NTNU – Trondheim
Norwegian University of
Science and Technology

Implementing Open flow switch using FPGA based platform

Ting Liu

Master of Telematics - Communication Networks and Networked Services (2
Submission date: June 2014
Supervisor: Yuming Jiang, ITEM
Co-supervisor: Ameen Chilwan, ITEM
Kashif Mahmood, Telenor

Norwegian University of Science and Technology
Department of Telematics

Title: Implementing OpenFlow Switch using FPGA based platform

Student: Ting Liu

Problem description:

Network infrastructure has become critical in our schools, homes and business. However, current network architecture is static and unprogrammable. Recently, SDN (Software-Defined Networking) is appealed to make network programmable. OpenFlow is a typical protocol of SDN, which has gained attention because of its flexibility in managing networks. Control plane and data plane are separated in OpenFlow Switch. The intelligence of the network is OpenFlow controller and the traffic forwarding is done in the data plane based on the input from the control plane. One of OpenFlow benefits is that researchers and developers can develop intelligent new service rapidly and independently without waiting for new features to be released from equipment vendors. Additionally, OpenFlow Switch has already been implemented on NetFPGA [1]. This implementation has high latency to insert a new flow into OpenFlow Switch, which is still bottleneck. However the implementation of distributed multimedia plays (DMP) network nodes indicates lower latency and scalability features on FPGA based platform [2]. Therefore, this project is motivated to implement OpenFlow Switch (data plane and control plane) using FPGA based platform (Xilinx Virtex6) and also to analyse the performance to figure out whether it is better than current implemented OpenFlow Switch.

Planned tasks:
- Reviewing literatures about SDN (Software-Defined Networking) and OpenFlow Switch
- Learning the FPGA-based platform for SDN (OpenFlow Switch), especially hardware architecture in NTNU (Xilinx Virtex6)
- Implementing OpenFlow Switch (data plane and control plane) by VHDL using FPGA based platform (Xilinx Virtex6)
- Performance analysis of OpenFlow Switch implemented on FPGA based platform(Xilinx Virtex6) (e.g, delay, latency, loss)

Responsible professor: Yuming Jiang, Telematics

Supervisor: Ameen Chilwan, Telematics

External Collaborator: Kashif Mahmood, Telenor

Abstract

OpenFlow based SDN, is currently implemented in various networking devices and software, providing high-performance and granular traffic control across multiple vendors network devices. OpenFlow, as the first standard interface designed specifically for SDN, has gained popularity with both academic researchers and industry as a framework for both network research and implementation. OpenFlow technology separates the Control Plane from the Data Path and this allows the network managers to develop their own algorithms to control data flows and packets. Several vendors have already added OpenFlow to their features such as HP Labs, Cisco researchers, NEC, etc. Currently, OpenFlow Switch is already implemented on several different platforms e.g, in software (Linux, OpenWRT) and hardware (NetFPGA). More and more researchers implement the switch on FPGA-based platform, because FPGA-based platform is flexible, fast and reprogrammable. However, there are limited number of studies about the performance of the OpenFlow switch, which motivates this project. In order to do the research of OpenFlow performance, the simulation model of OpenFlow system is implemented in this project. The main objective of this project has two sides. On one hand, it is to implement OpenFlow system (switch and controller) using a hardware language on FPGA-based platform. On the other hand, it is also to measure the performance metrics of the OpenFlow switch, especially the service time (switch and controller) and the sojourn time. More specifically, data plane and control plane are both implemented on FPGA-based platform. It is designed in VHDL language by ISE design tools. FPGA-platform is Virtex6 type from Xilinx. It is observed from the results that the service time and the sojourn time both have almost linear increase with the increase in payload size. Moreover, the results indicate that the switch takes 2 clock cycles to respond to the writing request of the controller.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

1.1 Background and Motivation

Network infrastructure has become critical in the Internet and enterprise network. However, with the explosion of mobile devices and the rise of cloud services but with limited available bandwidth, network architecture has become complex which results in that current network capacity can not match users' requirements. Networking technologies exert limitations such as complexity, inconsistent policies, inability to scale and vendor dependence, which can't satisfy high requirements of network architecture in enterprises, homes and schools [7]. At the same time, changing traffic patterns, "IT consumerization", the rise of cloud services and bandwidth limitation trigger the need of new network architecture [7]. Moreover, some network vendors are unhappy that researchers run experiments or test new protocols in their Internet environment, because it may lower or interrupt production traffic. Thus, the network innovation is needed to satisfy more users' requirements and also to optimize the current network.

Recently, Software-Defined Networking (SDN) created by Open Networking Foundation (ONF) attracted many academic researchers and vendors. ONF, a non-profit organization, is responsible to control and publish the different OpenFlow specifications and gives the trademark license "OpenFlow Switching" to companies that adopt this standard. OpenFlow is a new technology based on the SDN concept where is the switch that decides the actions that have to do. OpenFlow technology separates the control plane from the data path and this allows network managers to develop their own algorithms to control data flows and packets, resulting in more efficient network management, more flexibility in response to demands and faster innovation [7]. Furthermore, it implements the control logic on an external controller (typically an external PC) and this controller is responsible for deciding the actions that the switch must perform. The communication between the controller and the data path is made on the network itself, using the protocol that provides OpenFlow (OpenFlow

Protocol) via Secure Socket Layer (SSL) [7]. Now the researchers are not required to wait for new features to be released from equipment vendors and they can develop intelligent new services rapidly and independently in multiple-vendor environment [8]. Thus, OpenFlow has gained popularity with both academic researchers and industry as a framework for both network research and implementation, due to its advantages (decouple data and controller path, and routing intelligence).

