifically, the I/O ports of the **Data Demultiplexers** are:

- data_in_demux(8 downto 0)-input packet data from the corresponding port
- sel_demux(3 downto 0)- input destination port data from the corresponding port
- F_1, F_2 , F_3(7 downto 0)- output packet data directed to the corresponding FIFO.
- R_1, R_2 , R_3-output write enable signal, allowing data to be written inside the corresponding FIFO.

Moving on, the I/O ports of the **Length Demultiplexers** are:

- data_length_in_demux(10 downto 0)-input packet length for the corresponding packet
- sel_length_demux(3 downto 0)- input destination port data from the corresponding port
- L_1, L_2 , L_3(10 downto 0)- output packet length directed to the corresponding FIFO.
- RL_1, RL_2 , RL_3-output write enable signal, allowing length to be written inside the corresponding FIFO.

Regarding the data FIFOs. The FIFO length is of 8 bit, and the FIFO depth is of 2048 words which corresponds to 4096 bytes. We take into account that the maximum Ethernet packet size is of 1550 bytes. Accordingly, the FIFOs can store at least two packets of maximum size inside of them. The I/O ports of the **Data FIFOs** are

- clock
- data(7 downto 0)-input packet data stored in FIFO
- rdreq
- wrreq
- empty

- full
- q(7 downto 0)-output data from FIFO

Regarding the length FIFOs. The FIFO length is of 11 bit, and the FIFO depth is of 2048 words which corresponds to 4096 bytes. The I/O ports of the **Length FIFOs** are

- clock
- length_data(10 downto 0)-input packet length stored in FIFO
- rdreq
- wrreq
- empty
- full
- q_length (10 downto 0)-output packet length from FIFO

Further, the I/O ports of the **state machine** are

- clock
- reset
- empty R_1, empty R_2, empty R_3 -input empty flag from FIFO in state S_1, S_2 and S_3
- packet_length_1, packet_length_2, packet_length_3(10 downto 0)-input packet length in state S_1, S_2 and S_3
- dataR_1, dataR_2, dataR_3(7 downto 0)-input packet data from FIFO in state S_1, S_2 and S_3
- reable_1, reable_2, reable_3-output read enable signal for FIFO in state S_1, , S_2 and S_3
- tx_flag- output flag indicating data status
- sm_data- output packet data coming out of the state machine

Lastly, the I/O ports of the **state machine controllers** are

- clock
- tx_1, tx_2, tx_3, tx_4-input indicating the data status coming from state machine 1, 2, 3 and 4

- tx_data_1, tx_data_2, tx_data_3, tx_data_4(7 downto 0)-input packet data from state machine 1, 2, 3 and 4.
- tx(3 downto 0)-output tx flag
- tx_data(31 downto 0)-output tx_data

11 Simulation and Verification

Combining the three parts, the switch was tested by sending the exact same packet from every port at the same time. The packet is the following: 0010A47BEA8000123456789008004500002EB3FE000080110540C0A8002CC0A8000404000400001A2DE8000102030405060708090B0C0D0E0F1011E6C53DB2

Since the destination MAC address is not registered in the RAM table, because the only registered MAC address is the source one, then the packet will be broadcast.

When running this simulation the following output is seen in ModelSim:

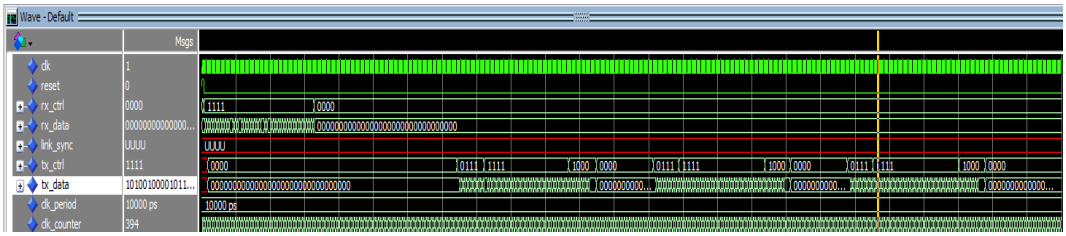


Figure 20: Simulation Results of the 3 main blocks together

In the following sections, different simulations will be provided for each main block:

11.a Simulation of Individual Block of Input Module of the Switch

As shown in Fig 21 and Fig 22, 3 ethernet frames were given as input to the individual block. The module was tested for three cases - (i) error-free frame and dummy acknowledgement signal from MAC module; (ii) a corrupted frame generated by changing one of the bytes in the frame; (iii) an error-free frame without any dummy acknowledgement signal. In case 1, the address signal in Fig 21 provides to the MAC module. When acknowledged and receiving the output port, it subsequently transfers data to the output module. The temp_fcs_error_check signal in Fig 22 remains high in case 2 when the block receives a corrupted frame. In case 3, the mac_req signal in Fig 21 remains high as no acknowledgement is received.

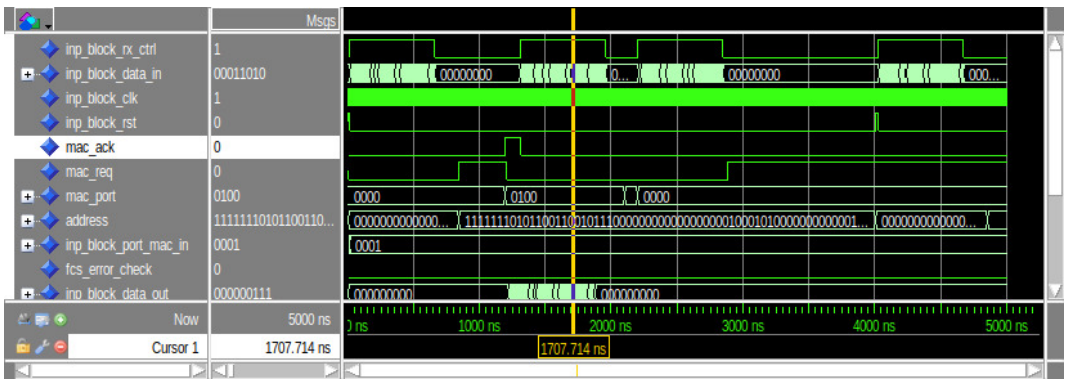


Figure 21: Simulation of Input Block - 1

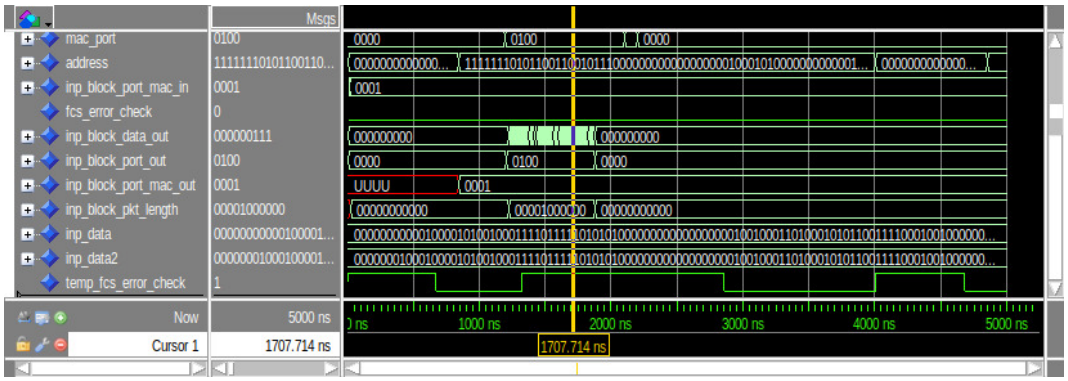


Figure 22: Simulation of Input Block - 2

11.b Simulation of MAC learning Module of the Switch

By testing the MAC learning module individually, it is possible to see that it actually works. If the following destination and source address are sent by port 3 respectively, "00000ABB28FC" and "00000ABB28FD", then the packet is supposed to be broadcast ("1111"). If the next addresses from the next packet has the destination MAC address as the same as the previous packet source address, then it is supposed to reach port 3 ("0100"). This happens considering that the destination and source address sent by port 2 are "00000ABB28FD" and "0011111128FD". The output from ModelSim can be seen below.

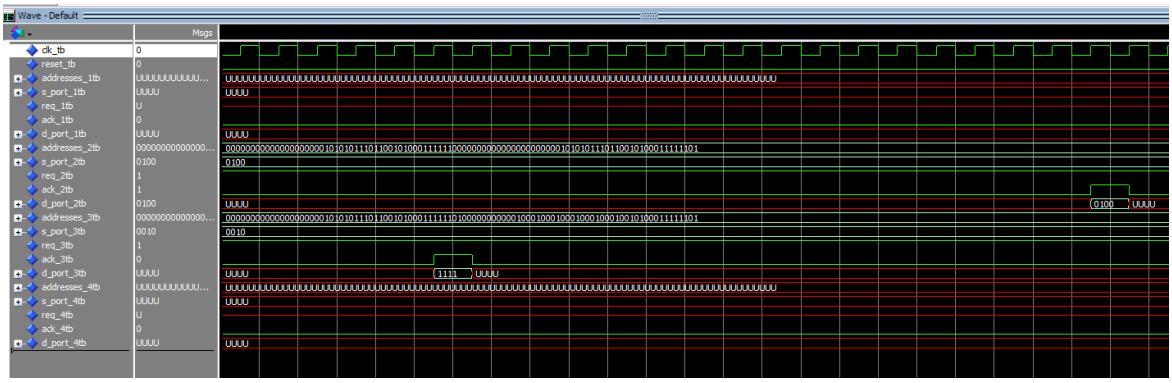


Figure 23: Simulation Results of the 3 main blocks together

11.c Simulation of Output Module of the Switch

11.c.1 Data Demultiplexers

A simulation of the demultiplexing operation for one demultiplexer of this block is showcased. The demultiplexer simulated is Data Demultiplexer 1.

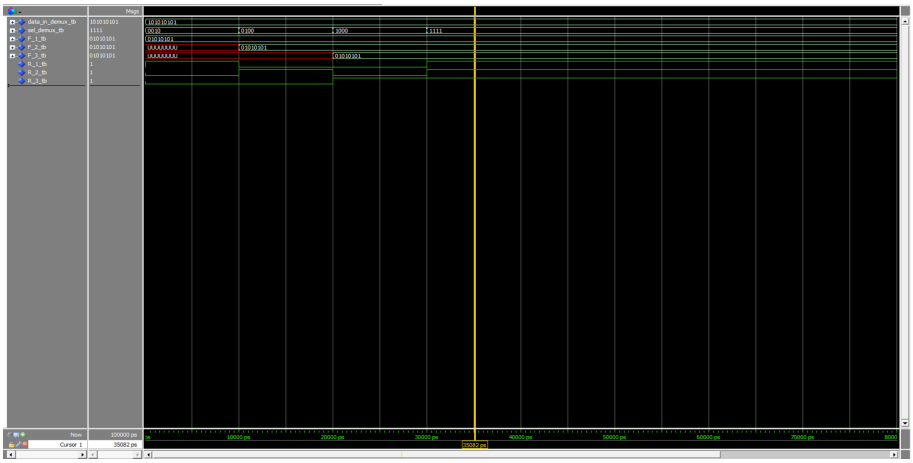


Figure 24: Simulation Results of Data Demultiplexer 1

In the figure above the signals **F_1,F_2,F_3** refer to the data transferred to the fifos, and **R_1,R_2,R_3** to the read enable signals to the fifos.

11.c.2 Length Demultiplexers

Below the simulation of the demultiplexing operation for one demultiplexer of this block is showcased.

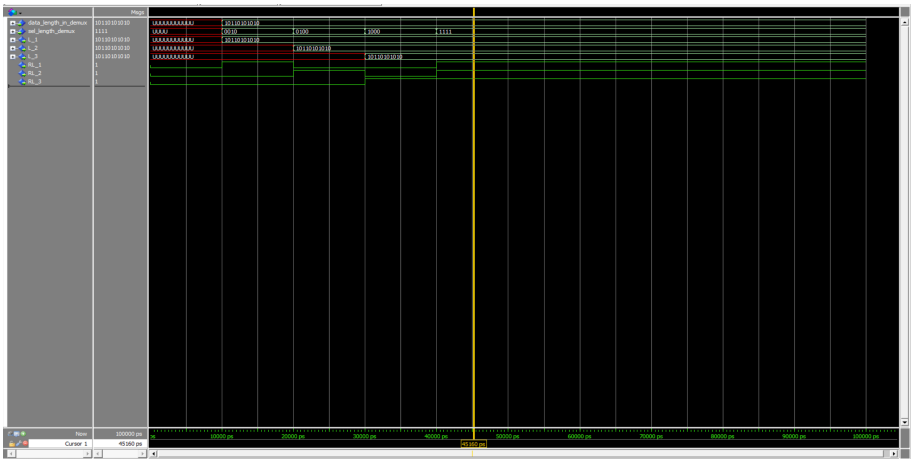


Figure 25: Simulation Results of Length Demultiplexer 1

In the figures above the signals **L_1,L_2,L_3** refer to the length data transferred to the FIFOs, and **RL_1,RL_2,RL_3** to the read enable signals to the FIFOs.

11.c.3 State Machine

A simulation of the state machine is presented. In this simulation, the state machine has 3 packets as inputs that are scheduled ,by the DRR algorithm of the state machine, to be served fairly. Furthermore, since the inputs originate from FIFOs, the empty flag signals are

also inputs of the state machine. In this simulation, the input signals are fixed to imitate the behavior of a FIFO. In more detail, the first packet arrives at the state machine(including packet data, length and empty flag). The packet length is 65 bytes. After the first packet ends, the empty flag is raised to indicate that the FIFO providing the data is empty. The next packet arrives from another input of the state machine and the same process takes place. This handling of the signals is made to test the transverse behavior of the state machine when every FIFO of the input has data. The purpose of this simulation is to test not only the change of states but also to test the output. After each packet ends, the interframe gap has to take place. While every read enable flag has to be dropped.

Regarding the outputs, the state machine can either provide packet data or no data, since for the idle state the entity's output is a tx flag equal to '0'. So, in the (State_S1), (State_S2) and (State_S3) the output ought to be the data of the corresponding FIFO, accompanied by a tx flag equal to '0'. In addition, the read enable flags have to taken into consideration, since they control the actual reading process from the FIFOs. The simulation results are presented below

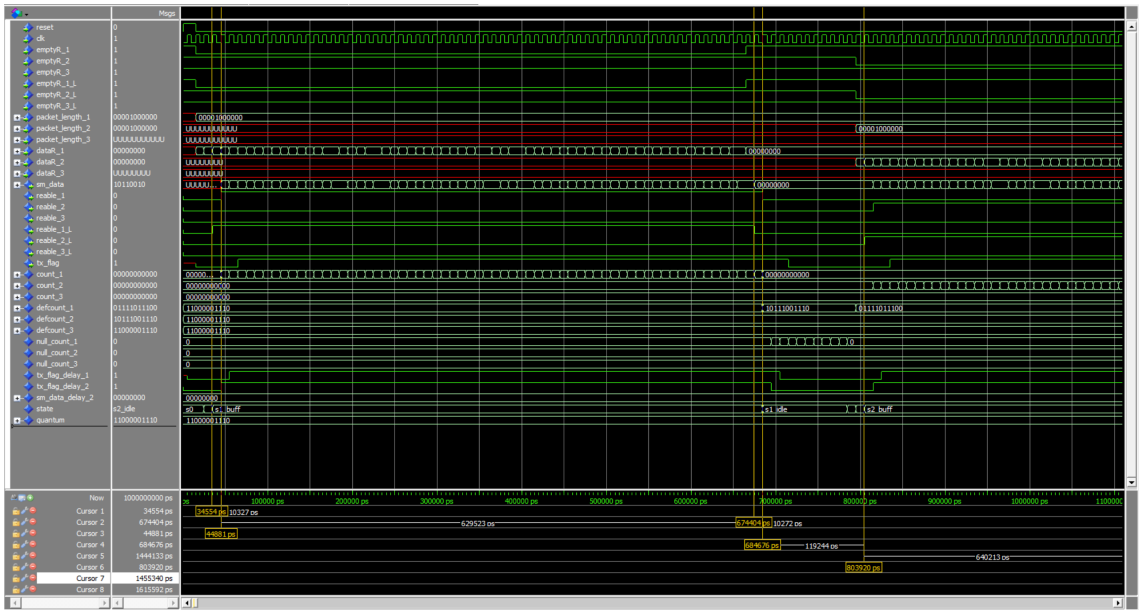


Figure 26: State Machine Simulation-One clock cycle delay between data and length read enable flags

In the figure above, the one clock cycle delay between the data and length read enable flags is illustrated. As presented in the next figure, the length read enable flag drops one clock cycle earlier before the state is transferred from State_S1_buff to State_S1_idle, which represents

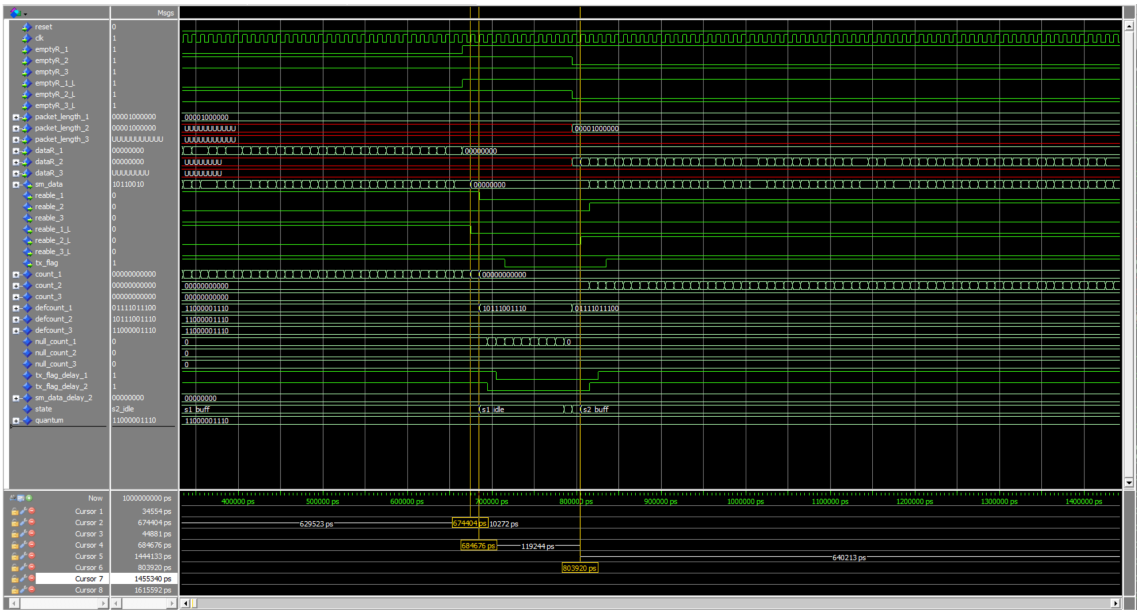


Figure 27: State Machine Simulation-Twelve clock cycle idle state between State_S1 and State_S2

As depicted above, the State_S1_idle lasts for twelve clock cycles. In that state all the read enable flags are '0'. The output might be an 8 bit vector of 0s. But this is just a coincidence because the last byte of the packet is an 8 bit vector of 0s. After, the specified number of cycles the state machine proceeds to state State_S2.

11.c.4 State Machine Controller

The simulation was performed for two clock cycles. During, these two clock cycles the inputs of the entity were : 2 bytes filled with 0s and 2 bytes filled with 1s. The tx flags, however ,were fixed to have a value of '1' only for the bytes filled with 0s and a value of '0' only for bytes filled with 1s. So the output of this clock cycle would be a 32 bit vector complete with 0s. On the next clock cycle, these values changed from 0 to 1. And the output would still be a 32 bit vector complete with 0s. The simulation results are presented below.