Actually, OpenFlow switch has already been implemented on several different platforms (Linux, OpenWRT and NetFPGA). There are many studies about the OpenFlow switch implementation on FPGA-based platform. Because FPGA-based switches are open-source hardware, which are faster than software-based switch. Besides, FPGA hardware is reprogrammable so that researchers can develop their own OpenFlow switches. And OpenFlow switch prototypes on FPGA-based platform can forward packets at 1-10Gbps. Thus, More and more people care about the OpenFlow switch implementation on FPGA-based platform and try to improve the OpenFlow switch.However, only a few works on the performance analysis of the OpenFlow switch are done. The implementation of OpenFlow Switch on NetFPGA has high latency to insert a new flow into OpenFlow Switch, which is still bottleneck [1]. However, the implementation of distributed multimedia plays (DMP) network nodes indicates lower latency and scalability features on FPGA based platform [2]. Therefore, this project is motivated to implement simulation model of OpenFlow system (data plane and control plane) on FPGA-based platform and also measure the performance metrics of the OpenFlow switch. However, our switch is designed only for research, not for the market. It is observed from the results that the service time and the sojourn time both have almost linear increase with the increase in payload size. Moreover, the results indicate that the switch takes only 2 clock cycles to respond to the writing request of the controller, which decreases the communication time between the switch and the controller.

1.2 Problem Statement

Current network architecture is static and non-programmable. Recently, SDN is appealed to make network programmable. OpenFlow is a typical protocol for SDN, which has gained attention because of its flexibility in managing networks. Many OpenFlow switch implementations have already been done on different platforms, but few works of performance analysis are available. Thus, this project is meant to implement both data plane and control plane on FPGA in order to do the performance simulation.

1.3 Objectives

The main objective of this project has two sides. On one hand, it is to implement OpenFlow Switch (data plane and control plane) using hardware language on FPGA-based platform. The OpenFlow specification v1.1 [3] is implemented in our switch. More specifically, the OpenFlow switch and controller are both implemented on FPGA due to lack of enough Ethernet ports and PCIe communication problem. On the other hand, the performance metrics of the OpenFlow switch are measured, especially the service time (switch and controller) and the sojourn time.

1.4 Methodology

It is implemented on hardware using Xilinx Virtex6 which is plugged into a Linux PC. Additionally, it is designed in VHDL language using ISE design tool. More specifically, flow table entry composer module, flow table controller, action processor, controller policy are implemented on FPGA. Besides, the analysis of the performance metrics is done from the data obtained from the hardware. In order to measure metrics, the method used is to generate packets with different sizes.

1.5 Outline

In Chapter 2, the theoretical background (e.g, SDN, OpenFlow Switch, OpenFlow controller, FPGA-based platform) is introduced in details and previous related works are described briefly. Chapter 3 depicts the details of the OpenFlow switch components (i.e, flow table entry composer module, flow table controller module, action processor module) and the controller (controller policy module). Chapter 4 is dedicated to show the results of performance simulation and comparison of resource utilization with other OpenFlow switches implemented on other platforms . Chapter 5 gives a conclusion, and highlights limitations and suggestions for future research.

Chapter 2
Theoretical Background and Related Work

In this chapter, related theoretical concepts such as software-defined networking (SDN), OpenFlow switch, OpenFlow controller and FPGA-based platform are described in details. It also gives the overview of previous related works.

2.1 Software-defined Networking (SDN)

Current network architecture is static and non-programmable. Network designers can not add new services into current network architecture arbitrarily because of the limitation of networking technologies (e.g, complexity, inconsistent policies, inability to scale and vendor dependence) [7]. Vendor dependence is the main reason of creating Software-Defined Networking (SDN). Recently, SDN has become popular in both academia and industry. Because of it, researchers and developers develop a new service or a new protocol without waiting many years for new features to be released from equipment vendors. More details of SDN can be found in [7].

SDN, created by non-profit ONF (open networking foundation) solved the problem, resulting in more efficient network management, more flexibility in response to demands and faster innovation. SDN in [7] is defined as "an emerging network architecture where network control is decoupled from forwarding and is directly programmable". The main characteristics of SDN include

- Control and data planes are decoupled and abstracted from each other
- Intelligence is logically centralized, resulting in having a global view of network and changing demands
- Underlying network infrastructure abstracted from applications, which makes it possible to develop different applications
- Programmable data plane brings automation and flexibility to networks

- Faster innovation [9]

Besides, SDN simplifies the network design and operations. For example, researchers and network vendors can program the network without disrupting production traffic and also develop new services rapidly and independently. Moreover, the flexibility of SDN allows network managers to configure, manage, secure, and optimize network resources automatically [7]. With SDN, the static network can evolve into an extensible service delivery platform capable of responding rapidly to changing business, end-user, and market needs. Thus, a variety of networking devices and software currently have adapted OpenFlow-based SDN which delivers substantial benefits to both enterprises and carriers, including

- Centralized management and control
- Improved automation and management
- Rapid innovation
- Programmability
- Increased network reliability and security
- More granular network control
- Better end-user experience [7].

2.2 Advantages of OpenFlow-based SDN

In [10], many advantages of SDN for network administrators are indicated. Firstly, network administrations expand SDN to the network, so network resources can be shared safely by multiple groups of users [10]. Secondly, through SDN, administrators can easily maintain entire virtual networks with their associated compute and storage resources even when VMs are migrated to different hosts [10]. Thirdly, with SDN, administrators can implement load-balancing with an OpenFlow switch and a commodity server [10]. This high cost-effective solution lets administrators better control traffic flow throughout the network to improve network performance. In addition, because administrators strive to expand the benefits of server and storage virtualization to the network, they are limited by the physical network infrastructure itself. However, a virtualized OpenFlow network removes these limitations, allowing administrators to create a flow-based virtual network abstraction that expands the benefits of virtualization to the network level.