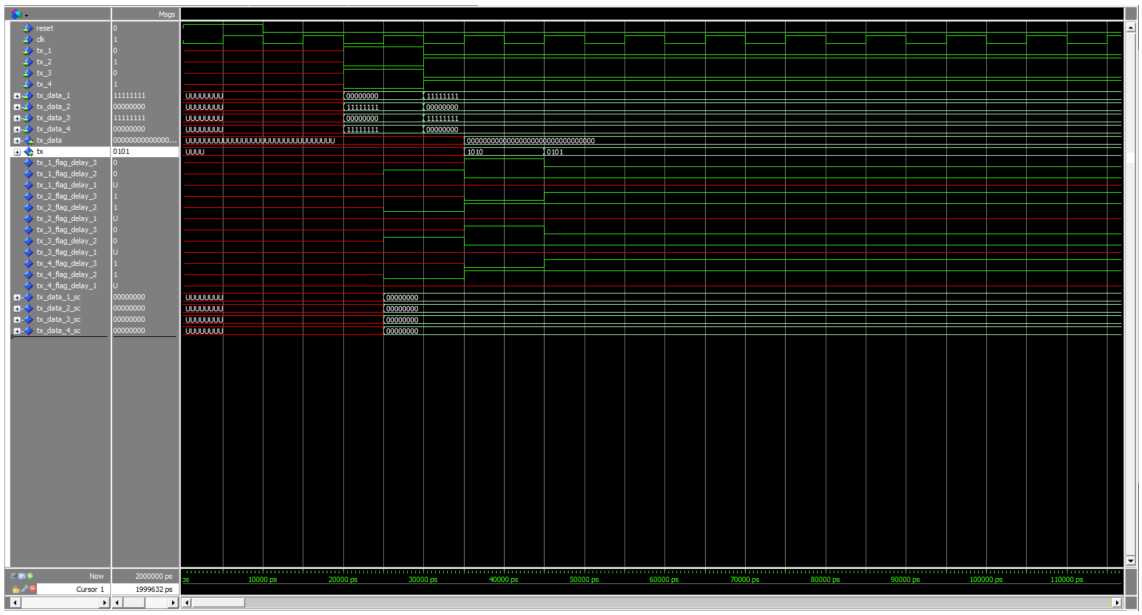


Figure 28: State Machine Controller Simulation Results

11.c.5 Output and Module Switch

In order to be able to assess the overall performance of this block a simulation is performed. The input of the simulation are four fixed packets. These packets are the same packets used in the simulation of the state machine, with the only difference being the fact that they include one extra bit ('0') in every byte so they can have the same vector length of the input of this block (9 bits). In detail the inputs are,

- Example Packet 1(576 bytes=64*9 bits),
- Example Packet Length 1 (11 bytes),
- Destination Port 1 (4 bytes),
- Example Packet 2(576 bytes=64*9 bits),
- Example Packet Length 2 (11 bytes),
- Destination Port 2 (4 bytes),
- Example Packet 3(576 bytes=64*9 bits),
- Example Packet Length 3 (11 bytes),
- Destination Port 3 (4 bytes),
- Example Packet 4(576 bytes=64*9 bits),
- Example Packet Length 4 (11 bytes),

- Destination Port 4 (4 bytes).

Every packet is scheduled to be passed to a different output port. To be more exact,

- Destination Port 1=0010(Output Port 2),
- Destination Port 2=0100(Output Port 3),
- Destination Port 3=1000(Output Port 4),
- Destination Port 4=0001(Output Port 1).

In accordance to the procedures described before, the data of the packets, as well as, the length of the packets should be stored in specific FIFOs which are,

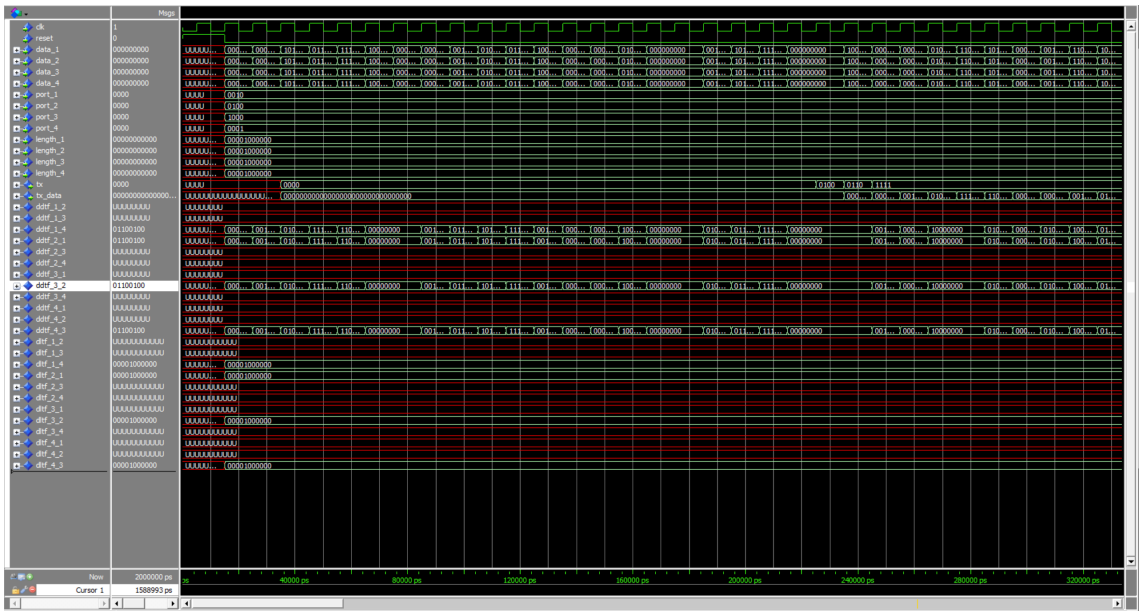


Figure 29: Switch and Output Buffering Block Simulation Result 1

From the figure above, it is evident that the data and length are stored in the desired FIFOs. The signals ddtf transfer packet data to the FIFOs, while dflt transfer packet length data to the FIFOs. Moreover, after a number of clock cycles packet data is being transferred to the output of the entity. Some state machines need more clock cycles before providing the packet data. This phenomenon can be seen on the change of the **tx** vector values. The signal acquires the value of "1111" after a number of clock cycles. On the other hand, since some packets are being transferred "faster" into the output, the bits of **tx** signal value will change earlier from others. This can be seen in the figure below, where some packet transmission ends faster than others, and as a consequence the idle state follows. However, after the packets have been successfully delivered to the corresponding ports there are no more data to be transferred and the output of the block is a 32 bit vector **tx_data** consisted of 0s and a 4 bit vector **tx** consisted of 0s, as well.

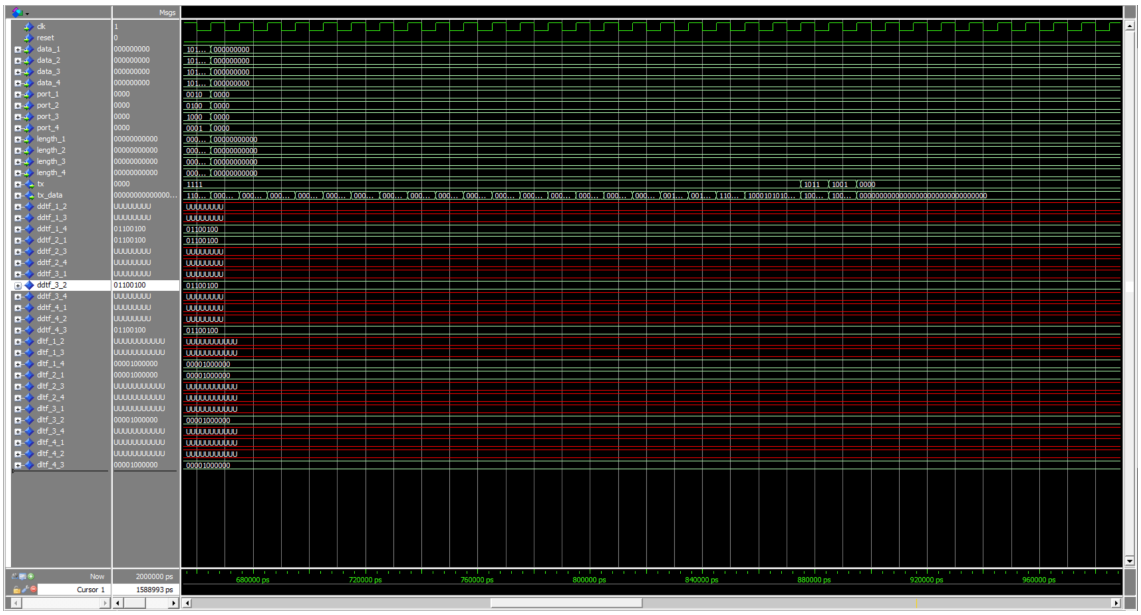


Figure 30: Switch and Output Buffering Block Simulation Result 2

11.d Verification

Since this is a gigabit switch, the clock used for timing analysis has a frequency of 1 GHz or time period of 1 ns. However the switch has 4 ports, each processing 1 byte thereby increasing the time period by 8 times to 8 ns or reducing the frequency to $\frac{1}{8ns} = 125$ MHz. The verification results are shown below:

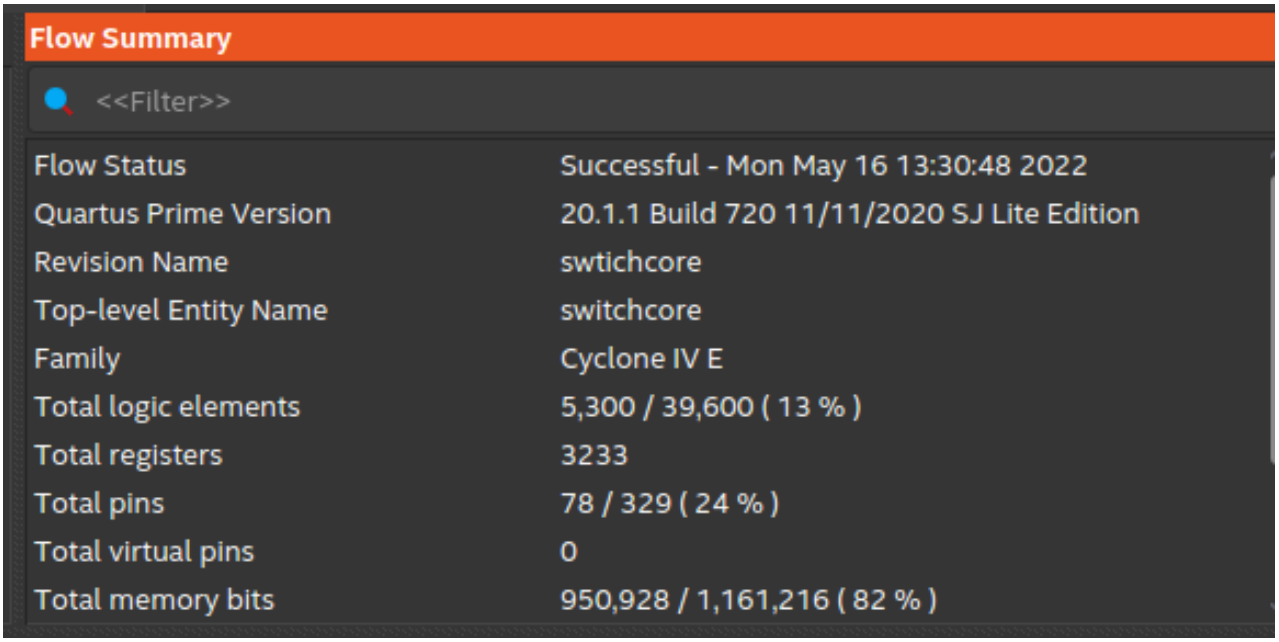


Figure 31: Flow Summary

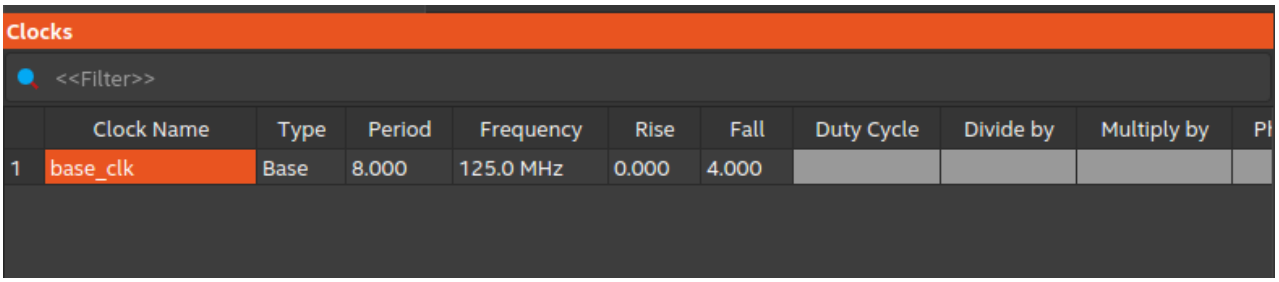


Figure 32: Base Clock

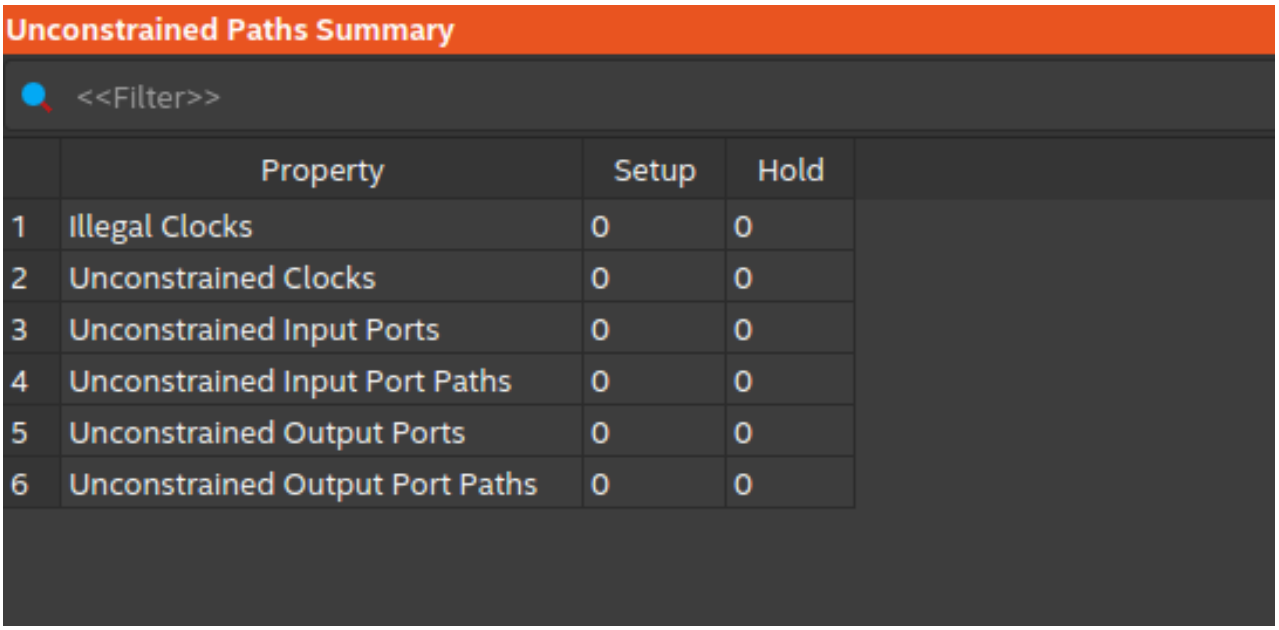
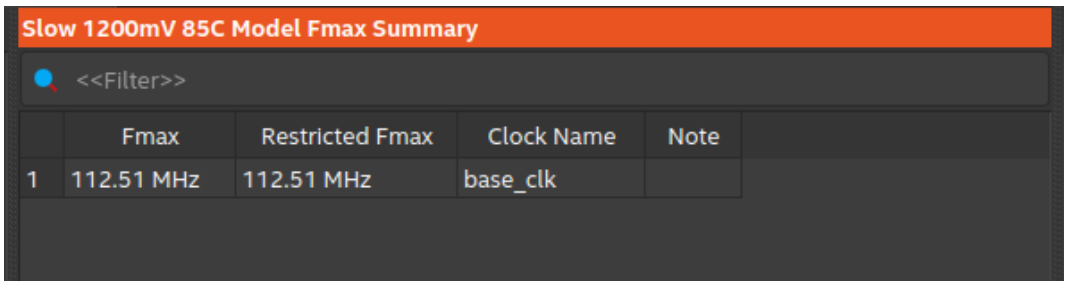


Figure 33: Unconstrained Path

12 Key Performance Parameters

Two important parameters that indicate performance are resource used and the maximum clock frequency denoted by Fmax. Fmax is the maximum clock frequency that can be achieved without violating internal setup and hold time requirements.

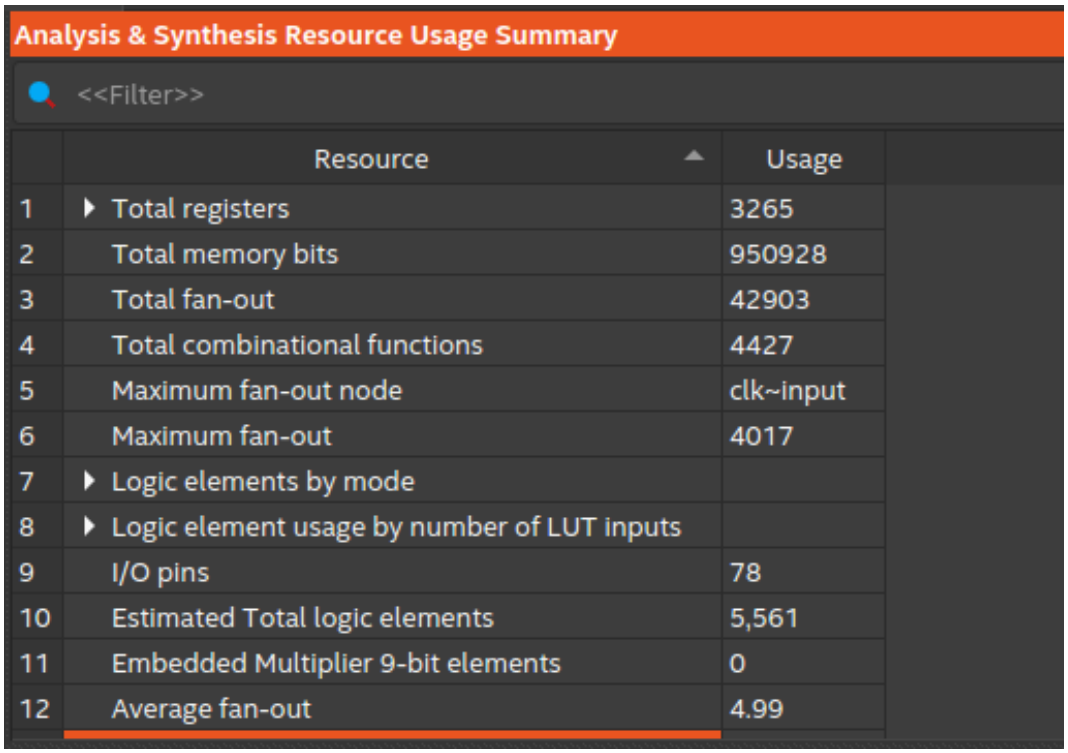


Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	112.51 MHz	112.51 MHz	base_clk	

Figure 34: Fmax



Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Total registers	3265
2	Total memory bits	950928
3	Total fan-out	42903
4	Total combinational functions	4427
5	Maximum fan-out node	clk~input
6	Maximum fan-out	4017
7	Logic elements by mode	
8	Logic element usage by number of LUT inputs	
9	I/O pins	78
10	Estimated Total logic elements	5,561
11	Embedded Multiplier 9-bit elements	0
12	Average fan-out	4.99

Figure 35: Resource Usage

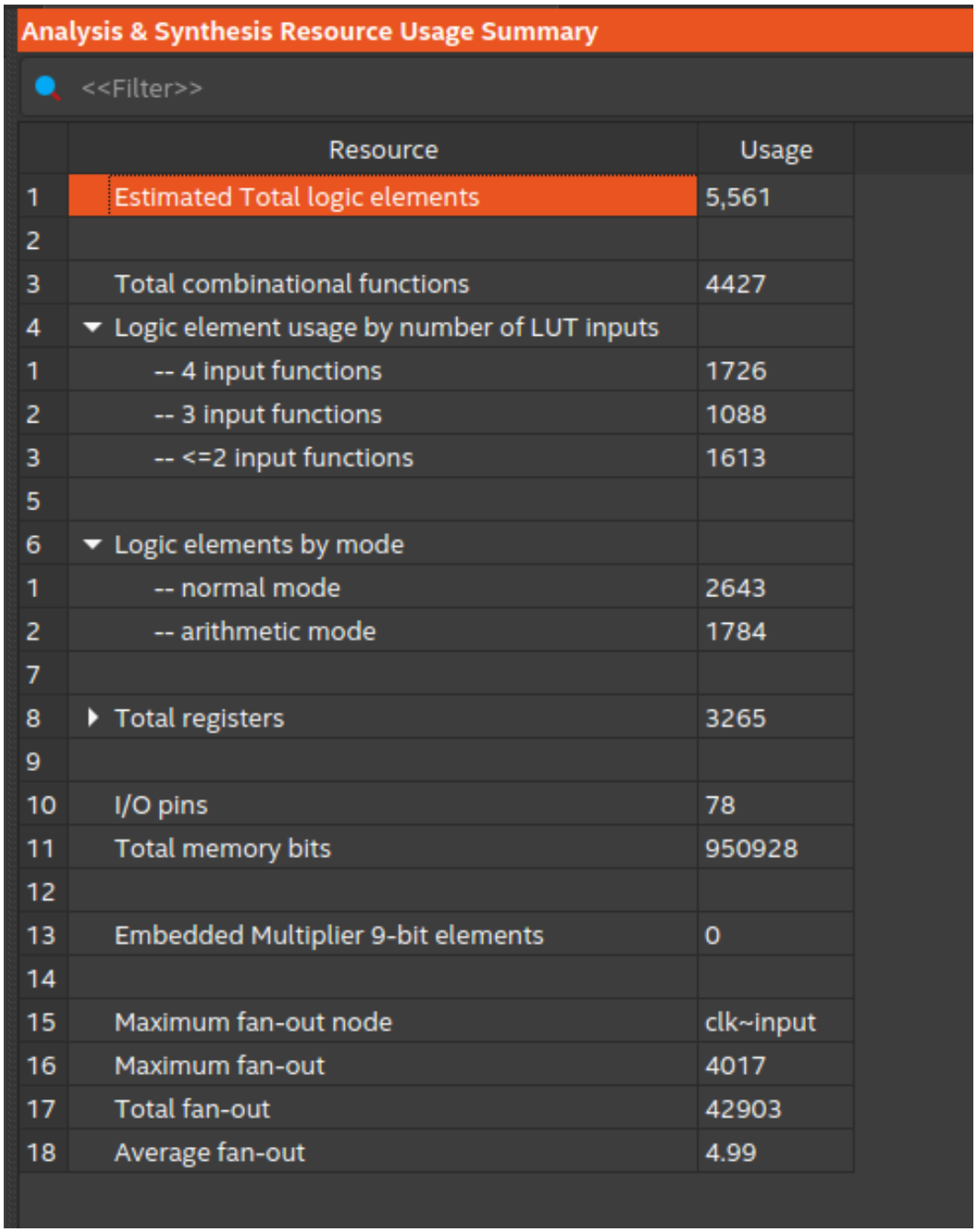


Figure 36: Resource Usage: Logical Elements

13 Conclusion

The gigabit ethernet switch was designed where the error check is performed in the input module and the cross-point switching architecture along with deficit round-robin is used. The design was simulated and verified using Intel Quartus Lite. As shown in 34, the Fmax value obtained is very close to the chosen clock in 32. Furthermore, the percentage of logical elements used is 13%.

14 References

[1] https://crccalc.com/

15 Appendix

15.a switchcore

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity switchcore is

        port
                        (
                        clk:                    in      std_logic;
                        reset:                  in      std_logic;

                                --Activity indicators
                        link_sync:              in      std_logic_vector(3 downto 0);

                                --Four GMII interfaces
                        tx_data:        out     std_logic_vector(31 downto 0);  --(7 dow
                        tx_ctrl:        out     std_logic_vector(3 downto 0);   --(0)=TX
                        rx_data:        in      std_logic_vector(31 downto 0);  --(7 dow
                        rx_ctrl:        in      std_logic_vector(3 downto 0)    --(0)=RX
                        );

end switchcore;

architecture arch of switchcore is

        component switchcore_input
                port
                        (
                                -- Input signals to the input block of switch
                        switch_inp_clk:                 in      std_logic;
                        switch_inp_rst:                 in      std_logic;
                        switch_inp_rx_data_in:  in      std_logic_vector(31 downto 0);
                        switch_inp_rx_ctrl:     in      std_logic_vector(3 downto 0);

                                -- I/O signals for input block and MAC learning block
                                -- Port 1
                        mac_ack1: in std_logic; -- Acknowledge signal for port 1 FIFO fr
                        inp_block_port_mac_in1: in std_logic_vector(3 downto 0); -- Port
                        mac_port1: in std_logic_vector(3 downto 0);     -- Destination p
                        mac_req1: out std_logic;        -- Request sent to MAC Learning
                        address1: out std_logic_vector(95 downto 0);    -- Sends src and
                        inp_block_port_mac_out1: out std_logic_vector(3 downto 0);
                                -- Port 2
                        mac_ack2: in std_logic; -- Acknowledge signal for port 2 FIFO fr
                        mac_port2: in std_logic_vector(3 downto 0);     -- Destination p
```

```
                inp_block_port_mac_in2: in std_logic_vector(3 downto 0); -- Port
                address2: out std_logic_vector(95 downto 0);    -- Sends src and
                inp_block_port_mac_out2: out std_logic_vector(3 downto 0);
                mac_req2: out std_logic;        -- Request sent to MAC Learning
                        -- Port 3
                mac_ack3: in std_logic; -- Acknowledge signal for port 3 FIFO fr
                mac_port3: in std_logic_vector(3 downto 0);     -- Destination p
                inp_block_port_mac_in3: in std_logic_vector(3 downto 0); -- Port
                address3: out std_logic_vector(95 downto 0);    -- Sends src and
                inp_block_port_mac_out3: out std_logic_vector(3 downto 0);
                mac_req3: out std_logic;        -- Request sent to MAC Learning
                        -- Port 4
                mac_ack4: in std_logic; -- Acknowledge signal for port 4 FIFO fr
                mac_port4: in std_logic_vector(3 downto 0);     -- Destination p
                inp_block_port_mac_in4: in std_logic_vector(3 downto 0); -- Port
                address4: out std_logic_vector(95 downto 0);    -- Sends src and
                inp_block_port_mac_out4: out std_logic_vector(3 downto 0);
                mac_req4: out std_logic;        -- Request sent to MAC Learning

                        -- Output signals from input block to the output block o
                        -- Port 1
                data_out1: out std_logic_vector(8 downto 0); -- Port 1 data to b
                pkt_length_port1: out std_logic_vector(10 downto 0);    -- Port
                output_port1: out std_logic_vector(3 downto 0); -- Destination p
                        -- Port 2
                data_out2: out std_logic_vector(8 downto 0); -- Port 2 data to b
                pkt_length_port2: out std_logic_vector(10 downto 0);    -- Port
                output_port2: out std_logic_vector(3 downto 0); -- Destination p
                        -- Port 3
                data_out3: out std_logic_vector(8 downto 0); -- Port 3 data to b
                pkt_length_port3: out std_logic_vector(10 downto 0);    -- Port
                output_port3: out std_logic_vector(3 downto 0); -- Destination p
                        -- Port 4
                data_out4: out std_logic_vector(8 downto 0); -- Port 4 data to b
                pkt_length_port4: out std_logic_vector(10 downto 0);    -- Port
                output_port4: out std_logic_vector(3 downto 0)  -- Destination p
                );
end component;

component MAC_arch
        port
                (
                clk:                    in      std_logic;
                reset:          in      std_logic;

                addresses_1:    in      std_logic_vector(95 downto 0);
                s_port_1:               in std_logic_vector(3 downto 0);
                req_1:                  in      std_logic;
```

```
                ack_1:                  out     std_logic;
                d_port_1:               out     std_logic_vector(3 downto 0);

                addresses_2:    in      std_logic_vector(95 downto 0);
                s_port_2:               in std_logic_vector(3 downto 0);
                req_2:                  in      std_logic;

                ack_2:                  out     std_logic;
                d_port_2:               out     std_logic_vector(3 downto 0);

                addresses_3:    in      std_logic_vector(95 downto 0);
                s_port_3:               in std_logic_vector(3 downto 0);
                req_3:                  in      std_logic;

                ack_3:                  out     std_logic;
                d_port_3:               out     std_logic_vector(3 downto 0);

                addresses_4:    in      std_logic_vector(95 downto 0);
                s_port_4:               in std_logic_vector(3 downto 0);
                req_4:                  in      std_logic;

                ack_4:                  out     std_logic;
                d_port_4:               out     std_logic_vector(3 downto 0)
        );
end component;

component scheduling_buffering
        port (
clk             : in  std_logic;
reset           : in  std_logic;
data_1          : in  std_logic_vector(8 downto 0);
data_2          : in  std_logic_vector(8 downto 0);
data_3          : in  std_logic_vector(8 downto 0);
data_4          : in  std_logic_vector(8 downto 0);
port_1          : in std_logic_vector (3 downto 0);
port_2          : in std_logic_vector (3 downto 0);
port_3          : in std_logic_vector (3 downto 0);
port_4          : in std_logic_vector (3 downto 0);
length_1        : in std_logic_vector(10 downto 0);
length_2        : in std_logic_vector(10 downto 0);
length_3        : in std_logic_vector(10 downto 0);
length_4        : in std_logic_vector(10 downto 0);
tx              : out std_logic_vector(3 downto 0);
tx_data         : out std_logic_vector(31 downto 0)
          );
end component;

-- Signals
-- Switch Input
```

```
        signal temp_mac_ack1: std_logic;
        signal temp_mac_req1: std_logic;
        signal temp_inp_block_port_mac_in1: std_logic_vector(3 downto 0);
        signal temp_mac_port1: std_logic_vector(3 downto 0);
        signal temp_address1: std_logic_vector(95 downto 0);
        signal temp_inp_block_port_mac_out1: std_logic_vector(3 downto 0);
        signal temp_data_out1: std_logic_vector(8 downto 0);
        signal temp_pkt_length_port1: std_logic_vector(10 downto 0);
        signal temp_output_port1: std_logic_vector(3 downto 0);
        signal temp_mac_ack2: std_logic;
        signal temp_mac_req2: std_logic;
        signal temp_inp_block_port_mac_in2: std_logic_vector(3 downto 0);
        signal temp_mac_port2: std_logic_vector(3 downto 0);
        signal temp_address2: std_logic_vector(95 downto 0);
        signal temp_inp_block_port_mac_out2: std_logic_vector(3 downto 0);
        signal temp_data_out2: std_logic_vector(8 downto 0);
        signal temp_pkt_length_port2: std_logic_vector(10 downto 0);
        signal temp_output_port2: std_logic_vector(3 downto 0);
        signal temp_mac_ack3: std_logic;
        signal temp_mac_req3: std_logic;
        signal temp_inp_block_port_mac_in3: std_logic_vector(3 downto 0);
        signal temp_mac_port3: std_logic_vector(3 downto 0);
        signal temp_address3: std_logic_vector(95 downto 0);
        signal temp_inp_block_port_mac_out3: std_logic_vector(3 downto 0);
        signal temp_data_out3: std_logic_vector(8 downto 0);
        signal temp_pkt_length_port3: std_logic_vector(10 downto 0);
        signal temp_output_port3: std_logic_vector(3 downto 0);
        signal temp_mac_ack4: std_logic;
        signal temp_mac_req4: std_logic;
        signal temp_inp_block_port_mac_in4: std_logic_vector(3 downto 0);
        signal temp_mac_port4: std_logic_vector(3 downto 0);
        signal temp_address4: std_logic_vector(95 downto 0);
        signal temp_inp_block_port_mac_out4: std_logic_vector(3 downto 0);
        signal temp_data_out4: std_logic_vector(8 downto 0);
        signal temp_pkt_length_port4: std_logic_vector(10 downto 0);
        signal temp_output_port4: std_logic_vector(3 downto 0);



BEGIN

-- Port maps
        switch_input: switchcore_input
                port map(
                        -- Port 1
                        switch_inp_clk => clk,
                        switch_inp_rst => reset,
                        switch_inp_rx_data_in => rx_data,
```

```
                    switch_inp_rx_ctrl => rx_ctrl,
                    mac_ack1 => temp_mac_ack1,
                    inp_block_port_mac_in1 => "0001",
                    mac_port1 => temp_mac_port1,
                    mac_req1 => temp_mac_req1,
                    address1 => temp_address1,
                    inp_block_port_mac_out1 => temp_inp_block_port_mac_out1,
                    data_out1 => temp_data_out1,
                    pkt_length_port1 => temp_pkt_length_port1,
                    output_port1 => temp_output_port1,
                    -- Port 2
                    mac_ack2 => temp_mac_ack2,
                    inp_block_port_mac_in2 => "0010",
                    mac_port2 => temp_mac_port2,
                    mac_req2 => temp_mac_req2,
                    address2 => temp_address2,
                    inp_block_port_mac_out2 => temp_inp_block_port_mac_out2,
                    data_out2 => temp_data_out2,
                    pkt_length_port2 => temp_pkt_length_port2,
                    output_port2 => temp_output_port2,
                    -- Port 3
                    mac_ack3 => temp_mac_ack3,
                    inp_block_port_mac_in3 => "0100",
                    mac_port3 => temp_mac_port3,
                    mac_req3 => temp_mac_req3,
                    address3 => temp_address3,
                    inp_block_port_mac_out3 => temp_inp_block_port_mac_out3,
                    data_out3 => temp_data_out3,
                    pkt_length_port3 => temp_pkt_length_port3,
                    output_port3 => temp_output_port3,
                    -- Port 4
                    mac_ack4 => temp_mac_ack4,
                    inp_block_port_mac_in4 => "1000",
                    mac_port4 => temp_mac_port4,
                    mac_req4 => temp_mac_req4,
                    address4 => temp_address4,
                    inp_block_port_mac_out4 => temp_inp_block_port_mac_out4,
                    data_out4 => temp_data_out4,
                    pkt_length_port4 => temp_pkt_length_port4,
                    output_port4 => temp_output_port4
);


mac_arc: MAC_arch
port map(
        clk => clk,
        reset => reset,
        addresses_1 => temp_address1,
        s_port_1 => temp_inp_block_port_mac_out1,
```

```
                req_1 => temp_mac_req1,
                ack_1 => temp_mac_ack1,
                d_port_1 => temp_mac_port1,
                addresses_2 => temp_address2,
                s_port_2 => temp_inp_block_port_mac_out2,
                req_2 => temp_mac_req2,
                ack_2 => temp_mac_ack2,
                d_port_2 => temp_mac_port2,
                addresses_3 => temp_address3,
                s_port_3 => temp_inp_block_port_mac_out3,
                req_3 => temp_mac_req3,
                ack_3 => temp_mac_ack3,
                d_port_3 => temp_mac_port3,
                addresses_4 => temp_address4,
                s_port_4 => temp_inp_block_port_mac_out4,
                req_4 => temp_mac_req4,
                ack_4 => temp_mac_ack4,
                d_port_4 => temp_mac_port4
        );

        switch_output: scheduling_buffering
                port map (

                clk         => clk,
                reset       => reset,
                data_1      => temp_data_out1,
                data_2      => temp_data_out2,
                data_3      => temp_data_out3,
                data_4      => temp_data_out4,
                port_1      => temp_output_port1,
                port_2      => temp_output_port2,
                port_3      => temp_output_port3,
                port_4      => temp_output_port4,
                length_1    => temp_pkt_length_port1,
                length_2    => temp_pkt_length_port2,
                length_3    => temp_pkt_length_port3,
                length_4    => temp_pkt_length_port4,
                tx          => tx_ctrl,
                tx_data     => tx_data

                        );

END arch;
```

15.b Input Module

15.b.1 switch_input

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity switchcore_input is

        port
                        (
                                -- Input signals to the input block of switch
                                switch_inp_clk:                 in      std_logic;
                                switch_inp_rst:                 in      std_logic;
                                switch_inp_rx_data_in:  in      std_logic_vector(31 down
                                switch_inp_rx_ctrl:     in      std_logic_vector(3 downt

                                -- I/O signals for input block and MAC learning block
                                -- Port 1
                                mac_ack1: in std_logic; -- Acknowledge signal for port 1
                                inp_block_port_mac_in1: in std_logic_vector(3 downto 0);
                                mac_port1: in std_logic_vector(3 downto 0);     -- Desti
                                mac_req1: out std_logic;        -- Request sent to MAC L
                                address1: out std_logic_vector(95 downto 0);    -- Sends
                                inp_block_port_mac_out1: out std_logic_vector(3 downto 0
                                -- Port 2
                                mac_ack2: in std_logic; -- Acknowledge signal for port 2
                                mac_port2: in std_logic_vector(3 downto 0);     -- Desti
                                inp_block_port_mac_in2: in std_logic_vector(3 downto 0);
                                address2: out std_logic_vector(95 downto 0);    -- Sends
                                inp_block_port_mac_out2: out std_logic_vector(3 downto 0
                                mac_req2: out std_logic;        -- Request sent to MAC L
                                -- Port 3
                                mac_ack3: in std_logic; -- Acknowledge signal for port 3
                                mac_port3: in std_logic_vector(3 downto 0);     -- Desti
                                inp_block_port_mac_in3: in std_logic_vector(3 downto 0);
                                address3: out std_logic_vector(95 downto 0);    -- Sends
                                inp_block_port_mac_out3: out std_logic_vector(3 downto 0
                                mac_req3: out std_logic;        -- Request sent to MAC L
                                -- Port 4
                                mac_ack4: in std_logic; -- Acknowledge signal for port 4
                                mac_port4: in std_logic_vector(3 downto 0);     -- Desti
                                inp_block_port_mac_in4: in std_logic_vector(3 downto 0);
                                address4: out std_logic_vector(95 downto 0);    -- Sends
                                inp_block_port_mac_out4: out std_logic_vector(3 downto 0
                                mac_req4: out std_logic;        -- Request sent to MAC L

                                -- Output signals from input block to the output block o
                                -- Port 1
                                data_out1: out std_logic_vector(8 downto 0); -- Port 1 d
                                pkt_length_port1: out std_logic_vector(10 downto 0);
                                output_port1: out std_logic_vector(3 downto 0); -- Desti
                                -- Port 2
```

```
                                data_out2: out std_logic_vector(8 downto 0); -- Port 2 d
                                pkt_length_port2: out std_logic_vector(10 downto 0);
                                output_port2: out std_logic_vector(3 downto 0); -- Desti
                                -- Port 3
                                data_out3: out std_logic_vector(8 downto 0); -- Port 3 d
                                pkt_length_port3: out std_logic_vector(10 downto 0);
                                output_port3: out std_logic_vector(3 downto 0); -- Desti
                                -- Port 4
                                data_out4: out std_logic_vector(8 downto 0); -- Port 4 d
                                pkt_length_port4:out std_logic_vector(10 downto 0);
                                output_port4: out std_logic_vector(3 downto 0)  -- Desti
                        );

end switchcore_input;

architecture behavioural of switchcore_input is

        component switchcore_input_block
                port (
                        inp_block_rx_ctrl:      in std_logic;
                        inp_block_data_in:      in std_logic_vector(7 downto 0);
                        inp_block_clk: in std_logic;
                        inp_block_rst: in std_logic;
                        mac_ack: in std_logic; -- Acknowledge signal from MAC block
                        mac_port: in std_logic_vector(3 downto 0);      -- Destination p
                        address: out std_logic_vector(95 downto 0);     -- Sends src and
                        inp_block_port_mac_in: in std_logic_vector(3 downto 0); -- Port
                        inp_block_port_mac_out: out std_logic_vector(3 downto 0);
                        mac_req: out std_logic; -- Request sent to MAC Learning block fr
                        -- fcs_error_check: out std_logic;
                        -- start_of_frame: out std_logic;
                        -- end_of_frame: out std_logic;
                        inp_block_data_out:     out std_logic_vector(8 downto 0);
                        inp_block_port_out:     out std_logic_vector(3 downto 0);
                        inp_block_pkt_length:   out std_logic_vector(10 downto 0) -- sen
                );
        end component;

        -- Signals
--      signal temp_mac_req1: std_logic;        -- MAC request from port 1 fifo
--      signal temp_mac_req2: std_logic;        -- MAC request from port 2 fifo
--      signal temp_mac_req3: std_logic;        -- MAC request from port 3 fifo
--      signal temp_mac_req4: std_logic;        -- MAC request from port 3 fifo

begin
--      mac_req(0) <= temp_mac_req1;
--      mac_req(1) <= temp_mac_req2;
--      mac_req(2) <= temp_mac_req3;
--      mac_req(3) <= temp_mac_req4;
```

```
unit_port1: switchcore_input_block
        port map (
                inp_block_rx_ctrl => switch_inp_rx_ctrl(0),
                inp_block_data_in => switch_inp_rx_data_in(7 downto 0),
                inp_block_clk => switch_inp_clk,
                inp_block_rst => switch_inp_rst,
                mac_ack => mac_ack1,
                mac_port => mac_port1,
                address => address1,
                inp_block_port_mac_in => inp_block_port_mac_in1,
                inp_block_port_mac_out => inp_block_port_mac_out1,
                mac_req => mac_req1,
                -- fcs_error_check: out std_logic;
                -- start_of_frame: out std_logic;
                -- end_of_frame: out std_logic;
                inp_block_data_out => data_out1,
                inp_block_port_out => output_port1,
                inp_block_pkt_length => pkt_length_port1
        );

unit_port2: switchcore_input_block
        port map (
                inp_block_rx_ctrl => switch_inp_rx_ctrl(1),
                inp_block_data_in => switch_inp_rx_data_in(15 downto 8),
                inp_block_clk => switch_inp_clk,
                inp_block_rst => switch_inp_rst,
                mac_ack => mac_ack2,
                mac_port => mac_port2,
                address => address2,
                inp_block_port_mac_in => inp_block_port_mac_in2,
                inp_block_port_mac_out => inp_block_port_mac_out2,
                mac_req => mac_req2,
                -- fcs_error_check: out std_logic;
                -- start_of_frame: out std_logic;
                -- end_of_frame: out std_logic;
                inp_block_data_out => data_out2,
                inp_block_port_out => output_port2,
                inp_block_pkt_length => pkt_length_port2
        );

unit_port3: switchcore_input_block
        port map (
                inp_block_rx_ctrl => switch_inp_rx_ctrl(2),
                inp_block_data_in => switch_inp_rx_data_in(23 downto 16),
                inp_block_clk => switch_inp_clk,
                inp_block_rst => switch_inp_rst,
                mac_ack => mac_ack3,
                mac_port => mac_port3,
```

```
                        address => address3,
                        inp_block_port_mac_in => inp_block_port_mac_in3,
                        inp_block_port_mac_out => inp_block_port_mac_out3,
                        mac_req => mac_req3,
                        -- fcs_error_check: out std_logic;
                        -- start_of_frame: out std_logic;
                        -- end_of_frame: out std_logic;
                        inp_block_data_out => data_out3,
                        inp_block_port_out => output_port3,
                        inp_block_pkt_length => pkt_length_port3
                );

        unit_port4: switchcore_input_block
                port map (
                        inp_block_rx_ctrl => switch_inp_rx_ctrl(3),
                        inp_block_data_in => switch_inp_rx_data_in(31 downto 24),
                        inp_block_clk => switch_inp_clk,
                        inp_block_rst => switch_inp_rst,
                        mac_ack => mac_ack4,
                        mac_port => mac_port4,
                        address => address4,
                        inp_block_port_mac_in => inp_block_port_mac_in4,
                        inp_block_port_mac_out => inp_block_port_mac_out4,
                        mac_req => mac_req4,
                        -- fcs_error_check: out std_logic;
                        -- start_of_frame: out std_logic;
                        -- end_of_frame: out std_logic;
                        inp_block_data_out => data_out4,
                        inp_block_port_out => output_port4,
                        inp_block_pkt_length => pkt_length_port4
                );


end behavioural;
```