2.3 OpenFlow Architecture

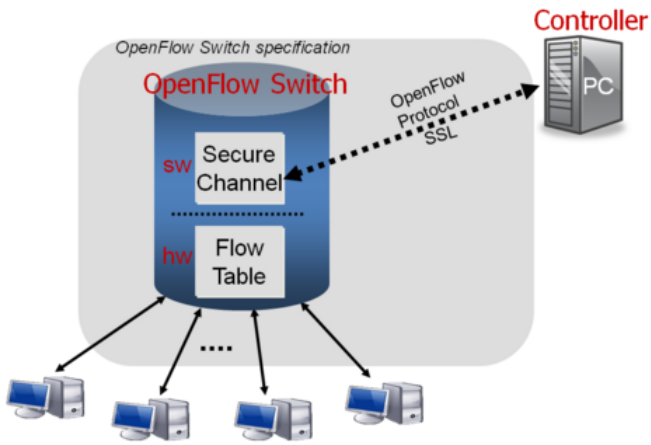

Figure 2.1: OpenFlow Switch [3]

OpenFlow, as the first standard interface for SDN, has gained popularity within both academia and industry as a framework for both network research and implementation. It provides high-performance and granular traffic control across multiple vendors network devices. Flexibility is the key advantages of OpenFlow compared to existing protocols such as IP and Ethernet. Generally, using OpenFlow results in the following advantages: network virtualization and route distribution [11].

The communication procedure between OpenFlow switch and OpenFlow controller is described briefly here. The OpenFlow switch mainly consists of two flow tables (exact match table and wildcard table) and an interface for modifying flow table entries (e.g, adding, deleting) [3]. The OpenFlow controller decides the path of new packet (unmatched packet). Figure 2.1 describes the OpenFlow Switch briefly. The controller connects to OpenFlow switch via Secure Socket Layer (SSL) and modifies flow table entries through interface. The communication procedure between them is easily understood. For example, unmatched packet is encapsulated and sent to the controller over SSL. Then controller examines it, updates flow table entries and sends it back to the switch. The next arriving packet belonging to the same flow is then forwarded through the switch without consulting the controller. Several vendors have already added OpenFlow to their features such as HP Labs, Cisco researchers, NEC etc. More information about the OpenFlow architecture is described in the OpenFlow

Standard[3]. The OpenFlow switch and OpenFlow controller are introduced in detail continually in the following subsection.

2.3.1 OpenFlow Controller

The controller is the main device, responsible for maintaining all of the network rules and distributing the appropriate instructions for the network devices. In other words, the OpenFlow controller is responsible for determining how to handle packets without valid flow entries, and it manages the switch flow table by adding and removing flow entries over the secure channel using the OpenFlow protocol. The controller essentially centralizes the network intelligence, while the network maintains a distributed forwarding plane through OpenFlow switches and routers. For this reason the controller provides an interface to manage, control and administrate the switch's flow tables. Because the network control plane is implemented in software, rather than the firmware of hardware devices, network traffic can be managed more dynamically and at a much more granular level [4]. The controller is able to program a switch through reactive behaviour and proactive behaviour, shown in Figure 2.2 and Figure 2.3. The reactive behaviour takes advantage of the flow table efficiently. In other words, the first packet of flow triggers controller to insert flow entries and the switch limits the utility if control connection lost [12]. While proactive behaviour means that the controller pre-populates flow table in switch and loss of control connection does not disrupt traffic [12]. More information about the OpenFlow controller can be found in the OpenFlow Standard [3]. There are different controller implementations available today, shown in the following Table 2.1.

Table 2.1: The top 5 Controllers available today and the main features

Controllers	**Main characteristic**
Beacon	A fast, cross-platform, modular, Java-based controller
	supporting both event-based and threaded operation
NOX	Open-source, a simplified platform written in C++ or Python
Trema	Full-Stack OpenFlow Framework for Ruby/C
Maestro	scalable, written in Java which supports OpenFlow switches
SNAC	using a web-based policy manager to manage the network

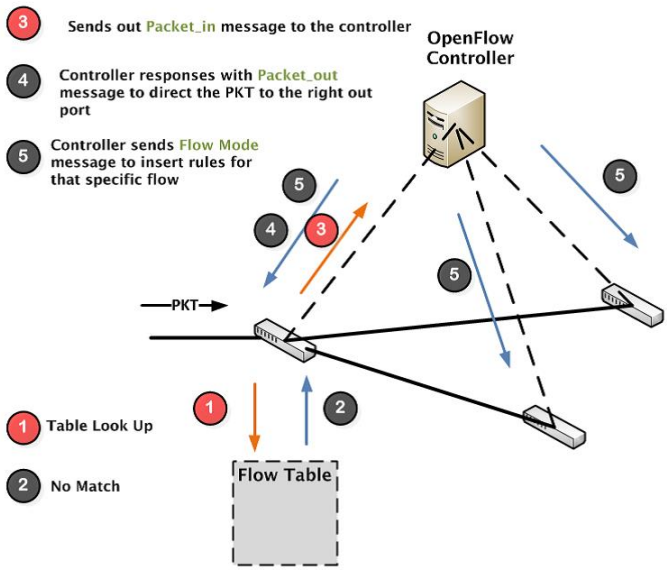

Figure 2.2: OpenFlow Reactive Module [4]

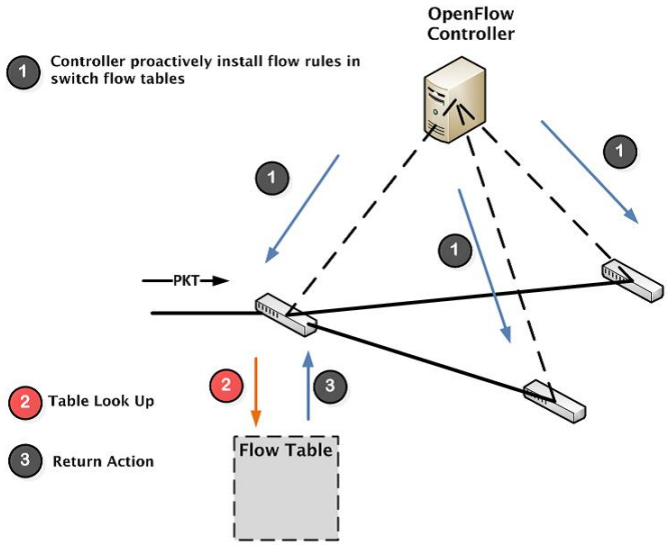

Figure 2.3: OpenFlow Proactive Module [4]

2.3.2 OpenFlow Switch

The theory of OpenFlow switch is introduced briefly here. As shown in Figure 2.1, OpenFlow switch mainly consists of three parts: OpenFlow table, OpenFlow secure channel and OpenFlow protocol [3]. Packets are forwarded based on flow tables and controller can modify these flow tables via secure channel using OpenFlow protocol. The flow tables consist of a set of flow entries and each flow entry is associated with actions [3]. When OpenFlow switch receives a packet, it looks up the flow table (comparing received packet header with entries of the flow tables). If the packet header matches the flow table, associated actions are executed. According to the OpenFlow specification [3], actions include packet forwarding, packet modification and addition, removing packet header, dropping packet etc. On the other hand, if the packet doesn't match, it is transmitted to controller and the controller builds a new flow table. More information about the OpenFlow switch is explained in the OpenFlow Standard [3]. The details of OpenFlow components are described in the following section.

2.4 Components of OpenFlow Switch

This section explains three components of OpenFlow Switch: OpenFlow protocol, OpenFlow flow tables and OpenFlow channel.

2.4.1 OpenFlow protocol

Three message types are defined in the OpenFlow protocol: *controller-to-switch*, *asynchronous* and *symmetric* [3]. Symmetric messages (see Table 2.2) are used to keep connection between the controller and the switch. Asynchronous messages (see Table 2.3) are sent from the switch to the controller to denote a packet arrival, switch state change, or error [3]. While controller-to-switch message (see Table 2.4) is sent from controller to switch. Controller can manage and modify the state of OpenFlow switch through those messages.

Table 2.2: Symmetric messages

Symmetric messages	Description
Hello	Exchanged upon connection startup
Echo	Request/reply messages from the switch or the controller
	Measures the latency or bandwidth of a connection
Experimenter	Offer additional functionality

Table 2.3: Asynchronous messages

Asynchronous messages	Description
Packet-in	Sent to the controller for unmatched packets
Flow-Removed	Remove as an idle timeout or a hard timeout occurs
Port-status	Send to the controller as port state changes
Error	Notify the controller of problems

Table 2.4: Controller-to-switch messages

Controller-to-switch messages	Description
Features	Query capabilities of a switch
Configurations	Set and Query configuration parameters
Modify-State	Add/delete and modify flows/groups table
	Set switch port properties
Read-State	Collect statistics
Packet-out	Send packets out of a specified port
	Forward packets received via Packet-in
Barrier	Ensure message dependencies

2.4.2 OpenFlow flow tables

This subsection introduces components of OpenFlow tables, along with the mechanics of matching and action handling. OpenFlow switch has two flow tables: exact match table and wildcard match table. Each flow table includes many flow entries. Main components of a flow entry in a flow table include the match fields (matching against packets), counters (updating for matching packets) and instructions (modifying action set) [3]. Packet flow through the pipeline processing is shown in Figure 2.4. The incoming packet is looked up orderly through each flow table. If the packet matches a flow entry, pipeline processing stops and the corresponding action is executed. If the packet does not match, the default is to send the packet to the controller. In our design, two match fields associated with actions are designed (exact match table and wild card table), which is described further in Chapter 3.

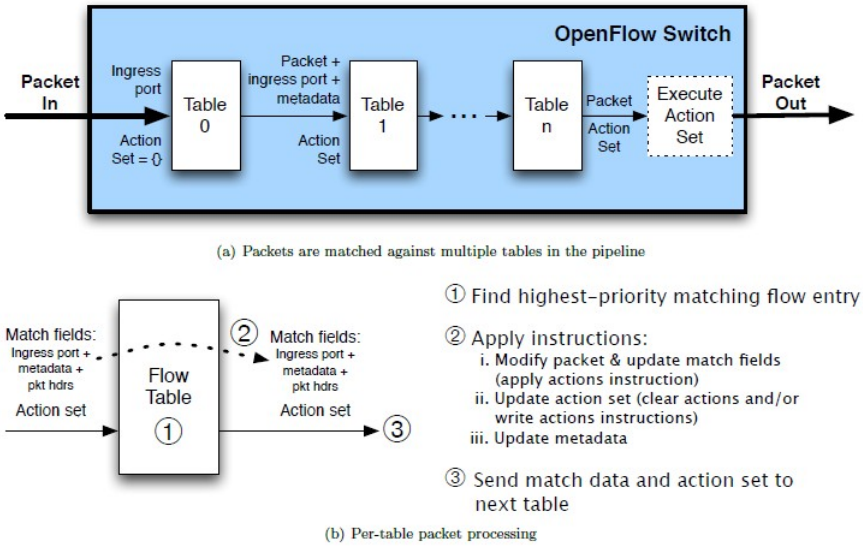

Figure 2.4: Pipeline processing [3]

Table 2.5: Main match fields

Fields	**When applicable**
Ingress port	All packets
Metadata	All packets
Ethernet source address	All packets on enabled ports
Ethernet destination address	All packets on enabled ports
Ethernet type	All packets on enabled ports
VLAN id	packets with VLAN tags
VlAN priority	packets with VLAN tags
MPLS label	packets with MPLS tags
MPLS traffic class	packets with MPLS tags
IPv4 source address	IPv4 and ARP packets
IPv4 destination address	IPv4 and ARP packets
IPv4 protocol/ARP opcode	IPv4, IPv4 over Ethernet, ARP packets
IPv4 ToS bits	IPv4 packets
Transport source port/ICMP Type	TCP, UDP,SCTP, and ICMP packets
Transport destination port/ICMP Code	TCP, UDP,SCTP, and ICMP packets

As for match fields, it is used to lookup the flow table depending on the packet type. Each entry in flow table contains a specific value. Table 2.5, Table 2.6 and Table 2.7 list the contents of the required match fields and details on the properties of each field of the OpenFlow specification v1.1 [3]. It can be seen from those tables that each header field has fixed size and is placed in the specific position of the match field. Flow table design procedure is explained further in Chapter 3.