15.b.2 switchcore_input_block

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity switchcore_input_block is
        port(
                inp_block_rx_ctrl:      in std_logic;
                inp_block_data_in:      in std_logic_vector(7 downto 0);
                inp_block_clk: in std_logic;
                inp_block_rst: in std_logic;
                mac_ack: in std_logic; -- Acknowledge signal from MAC block
```

```
                mac_port: in std_logic_vector(3 downto 0);      -- Destination port to s
                address: out std_logic_vector(95 downto 0);     -- Sends src and dest ma
                inp_block_port_mac_in: in std_logic_vector(3 downto 0); -- Port correspo
                inp_block_port_mac_out: out std_logic_vector(3 downto 0);       -- FIFO
                mac_req: out std_logic; -- Request sent to MAC Learning block from FIFO1
                -- fcs_error_check: out std_logic;
                -- start_of_frame: out std_logic;
                -- end_of_frame: out std_logic;
                inp_block_data_out:     out std_logic_vector(8 downto 0);       -- Data
                inp_block_port_out:     out std_logic_vector(3 downto 0);       -- Desti
                inp_block_pkt_length: out std_logic_vector(10 downto 0) -- Sends packet
                -- inp_block_pkt_length:        out integer range 0 to 1550 -- sends pac
        );
end switchcore_input_block;

architecture behavioural of switchcore_input_block is

        -- Performs FCS of every byte
        component fcs_check_parallel
                port(
                        clk: in std_logic;
                        reset: in std_logic;
                        start_of_frame: in std_logic;   -- Arrival of first byte
                        data_in: in std_logic_vector(7 downto 0);
                        fcs_error: out std_logic        -- Indicates an error in receive
                );
        end component;

        -- Input FIFO to store packet
        component sync_fifo
                port (
                        clock           : IN STD_LOGIC ;
                        data            : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
                        rdreq           : IN STD_LOGIC ;
                        wrreq           : IN STD_LOGIC ;
                        empty           : OUT STD_LOGIC ;
                        full            : OUT STD_LOGIC ;
                        q               : OUT STD_LOGIC_VECTOR (8 DOWNTO 0)
                );
        end component;

        -- FIFO to store each packets length
        component pkt_length_fifo
                port(
                        clock           : IN STD_LOGIC ;
                        data            : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
                        rdreq           : IN STD_LOGIC ;
                        wrreq           : IN STD_LOGIC ;
                        empty           : OUT STD_LOGIC ;
```

```
                full            : OUT STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (11 DOWNTO 0)
        );
end component;

-- Computes SOF and EOF for FCS
component rxctrl_delay
        port(
                rx_ctrl_clk: in std_logic;
                rx_ctrl : in std_logic;
                start_of_frame: out std_logic;
                end_of_frame: out std_logic
        );
end component;

-- Counter to count the length of a packet
component pkt_length_counter
        port(
                counter_clk: in std_logic;
                counter_rst: in std_logic;
                counter_rx_ctrl: in std_logic;
                counter_data_in: in std_logic_vector(7 downto 0);
                pkt_length: out integer range 0 to 1550;
                pkt_length_vector: out std_logic_vector(10 downto 0)
        );
end component;

-- Address FIFO
component address_fifo_wrapper
        port (
                add_rx_ctrl: in std_logic;      -- high when switch receives pac
                add_clock: in std_logic;        -- clock for this block
                add_reset: in std_logic;        -- reset for this block
                add_req: in std_logic;          -- request signal to retreive SR
                add_data_in: in std_logic_vector(7 downto 0);   -- input data to
                add_ack: out std_logic;         -- acknowledge signal when readi
                -- add_data_out: out std_logic_vector(7 downto 0);
                src_address: out std_logic_vector(47 downto 0); -- sends SRC add
                dest_address: out std_logic_vector(47 downto 0) -- sends DEST ad
        );
end component;

-- Signals
signal temp_mac_req: std_logic := '0';
signal SOF: std_logic;  -- SOF for data corresponding to each port
signal EOF: std_logic;  -- EOF for data corresponding to each port
signal temp_fcs_error_check: std_logic; -- Decides whether to discard currently
signal r_en: std_logic; -- Input packet FIFO read enable set in the state machin
signal w_en: std_logic; -- Input packet FIFO write enable set in the state machi
```

```
        signal fifo_full: std_logic;    -- FIFO full flag
        signal fifo_empty: std_logic; -- FIFO empty flag
        signal temp_dest_address: std_logic_vector(47 downto 0) := (others => '0');
        signal temp_src_address: std_logic_vector(47 downto 0) := (others => '0');
        signal address_counter: unsigned(4 downto 0); -- keeps r_en high until read_ptr
        -- signal address_idx_counter: integer range 0 to 100;
        signal pkt_counter: std_logic_vector(10 downto 0);
        signal fifo_data: std_logic_vector(8 downto 0); -- stores data from FIFO
        signal temp_packet_length1: integer range 0 to 1550;
        -- signal temp_packet_length2: integer range 0 to 1550;
        signal temp_packet_length_vector1: std_logic_vector(10 downto 0);
        signal temp_packet_length_vector2: std_logic_vector(11 downto 0) := (others => '
        signal temp_inp_block_data_in: std_logic_vector(7 downto 0);
        -- signal temp_inp_block_port_mac: std_logic_vector(3 downto 0) := "0000";
        -- signal temp_inp_block_port_out: std_logic_vector(3 downto 0);
        signal r_en_pkt: std_logic; -- Packet length FIFO read enable
        signal w_en_pkt: std_logic;     -- Packet length FIFO write enable
        signal fifo_full_pkt: std_logic;        -- Packet length FIFO full flag
        signal fifo_empty_pkt: std_logic; -- Packet length FIFO empty flag
        signal temp_add_req: std_logic; -- Request signal to fetch src and dest address
        signal temp_add_ack: std_logic; -- Acknowledge signal from address FIFO

        -- State machine states
        TYPE state_type is (ERROR_CHECK, DECIDE_NEXT_STATE, GET_ADDR, SEND_MAC_REQ, WAIT
    signal state: state_type := ERROR_CHECK;

begin
        -- inp_block_data_out <= fifo_data;
        temp_inp_block_data_in <= inp_block_data_in;
        mac_req <= temp_mac_req;
        -- inp_block_port_out <= temp_inp_block_port_out;

        -- Port maps
        -- Generating SOF and EOF signals for each port
        delay_rxctrl: rxctrl_delay
                port map(
                        rx_ctrl_clk => inp_block_clk,
                        rx_ctrl => inp_block_rx_ctrl,
                        start_of_frame => SOF,
                        end_of_frame => EOF
                );

        -- FCS Check
        FCS: fcs_check_parallel
                port map(
                        clk => inp_block_clk,
                        reset => inp_block_rst,
                        start_of_frame => SOF,
                        data_in => temp_inp_block_data_in,
```

```
                fcs_error => temp_fcs_error_check
        );

-- Storing in sync fifo
store_data: sync_fifo
        port map(
                clock   => inp_block_clk,
                data(7 downto 0) => temp_inp_block_data_in,
                data(8) => EOF,
                rdreq   =>      r_en,
                wrreq   => w_en,
                empty   => fifo_empty,
                full => fifo_full,
                q => fifo_data
        );

--      Counting pakcet length
counter: pkt_length_counter
        port map(
                counter_clk => inp_block_clk,
                counter_rst => inp_block_rst,
                counter_rx_ctrl => inp_block_rx_ctrl,
                counter_data_in => temp_inp_block_data_in,
                pkt_length => temp_packet_length1,
                pkt_length_vector => temp_packet_length_vector1
        );

-- Storing packet length in pkt_length_fifo
store_pkt_length: pkt_length_fifo
        port map(
                clock => inp_block_clk,
                data(10 downto 0) => temp_packet_length_vector1,
                data(11) => EOF,
                rdreq   => r_en_pkt,
                wrreq   => w_en_pkt,
                empty => fifo_empty_pkt,
                full => fifo_full_pkt,
                q => temp_packet_length_vector2
        );

-- stores SRC and DEST address as well as retrieves to send to MAC Learning
address_fifo: address_fifo_wrapper
        port map(
                add_clock => inp_block_clk,
                add_reset => inp_block_rst,
                add_rx_ctrl => inp_block_rx_ctrl,
                add_req => temp_add_req,
                add_data_in => temp_inp_block_data_in,
                -- add_data_out => add_data_out,
```

```
                add_ack => temp_add_ack,
                src_address => temp_src_address,
                dest_address => temp_dest_address
        );


        -- Writing to the input packet FIFO
        process(inp_block_rx_ctrl, inp_block_clk, inp_block_rst)
                begin
                        if (inp_block_rst = '1') then
                                w_en <= '0';
                        elsif (rising_edge(inp_block_clk)) then
                                if inp_block_rx_ctrl = '1' then
                                        w_en <= '1';
                                else
                                        w_en <= '0';
                                end if;
                        end if;
        end process;

        -- Writing to packet length FIFO
        process(inp_block_clk, inp_block_rst)
                begin
                        if (inp_block_rst = '1') then
                                w_en_pkt <= '0';
                        elsif (rising_edge(inp_block_clk)) then
                                if EOF = '1' then
                                        w_en_pkt <= '1';
                                else
                                        w_en_pkt <= '0';
                                end if;
                        end if;
        end process;


        -- Reading from the FIFO
--      process(inp_block_clk, inp_block_rst)
--      begin
--              if (inp_block_rst = '1') then
--                      r_en <= '0';
--              elsif (rising_edge(inp_block_clk)) then
--                      if (inp_block_rx_ctrl = '0') then
--                              r_en <= '1';
--                      else
--                              r_en <= '0';
--                      end if;
--              end if;
--      end process;
```

```
-- State machine for this input block
process(inp_block_clk, inp_block_rst)
        begin
                if (inp_block_rst = '1') then
                        inp_block_data_out <= "000000000";
                        inp_block_port_out <= "0000";
                        address_counter <= (others => '0');
                        address <= (others => '0');
                        pkt_counter <= (others => '0');
                        state <= ERROR_CHECK;
                        r_en <= '0';
                        r_en_pkt <= '0';

                elsif (rising_edge(inp_block_clk)) then
                        case state is
                                when ERROR_CHECK =>
                                        -- Initializing read enables to low
                                        r_en <= '0';
                                        r_en_pkt <= '0';
                                        temp_add_req <= '0';
                                        -- FCS error check and write to FIFO par
                                        if (EOF = '1') then
                                                state <= DECIDE_NEXT_STATE;
                                        else
                                                state <= ERROR_CHECK;
                                        end if;
                                when DECIDE_NEXT_STATE =>
                                        -- Decides between packet deletion or MA
                                        if (temp_fcs_error_check = '0') then
                                                address_counter <= (others => '0
                                                temp_add_req <= '0';
                                                state <= GET_ADDR;
                                        else
                                                state <= DELETE_PKT;
                                        end if;
                                when GET_ADDR =>
                                        -- Gets DEST and SRC ADDR from address F
                                        if (temp_add_ack = '1') then
                                                state <= SEND_MAC_REQ;
                                        elsif (address_counter <= "01011") then
                                                address_counter <= address_count
                                                temp_add_req <= '1';
                                                state <= GET_ADDR;
                                        elsif (address_counter >= "01100" and ad
                                                address_counter <= address_count
                                                temp_add_req <= '0';
                                                state <= GET_ADDR;
                                        end if;
                                when SEND_MAC_REQ =>
```

```
                -- Send a request to MAC block
                temp_mac_req <= '1';
                inp_block_port_mac_out <= inp_block_port
                address(95 downto 48) <= temp_src_addres
                address(47 downto 0) <= temp_dest_addres
                state <= WAIT_FOR_MAC;
            when WAIT_FOR_MAC =>
                -- Waits for acknowledgement from MAC bl
                if mac_ack = '1' then
                        inp_block_port_out <= mac_port;
                        r_en <= '1';
                        r_en_pkt <= '1';
                        temp_mac_req <= '0';
                        state <= SEND_DATA_TO_OUTPUT;
                else
                        state <= WAIT_FOR_MAC;
                        -- r_en <= '1';
                end if;
            when SEND_DATA_TO_OUTPUT =>
                -- Sends data to output block
                if (pkt_counter = "1000001") then
                        pkt_counter <= (others => '0');
                        r_en <= '0';
                        r_en_pkt <= '0';
                        state <= ERROR_CHECK;
                        -- temp_packet_length2 <= 0;
                else
                        inp_block_port_out <= "0000";
                        pkt_counter <= pkt_counter+1;
                        inp_block_data_out <= fifo_data;
                        inp_block_pkt_length <= temp_pac
                        -- temp_packet_length2 <= to_int
                        r_en <= '1';
                        r_en_pkt <= '1';
                        -- end_of_frame <= fifo_data(8);
                        state <= SEND_DATA_TO_OUTPUT;
                end if;
            when DELETE_PKT =>
                -- Deletes error packet in FIFO
                if (pkt_counter = temp_packet_length_vec
                        pkt_counter <= (others => '0');
                        r_en <= '0';
                        state <= ERROR_CHECK;
                else
                        r_en <= '1';
                        pkt_counter <= pkt_counter+1;
                        state <= DELETE_PKT;
                end if;
    end case;
```

```
                        end if;
        end process;

end behavioural;
```

15.b.3 address_fifo_wrapper

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity address_fifo_wrapper is
        port (
                add_rx_ctrl: in std_logic;      -- high when switch receives packet
                add_clock: in std_logic;        -- clock for this block
                add_reset: in std_logic;        -- reset for this block
                add_req: in std_logic;          -- request signal to retreive SRC and DE
                add_data_in: in std_logic_vector(7 downto 0);   -- input data to store i
                add_ack: out std_logic;         -- acknowledge signal when reading is co
                -- add_data_out: out std_logic_vector(7 downto 0);
                src_address: out std_logic_vector(47 downto 0); -- sends SRC address to
                dest_address: out std_logic_vector(47 downto 0) -- sends DEST address to
        );
end address_fifo_wrapper;

architecture behavioural of address_fifo_wrapper is

        component address_fifo
                port(
                        clock           : IN STD_LOGIC ;
                        data            : IN STD_LOGIC_VECTOR (8 DOWNTO 0);
                        rdreq           : IN STD_LOGIC ;
                        wrreq           : IN STD_LOGIC ;
                        empty           : OUT STD_LOGIC ;
                        full            : OUT STD_LOGIC ;
                        q               : OUT STD_LOGIC_VECTOR (8 DOWNTO 0)
                );
        end component;

        -- FIFO signals
        signal r_en_add: std_logic;     -- Address FIFO read enable
        signal EOF_add: std_logic := '0';               -- EOF to separate addresses of
        signal w_en_add: std_logic;     -- Address FIFO write enable
        signal fifo_full_add: std_logic; -- Address FIFO full flag
        signal fifo_empty_add: std_logic; -- Address FIFO empty flag
        signal r_add_counter: integer range 0 to 20 := 0; -- Counter read SRC and DESR a
        signal wr_add_counter: integer range 0 to 1550 := 0; -- Counter to write SRC and
        signal EOF_out: std_logic;
```

```
        signal temp_add_ack: std_logic := '0';
        signal fifo_data_add: std_logic_vector(7 downto 0); -- Gets data from fifo
        signal temp_src_add: std_logic_vector(47 downto 0) := (others => '0');  -- Tempo
        signal temp_dest_add: std_logic_vector(47 downto 0) := (others => '0'); -- Tempo

begin

        store_address: address_fifo
                port map(
                        clock => add_clock,
                        data(7 downto 0) => add_data_in,
                        data(8) => EOF_add,
                        rdreq   => r_en_add,
                        wrreq   => w_en_add,
                        empty   => fifo_empty_add,
                        full => fifo_full_add,
                        q(7 downto 0) => fifo_data_add,
                        q(8) => EOF_out
                );

        add_ack <= temp_add_ack;
        src_address <= temp_src_add;
        dest_address <= temp_dest_add;
        -- add_data_out <= fifo_data_add;

        -- Writing to the address FIFO
        process(add_reset, add_clock)
                begin
                        if (add_reset = '1') then
                                w_en_add <= '0';
                                wr_add_counter <= 0;

                        elsif (rising_edge(add_clock)) then
                                if (add_rx_ctrl = '1') then
                                        if(wr_add_counter >= 7 and wr_add_counter < 19)
                                                if(wr_add_counter = 18) then
                                                        EOF_add <= '1';
                                                else
                                                        EOF_add <= '0';
                                                end if;
                                                w_en_add <= '1';
                                                -- wr_add_counter <= wr_add_counter + 1;
                                        elsif (wr_add_counter = 19) then
                                                w_en_add <= '0';
                                                EOF_add <= '1';
                                        end if;
                                        wr_add_counter <= wr_add_counter + 1;
                                else
                                        wr_add_counter <= 0;
```

```
                                end if;
                        end if;
        end process;


        -- Reading address from FIFO
        process(add_reset, add_clock)
                begin
                        if (add_reset = '1') then
                                r_en_add <= '0';
                                r_add_counter <= 0;
                                temp_add_ack <= '0';
                                temp_dest_add <= (others => '0');
                                temp_src_add <= (others => '0');
                        elsif (rising_edge(add_clock)) then
                                if (add_req = '1') then
                                        if (r_add_counter > 1 and r_add_counter <= 7) th
                                                temp_dest_add((r_add_counter-1)*8-1 down
                                        elsif (r_add_counter > 7 and r_add_counter <= 11
                                                temp_src_add((r_add_counter-7)*8-1 downt
                                        end if;
                                        r_en_add <= '1';
                                        r_add_counter <= r_add_counter + 1;
                                elsif (r_add_counter > 11 and r_add_counter <= 13) then
                                        temp_src_add((r_add_counter-7)*8-1 downto (r_add
                                                if (r_add_counter = 13) then
                                                        temp_add_ack <= '1';
                                                else
                                                        temp_add_ack <= '0';
                                                end if;
                                                r_en_add <= '1';
                                                r_add_counter <= r_add_counter + 1;
                                else
                                        r_en_add <= '0';
                                        r_add_counter <= 0;
                                end if;
                        end if;
        end process;

end behavioural;
```

15.b.4 pkt_length_counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;

entity pkt_length_counter is
```

```
        port(
                counter_clk: in std_logic;
                counter_rst: in std_logic;
                counter_rx_ctrl: in std_logic;  -- receives control signal to count
                counter_data_in: in std_logic_vector(7 downto 0);       -- input data re
                pkt_length: out integer range 0 to 1550;        -- length of packet in i
                pkt_length_vector: out std_logic_vector(10 downto 0) -- length of packet
        );

end pkt_length_counter;


architecture behavioural of pkt_length_counter is

        signal counter: integer range 0 to 1550;        -- to counts the packet length

begin

        -- Process to implement counter
        process (counter_clk, counter_rst, counter_data_in, counter_rx_ctrl)
                begin
                        if(counter_rst = '1') then
                                counter <= 0;
                        elsif (rising_edge(counter_clk)) then
                                if (counter_rx_ctrl = '1') then
                                        counter <= counter + 1;
                                else
                                        counter <= 0;
                                end if;
                        end if;
        end process;

        -- Delaying by a clock cycle to fetch the final count (length)
        -- and store in pkt_length_fifo
        process(counter_clk)
                begin
                        if(rising_edge(counter_clk)) then
                                pkt_length <= counter;
                                pkt_length_vector <= std_logic_vector(to_unsigned(counte
                        end if;
        end process;


end behavioural;
```

15.b.5 rxctrl_delay

```
library IEEE;
use IEEE.std_logic_1164.all;
```

```
use IEEE.std_logic_unsigned.all;

entity rxctrl_delay is
        port (
                                rx_ctrl_clk: in std_logic;
                                rx_ctrl : in std_logic;
                                start_of_frame: out std_logic;  -- SOF for data correspo
                                end_of_frame: out std_logic             -- EOF for data
                                -- rx_ctrl_d: out std_logic
                        );

end rxctrl_delay;

architecture behavioural of rxctrl_delay is
        signal rx_ctrl_delayed: std_logic;

begin
        -- rx_ctrl_d <= rx_ctrl_delayed;
        -- Register to delay rx_ctrl
        process (rx_ctrl_clk)
                begin
                        if rising_edge(rx_ctrl_clk) then
                                rx_ctrl_delayed <= rx_ctrl;
                        end if;
        end process;

        -- Generating SOF and EOF bits for each port using delayed rx control signal
        process(rx_ctrl, rx_ctrl_delayed)
                begin
                        if (rx_ctrl = '1' and rx_ctrl_delayed = '0') then
                                start_of_frame <= '1';
                        else
                                start_of_frame <= '0';
                        end if;
                        if (rx_ctrl = '0' and rx_ctrl_delayed = '1') then
                                end_of_frame <= '1';
                        else
                                end_of_frame <= '0';
                        end if;
        end process;
end behavioural;
```

15.b.6 fcs_check_parallel

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity fcs_check_parallel is
```

```
        port (
        clk: in std_logic;
        reset: in std_logic;
        start_of_frame: in std_logic;   -- Arrival of first byte
        data_in: in std_logic_vector(7 downto 0);
        fcs_error: out std_logic        -- Indicates an error in received data
        );
end fcs_check_parallel;

architecture behavioural of fcs_check_parallel is

        signal reg: std_logic_vector(31 downto 0) := (others => '0'); -- initializing re
        signal shift_count: unsigned(1 downto 0);       -- takes 8 bits at once so there
        signal data: std_logic_vector(7 downto 0);
        signal is_fcs_done: std_logic := '0';

begin
        -- Ethernet requirements:
        -- Complementing the first 32 bits of the frame either by
        -- waiting for the frame (start_of_frame is'1') or
        -- by executing appropriate number of register shifts.
        process (shift_count, start_of_frame, data_in)
                begin
                        data <= data_in;
                        if (shift_count < 3 or start_of_frame = '1') then
                                data <= not data_in;
                        end if;
        end process;

        -- Implementing 8-bit parallel LFSR. The XOR operations are derived
        -- from a matrix generated in matlab
        process (clk, reset)
                begin
                        if reset = '1' then
                                reg <= (others => '0');
                                shift_count <= (others => '0');
                                fcs_error <= '1';
                                is_fcs_done <= '0';
                        elsif rising_edge(clk) then
--                              if end_of_frame = '1' then
--                                      is_fcs_done <= '1';
--                              end if;

                                if start_of_frame = '1' then
                                        -- Resetting shift counter when SOF or EOF is '1
                                        -- when frame's first or last byte enters
                                        shift_count <= (others => '0');
                                elsif shift_count < 3 then
                                        shift_count <= shift_count + 1;
```

```
                end if;

                -- XOR operations
                reg(0) <= reg(24) xor reg(30) xor data(0);
                reg(1) <= reg(24) xor reg(25) xor reg(30) xor reg(31) xo
                reg(2) <= reg(24) xor reg(25) xor reg(26) xor reg(30) xo
                reg(3) <= reg(25) xor reg(26) xor reg(27) xor reg(31) xo
                reg(4) <= reg(24) xor reg(26) xor reg(27) xor reg(28) xo
                reg(5) <= reg(24) xor reg(25) xor reg(27) xor reg(28) xo
                                  reg(31) xor data(5);
                reg(6) <= reg(25) xor reg(26) xor reg(28) xor reg(29) xo
                                  data(6);
                reg(7)  <= reg(24) xor reg(26) xor reg(27) xor reg(29) x
                reg(8)  <= reg(0) xor reg(24) xor reg(25) xor reg(27) xo
                reg(9)  <= reg(1) xor reg(25) xor reg(26) xor reg(28) xo
                reg(10) <= reg(2) xor reg(24) xor reg(26) xor reg(27) xo
                reg(11) <= reg(3) xor reg(24) xor reg(25) xor reg(27) xo
                reg(12) <= reg(4) xor reg(24) xor reg(25) xor reg(26) xo
                                     reg(30);
                reg(13) <= reg(5) xor reg(25) xor reg(26) xor reg(27) xo
                                     reg(31);
                reg(14) <= reg(6) xor reg(26) xor reg(27) xor reg(28) xo
                reg(15) <= reg(7) xor reg(27) xor reg(28) xor reg(29) xo
                reg(16) <= reg(8) xor reg(24) xor reg(28) xor reg(29);
                reg(17) <= reg(9) xor reg(25) xor reg(29) xor reg(30);
                reg(18) <= reg(10) xor reg(26) xor reg(30) xor reg(31);
                reg(19) <= reg(11) xor reg(27) xor reg(31);
                reg(20) <= reg(12) xor reg(28);
                reg(21) <= reg(13) xor reg(29);
                reg(22) <= reg(14) xor reg(24);
                reg(23) <= reg(15) xor reg(24) xor reg(25) xor reg(30);
                reg(24) <= reg(16) xor reg(25) xor reg(26) xor reg(31);
                reg(25) <= reg(17) xor reg(26) xor reg(27);
                reg(26) <= reg(18) xor reg(24) xor reg(27) xor reg(28) x
                reg(27) <= reg(19) xor reg(25) xor reg(28) xor reg(29) x
                reg(28) <= reg(20) xor reg(26) xor reg(29) xor reg(30);
                reg(29) <= reg(21) xor reg(27) xor reg(30) xor reg(31);
                reg(30) <= reg(22) xor reg(28) xor reg(31);
                reg(31) <= reg(23) xor reg(29);


                if reg = "11111111111111111111111111111111" then
                        fcs_error <= '0';
                end if;
            end if;
        end process;

end behavioural;
```

15.c MAC Learning Module

15.c.1 MAC Learning Architecture

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity MAC_arch is

        port
                        (
                clk:                    in      std_logic;
                reset:          in      std_logic;

                addresses_1:                    in      std_logic_vector(95 downto 0);
                s_port_1:               in std_logic_vector(3 downto 0);
                req_1:                  in      std_logic;

                ack_1:                  out     std_logic;
                d_port_1:               out     std_logic_vector(3 downto 0);

                addresses_2:                    in      std_logic_vector(95 downto 0);
                s_port_2:               in std_logic_vector(3 downto 0);
                req_2:                  in      std_logic;

                ack_2:                  out     std_logic;
                d_port_2:               out     std_logic_vector(3 downto 0);

                addresses_3:                    in      std_logic_vector(95 downto 0);
                s_port_3:               in std_logic_vector(3 downto 0);
                req_3:                  in      std_logic;

                ack_3:                  out     std_logic;
                d_port_3:               out     std_logic_vector(3 downto 0);

                addresses_4:                    in      std_logic_vector(95 downto 0);
                s_port_4:               in std_logic_vector(3 downto 0);
                req_4:                  in      std_logic;

                ack_4:                  out     std_logic;
                d_port_4:               out     std_logic_vector(3 downto 0)
                        );

end MAC_arch;

architecture arch of MAC_arch is

        component MAC_RAM
                port
```

```
                (
                        clock           : IN STD_LOGIC  := '1';
                        data            : IN STD_LOGIC_VECTOR (51 DOWNTO 0);
                        rdaddress               : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                        wraddress               : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                        wren            : IN STD_LOGIC  := '0';
                        q               : OUT STD_LOGIC_VECTOR (51 DOWNTO 0)
                );
        end component;

        component CRC16
                port
                (
                        dataIn : in std_logic_vector (47 downto 0);
                        rst, clk : in std_logic;
                        done: out std_logic;
                        crcOut : out std_logic_vector (15 downto 0)
                );
        end component;

TYPE state_type IS (check_p1, check_p2, check_p3,
                check_p4, hashing_src,
                hashing_dest, table_processes, compare_mac, send_ack);
SIGNAL state : state_type :=check_p1;

SIGNAL MACdata: STD_LOGIC_VECTOR (51 DOWNTO 0);
SIGNAL MACrdaddress: STD_LOGIC_VECTOR (12 DOWNTO 0);
SIGNAL MACwraddress: STD_LOGIC_VECTOR (12 DOWNTO 0);
SIGNAL MACwren: STD_LOGIC;
SIGNAL MACq: STD_LOGIC_VECTOR (51 DOWNTO 0);


SIGNAL CRCdata: std_logic_vector (47 downto 0);
SIGNAL crc_done: std_logic;
SIGNAL crc_reset: std_logic;
SIGNAL crc_out: std_logic_vector (15 downto 0);


SIGNAL src_addr: std_logic_vector(47 downto 0);
SIGNAL dest_addr: std_logic_vector(47 downto 0);
SIGNAL s_port: std_logic_vector(3 downto 0);
SIGNAL src_addr_hash: std_logic_vector(12 downto 0);
SIGNAL dest_addr_hash: std_logic_vector(12 downto 0);
SIGNAL returned_addr: std_logic_vector(47 downto 0);
SIGNAL returned_port: std_logic_vector(3 downto 0);
SIGNAL d_port_checked: std_logic_vector(3 downto 0);
SIGNAL done: std_logic;

SIGNAL counter:  integer range 4 downto 0 := 0;
```

```
SIGNAL last_check: integer range 4 downto 1;

BEGIN

MACRAM: MAC_RAM PORT MAP (
          clock  => clk,
          data => MACdata,
          rdaddress => MACrdaddress,
          wraddress => MACwraddress,
          wren => MACwren,
          q => MACq
        );

CRC16_MAP: CRC16 PORT MAP (
          clk  => clk,
          dataIn => CRCdata,
          done => crc_done,
          rst => crc_reset,
          crcOut => crc_out
        );

state_machine : PROCESS (state, req_1, req_2, req_3,req_4, s_port, CRCdata, clk, done)
        BEGIN



                if reset = '1' then

                        null;

                elsif (clk'event and clk = '1') then
                CASE state IS
         WHEN check_p1 =>

         if (req_1 = '1') then
                                src_addr <= addresses_1(47 downto 0);
                                dest_addr <= addresses_1(95 downto 48);
                                s_port <= s_port_1;
                                last_check <= 1;
                                state <= hashing_src;
                                crc_reset <= '1';
                        else
                                state <= check_p2;
                        end if;

                WHEN check_p2 =>

                        if (req_2 = '1') then
```

```
                src_addr <= addresses_2(47 downto 0);
                dest_addr <= addresses_2(95 downto 48);
                s_port <= s_port_2;
                last_check <= 2;
                state <= hashing_src;
                crc_reset <= '1';
        else
                state <= check_p3;
        end if;

WHEN check_p3 =>

        if (req_3 = '1') then
                src_addr <= addresses_3(47 downto 0);
                dest_addr <= addresses_3(95 downto 48);
                s_port <= s_port_3;
                last_check <= 3;
                state <= hashing_src;
                crc_reset <= '1';
        else
                state <= check_p4;
        end if;

WHEN check_p4 =>

        if (req_4 = '1') then
                src_addr <= addresses_4(47 downto 0);
                dest_addr <= addresses_4(95 downto 48);
                s_port <= s_port_4;
                last_check <= 4;
                state <= hashing_src;
                crc_reset <= '1';
        else
                state <= check_p1;
        end if;

WHEN hashing_src =>
        crc_reset <= '0';
        if crc_done = '1' then
                crc_reset <= '1';
                state <= hashing_dest;
        end if;

WHEN hashing_dest =>

        crc_reset <= '0';
        if crc_done = '1' then
                crc_reset <= '1';
                MACwren <= '1';
```

```
                        state <= table_processes;
                end if;

        WHEN table_processes=>

                if counter > 2 then
                        MACwren <= '0';
                        state <= compare_mac;
                end if;

        WHEN compare_mac=>
                state <= send_ack;

        WHEN send_ack=>
                if last_check = 1 then
                        state <= check_p2;
                elsif last_check = 2 then
                        state <= check_p3;
                elsif last_check = 3 then
                        state <= check_p4;
                elsif last_check = 4 then
                        state <= check_p1;
                end if;

        END CASE;
        end if;

END PROCESS;


process(clk, state)
begin
        if (clk'event and clk = '1') then
        CASE state IS

        WHEN hashing_src =>

                CRCdata <= src_addr;
                src_addr_hash <= crc_out(12 downto 0);

        WHEN hashing_dest =>

                CRCdata <= dest_addr;
                dest_addr_hash <= crc_out(12 downto 0);

        WHEN table_processes=>

                MACwraddress <= src_addr_hash;
                MACdata <= src_addr & s_port;
```

```
        MACrdaddress <= dest_addr_hash;

        returned_addr <= MACq(51 downto 4);
        returned_port <= MACq(3 downto 0);

        counter <= counter + 1;

WHEN compare_mac=>
        IF dest_addr = returned_addr THEN
                d_port_checked <= returned_port;

        ELSE
                d_port_checked <= "1111";

        END IF;

WHEN send_ack=>

        if s_port = "0001" then
                ack_1 <= '1';
                d_port_1 <= d_port_checked;
        elsif s_port = "0010" then
                ack_2 <= '1';
                d_port_2 <= d_port_checked;
        elsif s_port = "0100" then
                ack_3 <= '1';
                d_port_3 <= d_port_checked;
        elsif s_port = "1000" then
                ack_4 <= '1';
                d_port_4 <= d_port_checked;
        end if;


WHEN others =>
        ack_1 <= '0';
        ack_2 <= '0';
        ack_3 <= '0';
        ack_4 <= '0';

        counter <= 0;

        d_port_1 <= "UUUU";
        d_port_2 <= "UUUU";
        d_port_3 <= "UUUU";
        d_port_4 <= "UUUU";

END CASE;
end if;
```

```
        end process;



END arch;
```

15.c.2 CRC16

```
library ieee;
use ieee.std_logic_1164.all;

entity crc16 is
  port ( dataIn : in std_logic_vector (47 downto 0) := x"00_00_00_00_00_00";
    rst, clk : in std_logic;
    done: out std_logic;
    crcOut : out std_logic_vector (15 downto 0));
end crc16;

architecture imp_crc of crc16 is

signal data : std_logic_vector (47 downto 0) := x"00_00_00_00_00_00";
signal crcIn : std_logic_vector (15 downto 0) := x"00_00";
signal data_updated: std_logic := '0';

begin




    process (clk,rst) begin
        if (rst = '1') then

                crcOut <= "0000000000000000";
                data <= x"00_00_00_00_00_00";
                data_updated <= '0';
                done <= '0';

        elsif (clk'EVENT and clk = '1') then


                data <= dataIn;

                if data /= x"00_00_00_00_00_00" then
                        data_updated <= '1';
                end if;
                if data_updated = '1' then
                        crcOut(0) <= (crcIn(0) xor crcIn(1)
```

```
xor crcIn(3) xor crcIn(10)
xor data(0) xor data(4) xor data(8)
xor data(11) xor data(12)
xor data(19) xor data(20) xor data(22)
xor data(26) xor data(27)
xor data(28) xor data(32) xor data(33)
xor data(35) xor data(42));
crcOut(1) <= (crcIn(1) xor crcIn(2) xor
crcIn(4) xor crcIn(11)
xor data(1) xor data(5) xor data(9) xor
data(12) xor data(13)
xor data(20) xor data(21) xor data(23) xor
data(27) xor data(28)
xor data(29) xor data(33) xor data(34) xor
data(36) xor data(43));
crcOut(2) <= (crcIn(2) xor crcIn(3) xor
crcIn(5) xor crcIn(12)
xor data(2) xor data(6) xor data(10) xor
data(13) xor data(14)
xor data(21) xor data(22) xor data(24) xor
data(28) xor data(29)
xor data(30) xor data(34) xor data(35) xor
data(37) xor data(44));
crcOut(3) <= (crcIn(3) xor crcIn(4) xor
crcIn(6) xor crcIn(13)
xor data(3) xor data(7) xor data(11) xor
data(14) xor data(15)
xor data(22) xor data(23) xor data(25) xor
data(29) xor data(30)
xor data(31) xor data(35) xor data(36) xor
data(38) xor data(45));
crcOut(4) <= (crcIn(0) xor crcIn(4) xor
crcIn(5) xor crcIn(7)
xor crcIn(14) xor data(4) xor data(8) xor
data(12) xor data(15)
xor data(16) xor data(23) xor data(24) xor
data(26) xor data(30)
xor data(31) xor data(32) xor data(36) xor
data(37) xor data(39)
xor data(46));
crcOut(5) <= (crcIn(3) xor crcIn(5) xor
crcIn(6) xor crcIn(8)
xor crcIn(10) xor crcIn(15) xor data(0) xor
data(4) xor data(5)
xor data(8) xor data(9) xor data(11) xor
data(12) xor data(13)
xor data(16) xor data(17) xor data(19) xor
data(20) xor data(22)
xor data(24) xor data(25) xor data(26) xor
```