Table 2.6: Main match fields description

Fields	**When applicable**
Ingress port	a physical or switch-defined virtual port
Metadata	
Ethernet source address	Can use arbitrary bitmask
Ethernet destination address	Can use arbitrary bitmask
Ethernet type	after VLAN tags
VLAN id	VLAN identifier
VlAN priority	VLAN PCP field
MPLS label	MPLS tags
MPLS traffic class	MPLS tags
IPv4 source address	subnet mask or arbitrary bitmask
IPv4 destination address	subnet mask or arbitrary bitmask
IPv4 protocol/ARP opcode	ARP opcode (lower 8 bits)
IPv4 ToS bits	upper 6 bits
Transport source port/ICMP Type	ICMP Type(lower 8 bits)
Transport destination port/ICMP Code	ICMP Type(lower 8 bits)

As for instructions, it mainly consists of action set which is associated with each packet. Supported instructions include **Apply-Actions**, **Clear-Actions**, **Write-Actions**, **Write-Metadata** and **Goto-Table** [3]. The action list (see Table 2.8) is included in the *Apply-Actions* as well as in the *Packet-out* message. The matched packets are forwarded and also modified according to the action list. After matching, the header field shown in Table 2.9 is required to be updated in the packets. However, only *Output* action is implemented in our OpenFlow switch. More details of forwarding action are introduced in Chapter 3.

Table 2.7: Main match fields lengths

Fields	Bits
Ingress port	32
Metadata	64
Ethernet source address	48
Ethernet destination address	48
Ethernet type	16
VLAN id	12
VlAN priority	3
MPLS label	20
MPLS traffic class	3
IPv4 source address	32
IPv4 destination address	32
IPv4 protocol/ARP opcode	8
IPv4 ToS bits	6
Transport source port/ICMP Type	16
Transport destination port/ICMP Code	16

Table 2.8: OpenFlow actions description

Actions	Description
Output (Required)	Forwards to a specific port
Set-Queue (Optional)	Sets queue id
Drop (Required)	Drop packets with no output actions
Group (Required)	Process the packet through the specified group
Push-Tag/Pop-Tag (Optional)	Push and pop VLAN, MPLS, PBB tags
Set-Field (optional)	Modify the values of the packet header field

Table 2.9: Set-field action

Set-field Actions
Set Ethernet source MAC address
Set Ethernet destination MAC address
Set VLAN ID
Set VLAN priority
Set MPLS label
Set MPLS traffic class
Set MPLS TTL
Decrement MPLS TTL
Set IPv4 source address
Set IPv4 destination address
Set IPv4 ToS bits
Set IPv4 ECN bits
Set IPv4 TTL
Decrement IPv4 TTL
Set transport source port
Set transport destination port
Copy TTL outwards
Copy TTL inwards

2.4.3 OpenFlow Channel

OpenFlow switch connects to the controller through the OpenFlow channel. Through this interface, the controller can manage and modify the flow table. The OpenFlow channel may be run over TCP, and is usually encrypted [3]. Moreover, all OpenFlow channel messages between OpenFlow switch and controller must be formatted according to the OpenFlow protocol [3].

2.5 FPGA-based platform

This section introduces a hardware architecture in NTNU. The board (XC6VLX240T device) used in our OpenFlow switch is Virtex-6 from Xilinx, which is also used in the implementation of distributed multimedia plays (DMP) network nodes, and it indicates lower latency and scalability features on FPGA based platform [2]. Figure

2.5 is the picture of ML605 and Figure 2.6 illustrates the block diagram of ML605. The Virtex-6 FPGAs are the programmable silicon foundation for Targeted Design Platforms that deliver integrated software and hardware components to enable designers to focus on innovation as soon as their development cycle begins, which provides the newest, most advanced features [13]. The main features of ML605 are shown in Table 2.10. It can be seen from Figure 2.6 that it has high speed interface (SFP), 200 MHz clock, compatible with 10/100/1000 Ethernet PHY (MII/GMII/RMII) and supports PCIe ×8 edge connector [6]. In addition to the high-performance logic fabric, Virtex-6 FPGAs contain many built-in system-level blocks. These features allow designers to build the highest levels of performance and functionality into FPGA-based systems. More features of the FPGA-platform are described in [13].

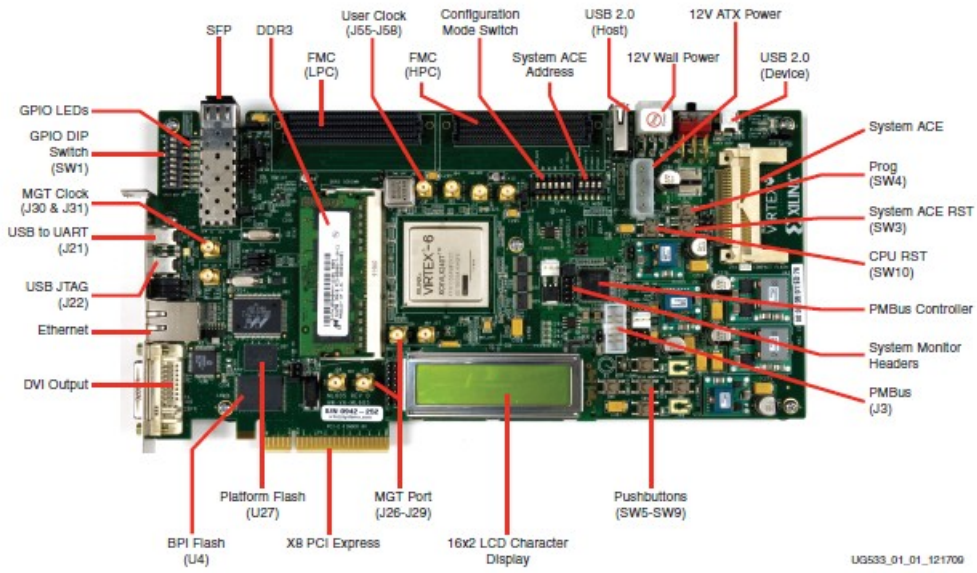

Figure 2.5: FPGA-based platform [5]