```
data(28) xor data(31)
xor data(35) xor data(37) xor data(38) xor
data(40) xor data(42)
xor data(47));
crcOut(6) <= (crcIn(0) xor crcIn(4) xor
crcIn(6) xor crcIn(7)
xor crcIn(9) xor crcIn(11) xor data(1) xor
data(5) xor data(6)
xor data(9) xor data(10) xor data(12) xor
data(13) xor data(14)
xor data(17) xor data(18) xor data(20) xor
data(21) xor data(23)
xor data(25) xor data(26) xor data(27) xor
data(29) xor data(32)
xor data(36) xor data(38) xor data(39) xor
data(41) xor data(43));
crcOut(7) <= (crcIn(1) xor crcIn(5) xor
crcIn(7) xor crcIn(8)
xor crcIn(10) xor crcIn(12) xor data(2) xor
data(6) xor data(7)
xor data(10) xor data(11) xor data(13) xor
data(14) xor data(15)
xor data(18) xor data(19) xor data(21) xor
data(22) xor data(24)
xor data(26) xor data(27) xor data(28) xor
data(30) xor data(33)
xor data(37) xor data(39) xor data(40) xor
data(42) xor data(44));
crcOut(8) <= (crcIn(2) xor crcIn(6) xor crcIn(8)
xor crcIn(9)
xor crcIn(11) xor crcIn(13) xor data(3) xor
data(7) xor data(8)
xor data(11) xor data(12) xor data(14) xor
data(15) xor data(16)
xor data(19) xor data(20) xor data(22) xor
data(23) xor data(25)
xor data(27) xor data(28) xor data(29) xor
data(31) xor data(34)
xor data(38) xor data(40) xor data(41) xor
data(43) xor data(45));
crcOut(9) <= (crcIn(0) xor crcIn(3) xor crcIn(7)
xor crcIn(9)
xor crcIn(10) xor crcIn(12) xor crcIn(14) xor
data(4) xor data(8)
xor data(9) xor data(12) xor data(13) xor
data(15) xor data(16)
xor data(17) xor data(20) xor data(21) xor
data(23) xor data(24)
xor data(26) xor data(28) xor data(29) xor
```

```
data(30) xor data(32)
xor data(35) xor data(39) xor data(41) xor
data(42) xor data(44)
xor data(46));
crcOut(10) <= (crcIn(1) xor crcIn(4) xor crcIn(8)
xor crcIn(10)
xor crcIn(11) xor crcIn(13) xor crcIn(15) xor
data(5) xor data(9)
xor data(10) xor data(13) xor data(14) xor data(16)
xor data(17)
xor data(18) xor data(21) xor data(22) xor data(24)
xor data(25)
xor data(27) xor data(29) xor data(30) xor data(31)
xor data(33)
xor data(36) xor data(40) xor data(42) xor data(43)
xor data(45)
xor data(47));
crcOut(11) <= (crcIn(0) xor crcIn(2) xor
crcIn(5) xor crcIn(9)
xor crcIn(11) xor crcIn(12) xor crcIn(14)
xor data(6) xor data(10)
xor data(11) xor data(14) xor data(15) xor
data(17) xor data(18)
xor data(19) xor data(22) xor data(23) xor
data(25) xor data(26)
xor data(28) xor data(30) xor data(31) xor
data(32) xor data(34)
xor data(37) xor data(41) xor data(43) xor
data(44) xor data(46));
crcOut(12) <= (crcIn(6) xor crcIn(12) xor
crcIn(13) xor crcIn(15)
xor data(0) xor data(4) xor data(7) xor
data(8) xor data(15) xor
data(16) xor data(18) xor data(22) xor
data(23) xor data(24) xor
data(28) xor data(29) xor data(31) xor
data(38) xor data(44) xor
data(45) xor data(47));
crcOut(13) <= (crcIn(0) xor crcIn(7) xor
crcIn(13) xor crcIn(14)
xor data(1) xor data(5) xor data(8) xor
data(9) xor data(16) xor
data(17) xor data(19) xor data(23) xor
data(24) xor data(25) xor
data(29) xor data(30) xor data(32) xor
data(39) xor data(45) xor
data(46));
crcOut(14) <= (crcIn(1) xor crcIn(8) xor
crcIn(14) xor crcIn(15)
```

```
                        xor data(2) xor data(6) xor data(9) xor
                        data(10) xor data(17)
                        xor data(18) xor data(20) xor data(24) xor
                        data(25) xor data(26)
                        xor data(30) xor data(31) xor data(33) xor
                        data(40) xor data(46)
                        xor data(47));
                        crcOut(15) <= (crcIn(0) xor crcIn(2) xor
                        crcIn(9) xor crcIn(15)
                        xor data(3) xor data(7) xor data(10) xor
                        data(11) xor data(18)
                        xor data(19) xor data(21) xor data(25) xor
                        data(26) xor data(27)
                        xor data(31) xor data(32) xor data(34) xor
                        data(41) xor data(47));
                        done <= '1';
                end if;

      end if;
    end process;

end architecture imp_crc;
```

15.c.3 MAC RAM

```
-- megafunction wizard: %RAM: 2-PORT%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: altsyncram

-- ============================================================
-- File Name: MAC_RAM.vhd
-- Megafunction Name(s):
--                      altsyncram
--
-- Simulation Library Files(s):
--                      altera_mf
-- ============================================================
-- ************************************************************
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
--
-- 20.1.1 Build 720 11/11/2020 SJ Lite Edition
-- ************************************************************


--Copyright (C) 2020  Intel Corporation. All rights reserved.
--Your use of Intel Corporation's design tools, logic functions
--and other software and tools, and any partner logic
--functions, and any output files from any of the foregoing
```

```
--(including device programming or simulation files), and any
--associated documentation or information are expressly subject
--to the terms and conditions of the Intel Program License
--Subscription Agreement, the Intel Quartus Prime License Agreement,
--the Intel FPGA IP License Agreement, or other applicable license
--agreement, including, without limitation, that your use is for
--the sole purpose of programming logic devices manufactured by
--Intel and sold by Intel or its authorized distributors.  Please
--refer to the applicable agreement for further details, at
--https://fpgasoftware.intel.com/eula.


LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY MAC_RAM IS
        PORT
        (
                clock           : IN STD_LOGIC  := '1';
                data            : IN STD_LOGIC_VECTOR (51 DOWNTO 0);
                rdaddress               : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                wraddress               : IN STD_LOGIC_VECTOR (12 DOWNTO 0);
                wren            : IN STD_LOGIC  := '0';
                q               : OUT STD_LOGIC_VECTOR (51 DOWNTO 0)
        );
END MAC_RAM;


ARCHITECTURE SYN OF mac_ram IS

        SIGNAL sub_wire0        : STD_LOGIC_VECTOR (51 DOWNTO 0);

BEGIN
        q    <= sub_wire0(51 DOWNTO 0);

        altsyncram_component : altsyncram
        GENERIC MAP (
                address_aclr_b => "NONE",
                address_reg_b => "CLOCK0",
                clock_enable_input_a => "BYPASS",
                clock_enable_input_b => "BYPASS",
                clock_enable_output_b => "BYPASS",
                intended_device_family => "Cyclone␣V",
                lpm_type => "altsyncram",
                numwords_a => 8000,
                numwords_b => 8000,
```

```
                operation_mode => "DUAL_PORT",
                outdata_aclr_b => "NONE",
                outdata_reg_b => "CLOCK0",
                power_up_uninitialized => "FALSE",
                read_during_write_mode_mixed_ports => "DONT_CARE",
                widthad_a => 13,
                widthad_b => 13,
                width_a => 52,
                width_b => 52,
                width_byteena_a => 1
        )
        PORT MAP (
                address_a => wraddress,
                address_b => rdaddress,
                clock0 => clock,
                data_a => data,
                wren_a => wren,
                q_b => sub_wire0
        );


END SYN;

-- ============================================================
-- CNX file retrieval info
-- ============================================================
-- Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
-- Retrieval info: PRIVATE: ADDRESSSTALL_B NUMERIC "0"
-- Retrieval info: PRIVATE: BYTEENA_ACLR_A NUMERIC "0"
-- Retrieval info: PRIVATE: BYTEENA_ACLR_B NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_ENABLE_A NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_ENABLE_B NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "1"
-- Retrieval info: PRIVATE: BlankMemory NUMERIC "1"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_B NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_B NUMERIC "0"
-- Retrieval info: PRIVATE: CLRdata NUMERIC "0"
-- Retrieval info: PRIVATE: CLRq NUMERIC "0"
-- Retrieval info: PRIVATE: CLRrdaddress NUMERIC "0"
-- Retrieval info: PRIVATE: CLRrren NUMERIC "0"
-- Retrieval info: PRIVATE: CLRwraddress NUMERIC "0"
-- Retrieval info: PRIVATE: CLRwren NUMERIC "0"
-- Retrieval info: PRIVATE: Clock NUMERIC "0"
-- Retrieval info: PRIVATE: Clock_A NUMERIC "0"
-- Retrieval info: PRIVATE: Clock_B NUMERIC "0"
-- Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
```

```
-- Retrieval info: PRIVATE: INDATA_ACLR_B NUMERIC "0"
-- Retrieval info: PRIVATE: INDATA_REG_B NUMERIC "0"
-- Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_B"
-- Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
-- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
-- Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
-- Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
-- Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
-- Retrieval info: PRIVATE: MEMSIZE NUMERIC "416000"
-- Retrieval info: PRIVATE: MEM_IN_BITS NUMERIC "0"
-- Retrieval info: PRIVATE: MIFfilename STRING ""
-- Retrieval info: PRIVATE: OPERATION_MODE NUMERIC "2"
-- Retrieval info: PRIVATE: OUTDATA_ACLR_B NUMERIC "0"
-- Retrieval info: PRIVATE: OUTDATA_REG_B NUMERIC "1"
-- Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "0"
-- Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_MIXED_PORTS NUMERIC "2"
-- Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
-- Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_B NUMERIC "3"
-- Retrieval info: PRIVATE: REGdata NUMERIC "1"
-- Retrieval info: PRIVATE: REGq NUMERIC "1"
-- Retrieval info: PRIVATE: REGrdaddress NUMERIC "1"
-- Retrieval info: PRIVATE: REGrren NUMERIC "1"
-- Retrieval info: PRIVATE: REGwraddress NUMERIC "1"
-- Retrieval info: PRIVATE: REGwren NUMERIC "1"
-- Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
-- Retrieval info: PRIVATE: USE_DIFF_CLKEN NUMERIC "0"
-- Retrieval info: PRIVATE: UseDPRAM NUMERIC "1"
-- Retrieval info: PRIVATE: VarWidth NUMERIC "1"
-- Retrieval info: PRIVATE: WIDTH_READ_A NUMERIC "52"
-- Retrieval info: PRIVATE: WIDTH_READ_B NUMERIC "52"
-- Retrieval info: PRIVATE: WIDTH_WRITE_A NUMERIC "52"
-- Retrieval info: PRIVATE: WIDTH_WRITE_B NUMERIC "52"
-- Retrieval info: PRIVATE: WRADDR_ACLR_B NUMERIC "0"
-- Retrieval info: PRIVATE: WRADDR_REG_B NUMERIC "0"
-- Retrieval info: PRIVATE: WRCTRL_ACLR_B NUMERIC "0"
-- Retrieval info: PRIVATE: enable NUMERIC "0"
-- Retrieval info: PRIVATE: rden NUMERIC "0"
-- Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
-- Retrieval info: CONSTANT: ADDRESS_ACLR_B STRING "NONE"
-- Retrieval info: CONSTANT: ADDRESS_REG_B STRING "CLOCK0"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_B STRING "BYPASS"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_B STRING "BYPASS"
-- Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone V"
-- Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
-- Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "8000"
-- Retrieval info: CONSTANT: NUMWORDS_B NUMERIC "8000"
-- Retrieval info: CONSTANT: OPERATION_MODE STRING "DUAL_PORT"
-- Retrieval info: CONSTANT: OUTDATA_ACLR_B STRING "NONE"
```

```
-- Retrieval info: CONSTANT: OUTDATA_REG_B STRING "CLOCK0"
-- Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
-- Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_MIXED_PORTS STRING "DONT_CARE"
-- Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "13"
-- Retrieval info: CONSTANT: WIDTHAD_B NUMERIC "13"
-- Retrieval info: CONSTANT: WIDTH_A NUMERIC "52"
-- Retrieval info: CONSTANT: WIDTH_B NUMERIC "52"
-- Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"
-- Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
-- Retrieval info: USED_PORT: data 0 0 52 0 INPUT NODEFVAL "data[51..0]"
-- Retrieval info: USED_PORT: q 0 0 52 0 OUTPUT NODEFVAL "q[51..0]"
-- Retrieval info: USED_PORT: rdaddress 0 0 13 0 INPUT NODEFVAL "rdaddress[12..0]"
-- Retrieval info: USED_PORT: wraddress 0 0 13 0 INPUT NODEFVAL "wraddress[12..0]"
-- Retrieval info: USED_PORT: wren 0 0 0 0 INPUT GND "wren"
-- Retrieval info: CONNECT: @address_a 0 0 13 0 wraddress 0 0 13 0
-- Retrieval info: CONNECT: @address_b 0 0 13 0 rdaddress 0 0 13 0
-- Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
-- Retrieval info: CONNECT: @data_a 0 0 52 0 data 0 0 52 0
-- Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
-- Retrieval info: CONNECT: q 0 0 52 0 @q_b 0 0 52 0
-- Retrieval info: GEN_FILE: TYPE_NORMAL MAC_RAM.vhd TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL MAC_RAM.inc FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL MAC_RAM.cmp TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL MAC_RAM.bsf FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL MAC_RAM_inst.vhd FALSE
-- Retrieval info: LIB_FILE: altera_mf
```

15.d Output Module

15.d.1 Data Demultiplexer

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.std_logic_unsigned.all;

entity demux_data_1 is
 port(

 data_in_demux : in std_logic_vector(8 downto 0);
 sel_demux: in std_logic_vector(3 downto 0);
 F_1,F_2,F_3: out std_logic_vector(7 downto 0);
 R_1,R_2,R_3: out std_logic
                );
end demux_data_1;

architecture demux_data_1_arch of demux_data_1 is
begin
process (data_in_demux,sel_demux)
begin
```

```
 if (sel_demux="0010") then
 F_1 <= data_in_demux(7 downto 0);
 R_1 <= '1';
 R_2 <= '0';
 R_3 <= '0';
 elsif (sel_demux="0100") then
 F_2 <= data_in_demux(7 downto 0);
 R_1 <= '0';
 R_2 <= '1';
 R_3 <= '0';
 elsif (sel_demux="1000") then
 F_3 <= data_in_demux(7 downto 0);
 R_1 <= '0';
 R_2 <= '0';
 R_3 <= '1';
 elsif (sel_demux="1111") then
 F_1 <= data_in_demux(7 downto 0);
 R_1 <= '1';
 F_2 <= data_in_demux(7 downto 0);
 R_2 <= '1';
 F_3 <= data_in_demux(7 downto 0);
 R_3 <= '1';
 else
 R_1 <= '0';
 R_2 <= '0';
 R_3 <= '0';
 end if;
end process;

end demux_data_1_arch;
```

15.d.2 Length Demultiplexer

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
 use IEEE.std_logic_unsigned.all;
entity demux_length_1 is
 port(

 data_length_in_demux : in std_logic_vector(10 downto 0);
 sel_length_demux: in std_logic_vector(3 downto 0);
 L_1,L_2,L_3: out std_logic_vector (10 downto 0);
 RL_1,RL_2,RL_3: out std_logic
                );
end demux_length_1;

architecture demux_length_1_arch of demux_length_1 is
begin
```

```
process (data_length_in_demux,sel_length_demux)
begin
 if (sel_length_demux="0010") then
 L_1 <= data_length_in_demux;
 RL_1 <= '1';
 RL_2 <= '0';
 RL_3 <= '0';
 elsif (sel_length_demux="0100") then
 L_2 <= data_length_in_demux;
 RL_1 <= '0';
 RL_2 <= '1';
 RL_3 <= '0';
 elsif (sel_length_demux="1000") then
 L_3 <= data_length_in_demux;
 RL_1 <= '0';
 RL_2 <= '0';
 RL_3 <= '1';
 elsif (sel_length_demux="1111") then
 L_1 <= data_length_in_demux;
 RL_1 <= '1';
 L_2 <= data_length_in_demux;
 RL_2 <= '1';
 L_3 <= data_length_in_demux;
 RL_3 <= '1';
 else
 RL_1 <= '0';
 RL_2 <= '0';
 RL_3 <= '0';
 end if;
end process;

end demux_length_1_arch;
```

15.d.3 Data FIFO

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY fifos IS
        PORT
        (
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                rdreq           : IN STD_LOGIC ;
                wrreq           : IN STD_LOGIC ;
```

```
                empty           : OUT STD_LOGIC ;
                full            : OUT STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
END fifos;


ARCHITECTURE SYN OF fifos IS

        SIGNAL sub_wire0        : STD_LOGIC ;
        SIGNAL sub_wire1        : STD_LOGIC ;
        SIGNAL sub_wire2        : STD_LOGIC_VECTOR (7 DOWNTO 0);



        COMPONENT scfifo
        GENERIC (
                add_ram_output_register         : STRING;
                intended_device_family          : STRING;
                lpm_numwords            : NATURAL;
                lpm_showahead           : STRING;
                lpm_type                : STRING;
                lpm_width               : NATURAL;
                lpm_widthu              : NATURAL;
                overflow_checking               : STRING;
                underflow_checking              : STRING;
                use_eab         : STRING
        );
        PORT (
                        clock   : IN STD_LOGIC ;
                        data    : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                        rdreq   : IN STD_LOGIC ;
                        wrreq   : IN STD_LOGIC ;
                        empty   : OUT STD_LOGIC ;
                        full    : OUT STD_LOGIC ;
                        q       : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
        );
        END COMPONENT;

BEGIN
        empty    <= sub_wire0;
        full    <= sub_wire1;
        q    <= sub_wire2(7 DOWNTO 0);

        scfifo_component : scfifo
        GENERIC MAP (
                add_ram_output_register => "OFF",
                intended_device_family => "Cyclone␣IV␣E",
                lpm_numwords => 2048,
```

```
                lpm_showahead => "OFF",
                lpm_type => "scfifo",
                lpm_width => 8,
                lpm_widthu => 11,
                overflow_checking => "ON",
                underflow_checking => "ON",
                use_eab => "ON"
        )
        PORT MAP (
                clock => clock,
                data => data,
                rdreq => rdreq,
                wrreq => wrreq,
                empty => sub_wire0,
                full => sub_wire1,
                q => sub_wire2
        );



END SYN;
```

15.d.4 Length FIFO

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.all;

ENTITY fifo_length IS
        PORT
        (
                clock           : IN STD_LOGIC ;
                data            : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
                rdreq           : IN STD_LOGIC ;
                wrreq           : IN STD_LOGIC ;
                empty           : OUT STD_LOGIC ;
                full            : OUT STD_LOGIC ;
                q               : OUT STD_LOGIC_VECTOR (10 DOWNTO 0)
        );
END fifo_length;


ARCHITECTURE SYN OF fifo_length IS

        SIGNAL sub_wire0        : STD_LOGIC ;
        SIGNAL sub_wire1        : STD_LOGIC ;
        SIGNAL sub_wire2        : STD_LOGIC_VECTOR (10 DOWNTO 0);
```

```
        COMPONENT scfifo
        GENERIC (
                add_ram_output_register         : STRING;
                intended_device_family          : STRING;
                lpm_numwords            : NATURAL;
                lpm_showahead           : STRING;
                lpm_type                : STRING;
                lpm_width               : NATURAL;
                lpm_widthu              : NATURAL;
                overflow_checking               : STRING;
                underflow_checking              : STRING;
                use_eab         : STRING
        );
        PORT (
                        clock   : IN STD_LOGIC ;
                        data    : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
                        rdreq   : IN STD_LOGIC ;
                        wrreq   : IN STD_LOGIC ;
                        empty   : OUT STD_LOGIC ;
                        full    : OUT STD_LOGIC ;
                        q       : OUT STD_LOGIC_VECTOR (10 DOWNTO 0)
        );
        END COMPONENT;

BEGIN
        empty    <= sub_wire0;
        full    <= sub_wire1;
        q    <= sub_wire2(10 DOWNTO 0);

        scfifo_component : scfifo
        GENERIC MAP (
                add_ram_output_register => "OFF",
                intended_device_family => "Cyclone␣IV␣E",
                lpm_numwords => 2048,
                lpm_showahead => "OFF",
                lpm_type => "scfifo",
                lpm_width => 11,
                lpm_widthu => 11,
                overflow_checking => "ON",
                underflow_checking => "ON",
                use_eab => "ON"
        )
        PORT MAP (
                clock => clock,
                data => data,
                rdreq => rdreq,
```

```
                wrreq => wrreq,
                empty => sub_wire0,
                full => sub_wire1,
                q => sub_wire2
        );



END SYN;
```

15.d.5 State Machine

```
library ieee;
use ieee.numeric_std.all;
use ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity state_machine is

port (
                reset : in  std_logic;
                clk : in  std_logic;
                emptyR_1 : in  std_logic;
                emptyR_2 : in  std_logic;
                emptyR_3 : in  std_logic;
                emptyR_1_L : in  std_logic;
                emptyR_2_L : in  std_logic;
                emptyR_3_L : in  std_logic;
                packet_length_1 : in std_logic_vector (10 downto 0);
                packet_length_2 : in std_logic_vector (10 downto 0);
                packet_length_3 : in std_logic_vector (10 downto 0);
                dataR_1 : in  std_logic_vector (7 downto 0);
                dataR_2 : in  std_logic_vector (7 downto 0);
                dataR_3 : in  std_logic_vector (7 downto 0);
                sm_data : out std_logic_vector (7 downto 0);
                reable_1 :out std_logic ;
                reable_2 :out std_logic ;
                reable_3 :out std_logic ;
                reable_1_L :out std_logic ;
                reable_2_L :out std_logic ;
                reable_3_L :out std_logic ;
                tx_flag : out std_logic
);
end entity ;

architecture state_machine_arch of state_machine is
constant quantum : std_logic_vector(10 downto 0) := "11000001110";
signal count_1 : std_logic_vector(10 downto 0):=(others=>'0');
signal count_2 : std_logic_vector(10 downto 0):=(others=>'0');
```

```
signal count_3 : std_logic_vector(10 downto 0):=(others=>'0');
signal defcount_1 : std_logic_vector(10 downto 0);
signal defcount_2 : std_logic_vector(10 downto 0);
signal defcount_3 : std_logic_vector(10 downto 0);
signal null_count_1 : integer range 10 downto 0:=0;
signal null_count_2 : integer range 10 downto 0:=0;
signal null_count_3 : integer range 10 downto 0:=0;
signal tx_flag_delay_1 : std_logic;
signal tx_flag_delay_2 : std_logic;
signal sm_data_delay_2 : std_logic_vector(7 downto 0):=(others=>'0');


TYPE STATE_TYPE IS (s0, s1, s2, s3, s0_buff, s1_idle,
s2_idle, s3_idle, s1_buff, s2_buff, s3_buff);
   SIGNAL state   : STATE_TYPE;
begin
process(clk)
begin
if rising_edge(clk) then
tx_flag_delay_1<=tx_flag_delay_2;
tx_flag<=tx_flag_delay_1;
end if;
end process;
process(clk,reset)
    begin
        if reset = '1' then
                defcount_1<= quantum;
                defcount_2<= quantum;
                defcount_3<= quantum;
                reable_1<= '0';
                reable_2<= '0';
                reable_3<= '0';
                reable_1_L<= '0';
                reable_2_L<= '0';
                reable_3_L<= '0';
                count_1<=(others=>'0');
                count_2<=(others=>'0');
                count_3<=(others=>'0');
                state <= s0;
                tx_flag_delay_2<='0';
                null_count_1<=0;
                null_count_2<=0;
                null_count_3<=0;
        elsif (clk'event and clk = '1') then
        case state is
        when s0=>
                if emptyR_1 = '1' and emptyR_2 = '1'
                and emptyR_3 = '1'  and emptyR_1_L = '1'
                and emptyR_2_L = '1'
```

```
         and emptyR_3_L = '1' then
                state<=s0_buff;
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<= '0';
                reable_2_L<= '0';
                reable_3_L<= '0';
        else
                state <= s1;
        end if;
when s0_buff=>
                tx_flag_delay_2<='0';
                state<=s0;
when s1=>
        if emptyR_1 = '1' or emptyR_1_L = '1' then
                state <= s2;
                defcount_1 <= defcount_1+quantum;
                count_1 <=(others=>'0');
        elsif defcount_1<= packet_length_1  then
                state<=s2;
                defcount_1 <= defcount_1+quantum;
                count_1 <= (others=>'0');
                null_count_1<=0;
        else
                reable_1_L<='1';
                reable_2_L<='0';
                reable_3_L<='0';
                state<=s1_buff;
        end if;
when s1_buff=>
        if count_1 < packet_length_1
        and  count_1 /= packet_length_1-"00000000001" then
                reable_1<='1';
                reable_2<='0';
                reable_3<='0';
                sm_data <= dataR_1;
                count_1<=count_1+'1';
                tx_flag_delay_2<= '1';
        elsif count_1 = packet_length_1-"00000000001" then
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                reable_1<='1';
                reable_2<='0';
                reable_3<='0';
                sm_data <= dataR_1;
                count_1<=count_1+'1';
                tx_flag_delay_2<= '1';
```

```
 elsif count_1 = packet_length_1 then
        defcount_1<=defcount_1-packet_length_1;
                count_1<=(others=>'0');
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                state<=s1_idle;
        end if;
when s1_idle=>
        if null_count_1 < 9 then
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                tx_flag_delay_2<='0';
                null_count_1<=null_count_1+1;
        else
                state<=s1;
                null_count_1<=0;
        end if;
when s2=>
        if emptyR_2 = '1' or emptyR_2_L = '1' then
                state<=s3;
                defcount_2 <= defcount_2+quantum;
                count_2<=(others=>'0');
        elsif defcount_2 <= packet_length_2  then
                state<=s3;
                defcount_2 <= defcount_2+quantum;
                count_2 <= (others=>'0');
                null_count_2<=0;
        else
                reable_1_L<='0';
                reable_2_L<='1';
                reable_3_L<='0';
                state<=s2_buff;
        end if;
when s2_buff=>
        if count_2 < packet_length_2
        and count_2 /= packet_length_2-"00000000001" then
                reable_1<='0';
                reable_2<='1';
                reable_3<='0';
                sm_data <= dataR_2;
                count_2<=count_2+'1';
```

```
                tx_flag_delay_2<= '1';
        elsif count_2 = packet_length_2-"00000000001" then
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                reable_1<='0';
                reable_2<='1';
                reable_3<='0';
                sm_data <= dataR_2;
                count_2<=count_2+'1';
                tx_flag_delay_2<= '1';
elsif count_2 = packet_length_2 then
                defcount_2<=defcount_2-packet_length_2;
                count_2<=(others=>'0');
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                state<=s2_idle;
        end if;
when s2_idle=>
        if null_count_2 < 9 then
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                tx_flag_delay_2<='0';
                null_count_2<=null_count_2+1;
        else
                state<=s2;
                null_count_2<=0;
        end if;
when s3=>
         if emptyR_3 = '1' or emptyR_3_L = '1' then
                state<=s0;
                defcount_3 <= defcount_3+quantum;
                count_3<=(others=>'0');
        elsif defcount_3 <= packet_length_3  then
                state<=s0;
                defcount_3 <= defcount_3+quantum;
                count_3 <= (others=>'0');
                null_count_3<=0;
        else
                reable_1_L<='0';
                reable_2_L<='0';
```

```
                reable_3_L<='1';
                state<=s3_buff;
            end if;
        when s3_buff=>
            if count_3 < packet_length_3
            and count_3 /= packet_length_3-"00000000001" then
                reable_1<='0';
                reable_2<='0';
                reable_3<='1';
                sm_data <= dataR_3;
                count_3<=count_3+'1';
                tx_flag_delay_2<= '1';
            elsif count_3 = packet_length_3-"00000000001" then
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                reable_1<='0';
                reable_2<='0';
                reable_3<='1';
                sm_data <= dataR_3;
                count_3<=count_3+'1';
                tx_flag_delay_2<= '1';
            elsif count_3 = packet_length_3 then
                defcount_3<=defcount_3-packet_length_3;
                count_3<=(others=>'0');
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                state<=s3_idle;
            end if;
        when s3_idle=>
            if null_count_3 < 9 then
                reable_1<='0';
                reable_2<='0';
                reable_3<='0';
                reable_1_L<='0';
                reable_2_L<='0';
                reable_3_L<='0';
                tx_flag_delay_2<='0';
                null_count_3<=null_count_3+1;
            else
                state<=s3;
                null_count_3<=0;
            end if;
        end case;
end process;
```

```
end architecture;
```

15.d.6 State Machine Controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;

entity state_machine_controller is

port (
                reset : in  std_logic;
                clk : in  std_logic;
                tx_1     : in std_logic;
                tx_2     : in std_logic;
                tx_3     : in std_logic;
                tx_4     : in std_logic;
                tx_data_1: in std_logic_vector(7 downto 0);
                tx_data_2: in std_logic_vector(7 downto 0);
                tx_data_3: in std_logic_vector(7 downto 0);
                tx_data_4: in std_logic_vector(7 downto 0);
                tx_data   : out std_logic_vector(31 downto 0);
                tx       : out std_logic_vector(3 downto 0)


);
end entity ;

architecture state_machine_controller_arch of state_machine_controller is
signal tx_1_flag_delay_3 : std_logic;
signal tx_1_flag_delay_2: std_logic;
signal tx_1_flag_delay_1 : std_logic;
signal tx_2_flag_delay_3 : std_logic;
signal tx_2_flag_delay_2: std_logic;
signal tx_2_flag_delay_1 : std_logic;
signal tx_3_flag_delay_3 : std_logic;
signal tx_3_flag_delay_2: std_logic;
signal tx_3_flag_delay_1 : std_logic;
signal tx_4_flag_delay_3 : std_logic;
signal tx_4_flag_delay_2: std_logic;
signal tx_4_flag_delay_1 : std_logic;
signal tx_data_1_sc : std_logic_vector(7 downto 0);
signal tx_data_2_sc : std_logic_vector(7 downto 0);
signal tx_data_3_sc : std_logic_vector(7 downto 0);
signal tx_data_4_sc : std_logic_vector(7 downto 0);
begin
process(clk)
begin
```

```
if rising_edge(clk) then
tx_1_flag_delay_2<=tx_1;
tx_1_flag_delay_3<=tx_1_flag_delay_2;
tx_2_flag_delay_2<=tx_2;
tx_2_flag_delay_3<=tx_2_flag_delay_2;
tx_3_flag_delay_2<=tx_3;
tx_3_flag_delay_3<=tx_3_flag_delay_2;
tx_4_flag_delay_2<=tx_4;
tx_4_flag_delay_3<=tx_4_flag_delay_2;
end if;
end process;
process (clk)
        begin
        if rising_edge(clk) then
                if tx_1='0' then
                        tx_data_1_sc<="00000000";
                elsif tx_1='1' then
                        tx_data_1_sc<=tx_data_1;
                end if;

                if tx_2='0' then
                        tx_data_2_sc<="00000000";
                elsif tx_2='1' then
                        tx_data_2_sc<=tx_data_2;
                end if;

                if tx_3='0' then
                        tx_data_3_sc<="00000000";
                elsif tx_3='1' then
                        tx_data_3_sc<=tx_data_3;
                end if;

                if tx_4='0' then
                        tx_data_4_sc<="00000000";
                elsif tx_4='1' then
                        tx_data_4_sc<=tx_data_4;
                end if;

                tx_data(31 downto 24)<=tx_data_4_sc;
                tx_data(23 downto 16)<=tx_data_3_sc;
                tx_data(15 downto 8)<=tx_data_2_sc;
                tx_data(7 downto 0)<=tx_data_1_sc;
                tx(3)<=tx_4_flag_delay_2;
                tx(2)<=tx_3_flag_delay_2;
                tx(1)<=tx_2_flag_delay_2;
                tx(0)<=tx_1_flag_delay_2;
                end if;
        end process;
end architecture;
```

15.d.7 Switch and Output Buffering Block

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use IEEE.std_logic_unsigned.all;
entity scheduling_buffering is

  port (
        clk             : in  std_logic;
        reset           : in  std_logic;
        data_1          : in  std_logic_vector(8 downto 0);
        data_2          : in  std_logic_vector(8 downto 0);
        data_3          : in  std_logic_vector(8 downto 0);
        data_4          : in  std_logic_vector(8 downto 0);
        port_1          : in std_logic_vector (3 downto 0);
        port_2          : in std_logic_vector (3 downto 0);
        port_3          : in std_logic_vector (3 downto 0);
        port_4          : in std_logic_vector (3 downto 0);
        length_1        : in std_logic_vector(10 downto 0);
        length_2        : in std_logic_vector(10 downto 0);
        length_3        : in std_logic_vector(10 downto 0);
        length_4        : in std_logic_vector(10 downto 0);
        tx              : out std_logic_vector(3 downto 0);
        tx_data         : out std_logic_vector(31 downto 0)
                  );

end scheduling_buffering;

architecture scheduling_buffering_arch of scheduling_buffering is

        component demux_data_1 is
                port(

                data_in_demux : in std_logic_vector(8 downto 0);
                sel_demux: in std_logic_vector(3 downto 0);
                F_1,F_2,F_3: out std_logic_vector(7 downto 0);
                R_1,R_2,R_3: out std_logic
                );
                end component;

        component demux_data_2 is
                port(

                data_in_demux : in std_logic_vector(8 downto 0);
                sel_demux: in std_logic_vector(3 downto 0);
                F_1,F_2,F_3: out std_logic_vector(7 downto 0);
                R_1,R_2,R_3: out std_logic
                );
                end component;
```

```
component demux_data_3 is
        port(

        data_in_demux : in std_logic_vector(8 downto 0);
        sel_demux: in std_logic_vector(3 downto 0);
        F_1,F_2,F_3: out std_logic_vector(7 downto 0);
        R_1,R_2,R_3: out std_logic
        );
        end component;

component demux_data_4 is
        port(

        data_in_demux : in std_logic_vector(8 downto 0);
        sel_demux: in std_logic_vector(3 downto 0);
        F_1,F_2,F_3: out std_logic_vector(7 downto 0);
        R_1,R_2,R_3: out std_logic
        );
        end component;

component fifos is
port(

        clock           : in std_logic ;
        data            : in std_logic_vector(7 downto 0);
        rdreq           : in std_logic;
        wrreq           : in std_logic ;
        empty           : out std_logic ;
        full            : out std_logic ;
        q               : out std_logic_vector(7 downto 0)
                );
end component;

component fifo_length is
port
(
        clock           : IN STD_LOGIC ;
        data            : IN STD_LOGIC_VECTOR (10 DOWNTO 0);
        rdreq           : IN STD_LOGIC ;
        wrreq           : IN STD_LOGIC ;
        empty           : OUT STD_LOGIC ;
        full            : OUT STD_LOGIC ;
        q               : OUT STD_LOGIC_VECTOR (10 DOWNTO 0)
);
end component;

component demux_length_1 is
port(
```

```
data_length_in_demux : in std_logic_vector(10 downto 0);
sel_length_demux: in std_logic_vector(3 downto 0);
L_1,L_2,L_3: out std_logic_vector (10 downto 0);
RL_1,RL_2,RL_3: out std_logic
        );
end component;

component demux_length_2 is
port(

data_length_in_demux : in std_logic_vector(10 downto 0);
sel_length_demux: in std_logic_vector(3 downto 0);
L_1,L_2,L_3: out std_logic_vector (10 downto 0);
RL_1,RL_2,RL_3: out std_logic
        );
end component;

component demux_length_3 is
port(

data_length_in_demux : in std_logic_vector(10 downto 0);
sel_length_demux: in std_logic_vector(3 downto 0);
L_1,L_2,L_3: out std_logic_vector (10 downto 0);
RL_1,RL_2,RL_3: out std_logic
        );
end component;

component demux_length_4 is
port(

data_length_in_demux : in std_logic_vector(10 downto 0);
sel_length_demux: in std_logic_vector(3 downto 0);
L_1,L_2,L_3: out std_logic_vector (10 downto 0);
RL_1,RL_2,RL_3: out std_logic
        );
end component;

component state_machine is

port (
         reset : in  std_logic;
         clk : in  std_logic;
         emptyR_1 : in  std_logic;
         emptyR_2 : in  std_logic;
         emptyR_3 : in  std_logic;
         emptyR_1_L : in  std_logic;
         emptyR_2_L : in  std_logic;
         emptyR_3_L : in  std_logic;
```

```
        packet_length_1 : in std_logic_vector(10 downto 0);
        packet_length_2 : in std_logic_vector(10 downto 0);
        packet_length_3 : in std_logic_vector(10 downto 0);
        dataR_1 : in  std_logic_vector (7 downto 0);
        dataR_2 : in  std_logic_vector (7 downto 0);
        dataR_3 : in  std_logic_vector (7 downto 0);
        sm_data : out std_logic_vector (7 downto 0);
        reable_1 :out std_logic ;
        reable_2 :out std_logic ;
        reable_3 :out std_logic ;
        reable_1_L :out std_logic ;
        reable_2_L :out std_logic ;
        reable_3_L :out std_logic ;
        tx_flag : out std_logic
              );

end component ;

component state_machine_controller is

port (
        reset : in  std_logic;
        clk : in  std_logic;
        tx_1     : in std_logic;
        tx_2     : in std_logic;
        tx_3     : in std_logic;
        tx_4     : in std_logic;
        tx_data_1: in std_logic_vector(7 downto 0);
        tx_data_2: in std_logic_vector(7 downto 0);
        tx_data_3: in std_logic_vector(7 downto 0);
        tx_data_4: in std_logic_vector(7 downto 0);
        tx_data   : out std_logic_vector(31 downto 0);
        tx       : out std_logic_vector(3 downto 0)


       );
end component ;

--demux_data_to_fifo
signal ddtf_1_2 :std_logic_vector (7 downto 0);
signal ddtf_1_3 :std_logic_vector (7 downto 0);
signal ddtf_1_4 :std_logic_vector (7 downto 0);
signal ddtf_2_1 :std_logic_vector (7 downto 0);
signal ddtf_2_3 :std_logic_vector (7 downto 0);
signal ddtf_2_4 :std_logic_vector (7 downto 0);
signal ddtf_3_1 :std_logic_vector (7 downto 0);
signal ddtf_3_2 :std_logic_vector (7 downto 0);
signal ddtf_3_4 :std_logic_vector (7 downto 0);
signal ddtf_4_1 :std_logic_vector (7 downto 0);
```

```
signal ddtf_4_2 :std_logic_vector (7 downto 0);
signal ddtf_4_3 :std_logic_vector (7 downto 0);

--demux_length_to_fifo
signal dltf_1_2 :std_logic_vector(10 downto 0);
signal dltf_1_3 :std_logic_vector(10 downto 0);
signal dltf_1_4 :std_logic_vector(10 downto 0);
signal dltf_2_1 :std_logic_vector(10 downto 0);
signal dltf_2_3 :std_logic_vector(10 downto 0);
signal dltf_2_4 :std_logic_vector(10 downto 0);
signal dltf_3_1 :std_logic_vector(10 downto 0);
signal dltf_3_2 :std_logic_vector(10 downto 0);
signal dltf_3_4 :std_logic_vector(10 downto 0);
signal dltf_4_1 :std_logic_vector(10 downto 0);
signal dltf_4_2 :std_logic_vector(10 downto 0);
signal dltf_4_3 :std_logic_vector(10 downto 0);

--fifos_read_req_signals
signal rdreq_1_2 :std_logic;
signal rdreq_1_3 :std_logic;
signal rdreq_1_4 :std_logic;
signal rdreq_2_1 :std_logic;
signal rdreq_2_3 :std_logic;
signal rdreq_2_4 :std_logic;
signal rdreq_3_1 :std_logic;
signal rdreq_3_2 :std_logic;
signal rdreq_3_4 :std_logic;
signal rdreq_4_1 :std_logic;
signal rdreq_4_2 :std_logic;
signal rdreq_4_3 :std_logic;