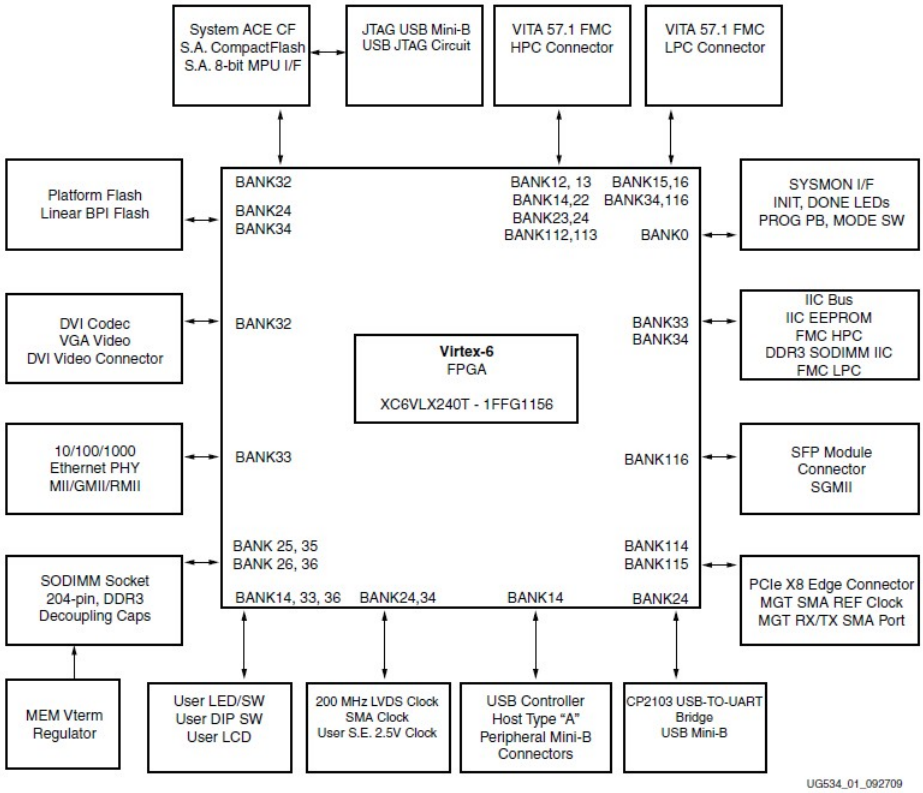

Figure 2.6: Xilinx Virtex 6 block digram [6]

Table 2.10: XC6VLX240T main features

Logic cells	Configurable Logic Blocks		BRAM	PCIe	Ethernet port	I/O
	Slices	DRAM (Kb)	Max(Kb)			
241,152	37,680	3,650	14,976	2	1	720

In the current work, PC and FPGA are the two hardware sections. FPGA board is plugged into PC through PCI slots (see Figure 2.7). Our hardware platform only has one Ethernet port, which limits the OpenFlow switch implementation. However, the simulation model of OpenFlow switch is implemented on this FPGA-based paltform, which is to test the performance of OpenFlow switch. In our OpenFlow switch implementation, the design environment is as follows:

- ML605 board with XC6VLX240T FPGA

- Intel Core i7 CPU 2.8 GHz 930 (4 core, 8 thread processor)
- Linux Ubuntu 10.04 LTS (2.6.32-41-generic)
- Motherboard: Gigabyte, X58A–UD7
- Xilinx ISE 14.7

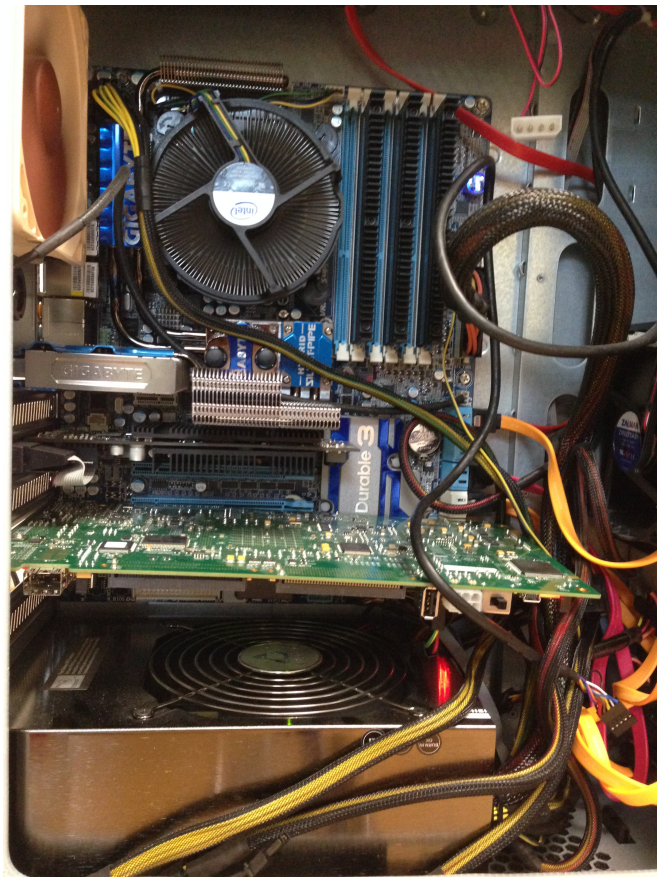

Figure 2.7: FPGA plugged in PC

2.6 Related Work

Many OpenFlow switches have already been implemented in different platforms such as Linux (software), OpenWRT (software) and NetFPGA (hardware). Hardware/commercial switches (e.g, HP ProCurve, NEC IP8800) and software/Test switches (NetFPGA switch, OpenWRT) have been released and used in real network

environments. Switches are open sources and can be found on the website so that everyone can download for using or modifying. This section briefly introduces some related work about current OpenFlow switch implementations.

[1] describes the Type-0 OpenFlow switch implementation on the NetFPGA platform. NetFPGA used in [1] is a Gigabit rate networking hardware, consisting of a PCI card with an FPGA, memory, and four 1-Gig Ethernet ports. This implementation could hold more than 32,000 exact-match flow entries running across four ports and the exact-match flow table can be expanded to more than 65000 entries. The implementation enhances flexibility and reusability of hardware. The performance results claim that it takes 11 μs to insert a new flow into switch due to the PCI bus bottleneck [1]. At the same time, the bottleneck of their OpenFlow switch is that it has 240 bits flow header entry currently along with the actions, which can be aggravated when packet re-injected from controller into switch using the same communication channel (PCI bus). Additionally, OpenFlow switch can provide many functionalities at lower logic gate cost in comparison with IPv4 Router, the NIC and the learning Ethernet switch [1].

While another OpenFlow switch implemented on NetFPGA can hold more than 100000 flow entries and it is also capable of running at line-rate across the four NetFPGA ports [14]. In the software plane, OpenFlow reference software implementation is extended by using a new δFA data structure to create the rules instead of hash function, which is more advanced than the switch implemented on the NetFPGA. This switch provides a flexible packet forwarding architecture based on regular expression [14]. Besides, it also enables the standard-compliant OpenFlow switching, which can be easily reconfigured through its control plane to support other kinds of applications [14]. Furthermore, the performance analysis is also done in this article. The results of performance analysis indicate that the switch is able to process all traffic data even in the case of a Gigabit link saturated with minimum-sized packets [14].

Because low-level Verilog RTL severely limits the portability of OpenFlow switch, the switch in [15] is implemented with Bluespec System Verilog (BSV) which is a high-level HDL, and addresses the challenges of its flexibility and portability. The design comprises of approximately 2400 lines of BSV code. This switch meets the OpenFlow 1.0 specification and achieves a line rate of 10 Gbps, which is highly modular and parameterized, and makes use of latency-insensitivity, split-transaction interfaces and isolated platform-specific features [15]. In this article, the OpenFlow Switch is also ported into NetFPGA-10G, the ML605 (Xilinx) and DE4 (Altera). The exact match flow tables of this switch is implemented on both Block RAM and DRAM. It is found that it has lower pipeline latency of 19 cycles for a packet to go from ingress to egress when implementing exact flow tables on Block RAM

[15]. Furthermore, the switch is implemented in two configurations, one is in an FPGA communication with controller via PCIe or the serial link, another is in an FPGA-based MIPS64 softcore. It is found that the switch responds to controller requests in less cycles used with the PCIe than serial link [15].

The related works about OpenFlow switch implementation have already mentioned above and most of OpenFlow switches are implemented on the NetFPGA. Except these related works, there is limited number of studies on performance analysis of the OpenFlow switch.

In order to improve the OpenFlow switching performance, the mechanisms are introduced in [16] and [17]. An architectural design is proposed to improve the lookup performance of OpenFlow switching in Linux by using a standard commodity network interface card. The results in [16] show a packet switching throughput increase of up to 25 percent compared to the throughput of regular software-based OpenFlow switching [16]. Instead, the results in [17] show a 20 percent reduction using network processor based acceleration cards to perform OpenFlow switching [17]. However, only one paper studies the performance measures of OpenFlow [18]. It is concluded that the OpenFlow implementation in Linux systems can offer very good performance and it shows good fairness capability in dealing with multiple flows [18]. Furthermore, it is also found from the results that large forwarding tables are generated due to L2 switching [18]. [19] also studies the performance measurements of not only OpenFlow switch but also OpenFlow controller. In [19], a performance model of an OpenFlow system is provided, which is based on the results from queuing theory and is verified by simulations and measurement experiments with a real OpenFlow switch and controller. The results in this article show that the sojourn time mainly depends on the processing speed of the OpenFlow controller [19]. Moreover, it indicates that lower is the coefficient of variation when the probability of new flows arriving at the switch is higher, but longer is the sojourn time [19].

Thus, it can be seen from the description above that OpenFlow-SDN has already appealed to some attentions in both researchers and vendors. At the same time, the increasing number of researchers gradually has implemented their own OpenFlow switch on FPGA-based platform. The OpenFlow network implementation described in this thesis is a little different from the related work. Our work is to do the simulation test of OpenFlow performance so that data plane and control plane are both implemented on FPGA (Virtex6). OpenFlow switch design framework is explained in details in the following chapter.

Chapter 3
OpenFlow Switch Design Framework

As it is mentioned in the previous chapter, the OpenFlow network architecture includes the OpenFlow switch, the OpenFlow controller and a secure channel based on the OpenFlow protocol which connects the OpenFlow switch to the OpenFlow controller. In this chapter, the main modules of OpenFlow switch designed on FPGA are described in detail, which are flow table entry composer, flow table controller, action processor and controller policy.