--fifos_write_req_signals
signal wrreq_1_2 :std_logic;
signal wrreq_1_3 :std_logic;
signal wrreq_1_4 :std_logic;
signal wrreq_2_1 :std_logic;
signal wrreq_2_3 :std_logic;
signal wrreq_2_4 :std_logic;
signal wrreq_3_1 :std_logic;
signal wrreq_3_2 :std_logic;
signal wrreq_3_4 :std_logic;
signal wrreq_4_1 :std_logic;
signal wrreq_4_2 :std_logic;
signal wrreq_4_3 :std_logic;

--fifos_empty_signals
signal empty_1_2 :std_logic;
signal empty_1_3 :std_logic;
signal empty_1_4 :std_logic;
```

```
signal empty_2_1 :std_logic;
signal empty_2_3 :std_logic;
signal empty_2_4 :std_logic;
signal empty_3_1 :std_logic;
signal empty_3_2 :std_logic;
signal empty_3_4 :std_logic;
signal empty_4_1 :std_logic;
signal empty_4_2 :std_logic;
signal empty_4_3 :std_logic;

--fifos_full_signals
signal full_1_2 :std_logic;
signal full_1_3 :std_logic;
signal full_1_4 :std_logic;
signal full_2_1 :std_logic;
signal full_2_3 :std_logic;
signal full_2_4 :std_logic;
signal full_3_1 :std_logic;
signal full_3_2 :std_logic;
signal full_3_4 :std_logic;
signal full_4_1 :std_logic;
signal full_4_2 :std_logic;
signal full_4_3 :std_logic;

--fifos_read_req_signals(length)
signal rdreq_1_2_L :std_logic;
signal rdreq_1_3_L :std_logic;
signal rdreq_1_4_L :std_logic;
signal rdreq_2_1_L :std_logic;
signal rdreq_2_3_L :std_logic;
signal rdreq_2_4_L :std_logic;
signal rdreq_3_1_L :std_logic;
signal rdreq_3_2_L :std_logic;
signal rdreq_3_4_L :std_logic;
signal rdreq_4_1_L :std_logic;
signal rdreq_4_2_L :std_logic;
signal rdreq_4_3_L :std_logic;

--fifos_write_req_signals(length)
signal wrreq_1_2_L :std_logic;
signal wrreq_1_3_L :std_logic;
signal wrreq_1_4_L :std_logic;
signal wrreq_2_1_L :std_logic;
signal wrreq_2_3_L :std_logic;
signal wrreq_2_4_L :std_logic;
signal wrreq_3_1_L :std_logic;
signal wrreq_3_2_L :std_logic;
signal wrreq_3_4_L :std_logic;
signal wrreq_4_1_L:std_logic;
```

```
signal wrreq_4_2_L :std_logic;
signal wrreq_4_3_L :std_logic;

--fifos_empty_signals(length)
signal empty_1_2_L :std_logic;
signal empty_1_3_L :std_logic;
signal empty_1_4_L :std_logic;
signal empty_2_1_L :std_logic;
signal empty_2_3_L :std_logic;
signal empty_2_4_L :std_logic;
signal empty_3_1_L :std_logic;
signal empty_3_2_L :std_logic;
signal empty_3_4_L :std_logic;
signal empty_4_1_L:std_logic;
signal empty_4_2_L:std_logic;
signal empty_4_3_L :std_logic;

--fifos_full_signals(length)
signal full_1_2_L :std_logic;
signal full_1_3_L :std_logic;
signal full_1_4_L :std_logic;
signal full_2_1_L :std_logic;
signal full_2_3_L :std_logic;
signal full_2_4_L :std_logic;
signal full_3_1_L :std_logic;
signal full_3_2_L :std_logic;
signal full_3_4_L :std_logic;
signal full_4_1_L :std_logic;
signal full_4_2_L :std_logic;
signal full_4_3_L :std_logic;

--data_to_state_machine
signal dtsm_1_2 :std_logic_vector (7 downto 0);
signal dtsm_1_3 :std_logic_vector (7 downto 0);
signal dtsm_1_4 :std_logic_vector (7 downto 0);
signal dtsm_2_1 :std_logic_vector (7 downto 0);
signal dtsm_2_3 :std_logic_vector (7 downto 0);
signal dtsm_2_4 :std_logic_vector (7 downto 0);
signal dtsm_3_1 :std_logic_vector (7 downto 0);
signal dtsm_3_2 :std_logic_vector (7 downto 0);
signal dtsm_3_4 :std_logic_vector (7 downto 0);
signal dtsm_4_1 :std_logic_vector (7 downto 0);
signal dtsm_4_2 :std_logic_vector (7 downto 0);
signal dtsm_4_3 :std_logic_vector (7 downto 0);

--length_to_state_machine
signal dlts_1_2 :std_logic_vector(10 downto 0);
signal dlts_1_3 :std_logic_vector(10 downto 0);
signal dlts_1_4 :std_logic_vector(10 downto 0);
```

```
signal dlts_2_1 :std_logic_vector(10 downto 0);
signal dlts_2_3 :std_logic_vector(10 downto 0);
signal dlts_2_4 :std_logic_vector(10 downto 0);
signal dlts_3_1 :std_logic_vector(10 downto 0);
signal dlts_3_2 :std_logic_vector(10 downto 0);
signal dlts_3_4 :std_logic_vector(10 downto 0);
signal dlts_4_1 :std_logic_vector(10 downto 0);
signal dlts_4_2 :std_logic_vector(10 downto 0);
signal dlts_4_3 :std_logic_vector(10 downto 0);

--state_machine_data
signal tx_data_1 : std_logic_vector(7 downto 0);
signal tx_data_2 : std_logic_vector(7 downto 0);
signal tx_data_3 : std_logic_vector(7 downto 0);
signal tx_data_4 : std_logic_vector(7 downto 0);

--tx_state_machine
signal tx_1 : std_logic;
signal tx_2 : std_logic;
signal tx_3 : std_logic;
signal tx_4 : std_logic;


begin

demux_data_control_1 : demux_data_1
port map (

        data_in_demux => data_1,
        sel_demux => port_1,
        F_1 => ddtf_2_1,
        F_2 => ddtf_3_1,
        F_3 => ddtf_4_1,
        R_1 => wrreq_2_1,
        R_2 => wrreq_3_1,
        R_3 => wrreq_4_1
               );


demux_data_control_2 : demux_data_2
port map (

        data_in_demux => data_2,
        sel_demux => port_2,
        F_1 => ddtf_1_2,
        F_2 => ddtf_3_2,
        F_3 => ddtf_4_2,
        R_1 => wrreq_1_2,
        R_2 => wrreq_3_2,
```

```
        R_3 => wrreq_4_2
                );

demux_data_control_3 : demux_data_3
port map (

        data_in_demux => data_3,
        sel_demux => port_3,
        F_1 => ddtf_1_3,
        F_2 => ddtf_2_3,
        F_3 => ddtf_4_3,
        R_1 => wrreq_1_3,
        R_2 => wrreq_2_3,
        R_3 => wrreq_4_3
                );

demux_data_control_4 : demux_data_4
port map (

        data_in_demux => data_4,
        sel_demux => port_4,
        F_1 => ddtf_1_4,
        F_2 => ddtf_2_4,
        F_3 => ddtf_3_4,
        R_1 => wrreq_1_4,
        R_2 => wrreq_2_4,
        R_3 => wrreq_3_4
                );

demux_length_control_1 : demux_length_1
port map (

        data_length_in_demux => length_1,
        sel_length_demux=> port_1,
        L_1=> dltf_2_1,
        L_2=> dltf_3_1,
        L_3=> dltf_4_1,
        RL_1=> wrreq_2_1_L,
        RL_2=> wrreq_3_1_L,
        RL_3=> wrreq_4_1_L
                );

demux_lentgh_control_2 : demux_length_2
port map (

        data_length_in_demux => length_2,
        sel_length_demux=> port_2,
        L_1=> dltf_1_2,
        L_2=> dltf_3_2,
```

```
        L_3=> dltf_4_2,
        RL_1=> wrreq_1_2_L,
        RL_2=> wrreq_3_2_L,
        RL_3=> wrreq_4_2_L
                );

demux_lentgh_control_3 : demux_length_3
port map (

        data_length_in_demux => length_3,
        sel_length_demux=> port_3,
        L_1=> dltf_1_3,
        L_2=> dltf_2_3,
        L_3=> dltf_4_3,
        RL_1=> wrreq_1_3_L,
        RL_2=> wrreq_2_3_L,
        RL_3=> wrreq_4_3_L
                );

demux_lentgh_control_4 : demux_length_4
port map (

        data_length_in_demux => length_4,
        sel_length_demux=> port_4,
        L_1=> dltf_1_4,
        L_2=> dltf_2_4,
        L_3=> dltf_3_4,
        RL_1=> wrreq_1_4_L,
        RL_2=> wrreq_2_4_L,
        RL_3=> wrreq_3_4_L
                );

fifo_1_2 :fifos
port map (

        clock => clk,
        data  => ddtf_1_2,
        rdreq => rdreq_1_2,
        wrreq => wrreq_1_2,
        empty => empty_1_2,
        full  => full_1_2,
        q     => dtsm_1_2
                );

fifo_1_3 :fifos
port map (

        clock => clk,
        data  => ddtf_1_3,
```

```
        rdreq => rdreq_1_3,
        wrreq => wrreq_1_3,
        empty => empty_1_3,
        full  => full_1_3,
        q     => dtsm_1_3
                 );

fifo_1_4 :fifos
port map (

        clock => clk,
        data  => ddtf_1_4,
        rdreq => rdreq_1_4,
        wrreq => wrreq_1_4,
        empty => empty_1_4,
        full  => full_1_4,
        q     => dtsm_1_4
                 );

fifo_2_1 :fifos
port map (

        clock => clk,
        data  => ddtf_2_1,
        rdreq => rdreq_2_1,
        wrreq => wrreq_2_1,
        empty => empty_2_1,
        full  => full_2_1,
        q     => dtsm_2_1
                 );

fifo_2_3 :fifos
port map (

        clock => clk,
        data  => ddtf_2_3,
        rdreq => rdreq_2_3,
        wrreq => wrreq_2_3,
        empty => empty_2_3,
        full  => full_2_3,
        q     => dtsm_2_3
                 );

fifo_2_4 :fifos
port map (

        clock => clk,
        data  => ddtf_2_4,
        rdreq => rdreq_2_4,
```

```
        wrreq => wrreq_2_4,
        empty => empty_2_4,
        full  => full_2_4,
        q     => dtsm_2_4
                 );

fifo_3_1 :fifos
port map (

        clock => clk,
        data  => ddtf_3_1,
        rdreq => rdreq_3_1,
        wrreq => wrreq_3_1,
        empty => empty_3_1,
        full  => full_3_1,
        q     => dtsm_3_1
                 );

fifo_3_2 :fifos
port map (

        clock => clk,
        data  => ddtf_3_2,
        rdreq => rdreq_3_2,
        wrreq => wrreq_3_2,
        empty => empty_3_2,
        full  => full_3_2,
        q     => dtsm_3_2
                 );

fifo_3_4 :fifos
port map (

        clock => clk,
        data  => ddtf_3_4,
        rdreq => rdreq_3_4,
        wrreq => wrreq_3_4,
        empty => empty_3_4,
        full  => full_3_4,
        q     => dtsm_3_4
                 );


fifo_4_1 :fifos
port map (

        clock => clk,
        data  => ddtf_4_1,
        rdreq => rdreq_4_1,
```

```
        wrreq => wrreq_4_1,
        empty => empty_4_1,
        full  => full_4_1,
        q     => dtsm_4_1
                 );

fifo_4_2 :fifos
port map (

        clock => clk,
        data  => ddtf_4_2,
        rdreq => rdreq_4_2,
        wrreq => wrreq_4_2,
        empty => empty_4_2,
        full  => full_4_2,
        q     => dtsm_4_2
                 );

fifo_4_3 :fifos
port map (

        clock => clk,
        data  => ddtf_4_3,
        rdreq => rdreq_4_3,
        wrreq => wrreq_4_3,
        empty => empty_4_3,
        full  => full_4_3,
        q     => dtsm_4_3
                 );

fifo_1_2_length :fifo_length
port map (

        clock => clk,
        data  => dltf_1_2,
        rdreq => rdreq_1_2_L,
        wrreq => wrreq_1_2_L,
        empty => empty_1_2_L,
        full  => full_1_2_L,
        q     => dlts_1_2
                 );

fifo_1_3_length :fifo_length
port map (

        clock => clk,
        data  => dltf_1_3,
        rdreq => rdreq_1_3_L,
        wrreq => wrreq_1_3_L,
```

```
        empty => empty_1_3_L,
        full  => full_1_3_L,
        q     => dlts_1_3
                 );

fifo_1_4_length :fifo_length
port map (

        clock => clk,
        data  => dltf_1_4,
        rdreq => rdreq_1_4_L,
        wrreq => wrreq_1_4_L,
        empty => empty_1_4_L,
        full  => full_1_4_L,
        q     => dlts_1_4
                 );

fifo_2_1_length :fifo_length
port map (

        clock => clk,
        data  => dltf_2_1,
        rdreq => rdreq_2_1_L,
        wrreq => wrreq_2_1_L,
        empty => empty_2_1_L,
        full  => full_2_1_L,
        q     => dlts_2_1
                 );

fifo_2_3_length :fifo_length
port map (

        clock => clk,
        data  => dltf_2_3,
        rdreq => rdreq_2_3_L,
        wrreq => wrreq_2_3_L,
        empty => empty_2_3_L,
        full  => full_2_3_L,
        q     => dlts_2_3
                 );

fifo_2_4_length :fifo_length
port map (

        clock => clk,
        data  => dltf_2_4,
        rdreq => rdreq_2_4_L,
        wrreq => wrreq_2_4_L,
        empty => empty_2_4_L,
```

```
        full  => full_2_4_L,
        q     => dlts_2_4
                 );

fifo_3_length :fifo_length
port map (

        clock => clk,
        data  => dltf_3_1,
        rdreq => rdreq_3_1_L,
        wrreq => wrreq_3_1_L,
        empty => empty_3_1_L,
        full  => full_3_1_L,
        q     => dlts_3_1
                 );

fifo_3_2_length :fifo_length
port map (

        clock => clk,
        data  => dltf_3_2,
        rdreq => rdreq_3_2_L,
        wrreq => wrreq_3_2_L,
        empty => empty_3_2_L,
        full  => full_3_2_L,
        q     => dlts_3_2
                 );

fifo_3_4_length :fifo_length
port map (

        clock => clk,
        data  => dltf_3_4,
        rdreq => rdreq_3_4_L,
        wrreq => wrreq_3_4_L,
        empty => empty_3_4_L,
        full  => full_3_4_L,
        q     => dlts_3_4
                 );


fifo_4_1_length :fifo_length
port map (

        clock => clk,
        data  => dltf_4_1,
        rdreq => rdreq_4_1_L,
        wrreq => wrreq_4_1_L,
        empty => empty_4_1_L,
```

```
        full  => full_4_1_L,
        q     => dlts_4_1
                 );

fifo_4_2_length :fifo_length
port map (

        clock => clk,
        data  => dltf_4_2,
        rdreq => rdreq_4_2_L,
        wrreq => wrreq_4_2_L,
        empty => empty_4_2_L,
        full  => full_4_2_L,
        q     => dlts_4_2
                 );

fifo_4_3_length :fifo_length
port map (

        clock => clk,
        data  => dltf_4_3,
        rdreq => rdreq_4_3_L,
        wrreq => wrreq_4_3_L,
        empty => empty_4_3_L,
        full  => full_4_3_L,
        q     => dlts_4_3
                 );

state_machine_port_1 : state_machine
port map (

 reset           => reset,
 clk             => clk,
 emptyR_1        => empty_1_2,
 emptyR_2        => empty_1_3,
 emptyR_3        => empty_1_4,
 emptyR_1_L        => empty_1_2_L,
 emptyR_2_L        => empty_1_3_L,
 emptyR_3_L        => empty_1_4_L,
 packet_length_1 => dlts_1_2,
 packet_length_2 => dlts_1_3,
 packet_length_3 => dlts_1_4,
 dataR_1         => dtsm_1_2,
 dataR_2         => dtsm_1_3,
 dataR_3         => dtsm_1_4,
 sm_data         => tx_data_1,
 reable_1        => rdreq_1_2,
 reable_2        => rdreq_1_3,
 reable_3        => rdreq_1_4,
```

```
 reable_1_L        => rdreq_1_2_L,
 reable_2_L        => rdreq_1_3_L,
 reable_3_L        => rdreq_1_4_L,
 tx_flag            => tx_1
                       );

state_machine_port_2 : state_machine
port map (

 reset           => reset,
 clk             => clk,
 emptyR_1        => empty_2_1,
 emptyR_2        => empty_2_3,
 emptyR_3        => empty_2_4,
 emptyR_1_L        => empty_2_1_L,
 emptyR_2_L        => empty_2_3_L,
 emptyR_3_L        => empty_2_4_L,
 packet_length_1 => dlts_2_1,
 packet_length_2 => dlts_2_3,
 packet_length_3 => dlts_2_4,
 dataR_1         => dtsm_2_1,
 dataR_2         => dtsm_2_3,
 dataR_3         => dtsm_2_4,
 sm_data         => tx_data_2,
 reable_1        => rdreq_2_1,
 reable_2        => rdreq_2_3,
 reable_3        => rdreq_2_4,
 reable_1_L        => rdreq_2_1_L,
 reable_2_L        => rdreq_2_3_L,
 reable_3_L        => rdreq_2_4_L,
 tx_flag         => tx_2
                       );

 state_machine_port_3 : state_machine
 port map (

 reset           => reset,
 clk             => clk,
 emptyR_1        => empty_3_1,
 emptyR_2        => empty_3_2,
 emptyR_3        => empty_3_4,
 emptyR_1_L        => empty_3_1_L,
 emptyR_2_L        => empty_3_2_L,
 emptyR_3_L        => empty_3_4_L,
 packet_length_1 => dlts_3_1,
 packet_length_2 => dlts_3_2,
 packet_length_3 => dlts_3_4,
 dataR_1         => dtsm_3_1,
 dataR_2         => dtsm_3_2,
```

```
 dataR_3          => dtsm_3_4,
 sm_data          => tx_data_3,
 reable_1         => rdreq_3_1,
 reable_2         => rdreq_3_2,
 reable_3         => rdreq_3_4,
 reable_1_L         => rdreq_3_1_L,
 reable_2_L         => rdreq_3_2_L,
 reable_3_L         => rdreq_3_4_L,
 tx_flag          => tx_3
                       );

state_machine_port_4 : state_machine
port map (

 reset            => reset,
 clk              => clk,
 emptyR_1         => empty_4_1,
 emptyR_2         => empty_4_2,
 emptyR_3         => empty_4_3,
 emptyR_1_L         => empty_4_1_L,
 emptyR_2_L         => empty_4_2_L,
 emptyR_3_L         => empty_4_3_L,
 packet_length_1  => dlts_4_1,
 packet_length_2  => dlts_4_2,
 packet_length_3  => dlts_4_3,
 dataR_1          => dtsm_4_1,
 dataR_2          => dtsm_4_2,
 dataR_3          => dtsm_4_3,
 sm_data          => tx_data_4,
 reable_1         => rdreq_4_1,
 reable_2         => rdreq_4_2,
 reable_3         => rdreq_4_3,
 reable_1_L         => rdreq_4_1_L,
 reable_2_L         => rdreq_4_2_L,
 reable_3_L         => rdreq_4_3_L,
 tx_flag          => tx_4
               );

state_machines_control : state_machine_controller
port map (

reset            => reset,
clk              => clk,
tx_1             => tx_1,
tx_2             => tx_2,
tx_3             => tx_3,
tx_4             => tx_4,
tx_data_1        => tx_data_1,
tx_data_2        => tx_data_2,
```

```
        tx_data_3       => tx_data_3,
        tx_data_4       => tx_data_4,
        tx_data         => tx_data,
        tx              => tx
                        );

end scheduling_buffering_arch;
```