3.1 OpenFlow Switch Framework

3.1.1 Brief description

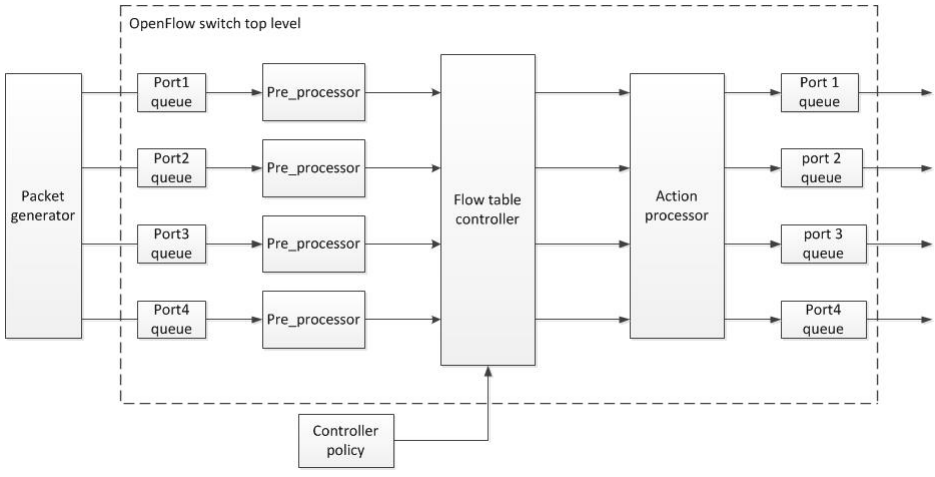

Figure 3.1: OpenFlow System Architecture

In our OpenFlow switch design, OpenFlow datapath receives the packets via packet generator. All peripherals share the same clocks (100MHz) and a reset. However, only the composed flow entry goes to the flow table controller module. 64-bit pipelines are beneficial for executing many tasks per clock and also for successful FPGA implementation. Since there is the only one Ethernet port, four datapath pipelines are designed to simulate more ports with using input queue and output queue as a switching facility in the top level module. Incoming packets from each physical input port go through dedicated pipeline. Figure 3.1 illustrates the OpenFlow system architecture and the brief framework of the OpenFlow switch design. The packets are generated, and have to stay in the output queue after being processed due to only one Ethernet port. The main three parts of OpenFlow architecture are the input queue module, the output port lookup module and the output queue module. The input queue and the output queue both consist of generic modules generated by two IP cores (FIFO generator [20] and Block RAM [21]) supported by Xilinx design tools (ISE 14.7 [22]). Each input queue connects to each port and buffers the received packets. And the sizes of both the FIFO queue block and buffer block are 64 (width)× 1024 (depth). The output port lookup module, clearly shown in Figure 3.2, is the most important part in the OpenFlow switch design framework, mainly consisting of flow table entry composer, flow table controller and action processor.

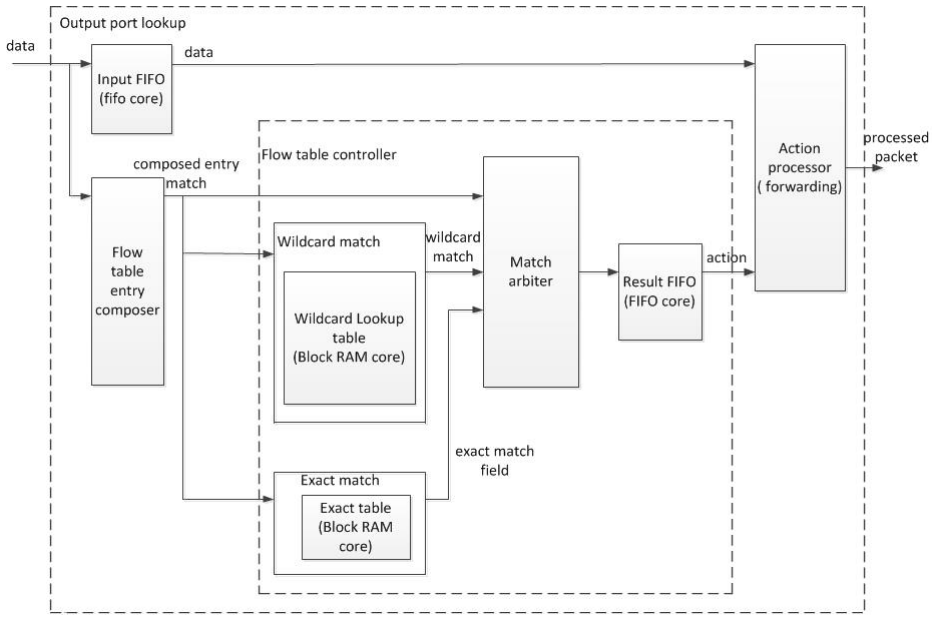

Figure 3.2: Output Port Lookup

When new packets generated from the packet generator stream into the OpenFlow

switch, important header information is extracted and then composed into fixed format which is compared with the flow table entries in two flow tables (exact match table and wildcard table). At the same time, incoming packets are buffered in the input FIFO buffer block, waiting for being forwarded. Then the matching results associated with forwarding action are sent to the action processor in order to tell the action processor how to deal with the packet (forwarding to the corresponding output queue). If the packet matches, it is forwarded to the corresponding output port according to the forwarding information in the action list. While if it doesn't match, the OpenFlow switch requests to the controller policy model to make decision of this unmatched packet. The policy of the controller policy module is to add flow entry information including the flow entry, the flow mask and the action. Here, both matched packets and unmatched packets are forwarded to the output queues finally. Output port lookup module and the policy module are depicted more in the following section.

3.2 Flow Table Entry Composer

After going through the FIFO queue, the packet initially goes to the flow table entry composer module. In this section, the implementation of flow table entry composer is described.

The purpose of the flow table entry composer is to extract packet headers and organizes them as a fixed form of the flow table entry. Figure 3.3 shows the process of the flow table entry composer module. It can be seen from Figure 3.3 that it is made up of the input FIFO queue block, header parser block, lookup entry composer block. Here, input FIFO queue block is also generated by IP cores (FIFO generator [20]). When a new flow comes in, the header fields are extracted. After being parsed, these extracted header information are composed into the flow entry with the fixed format in the lookup entry composer block. Then the composed flow entry is sent to flow table table modules for matching.

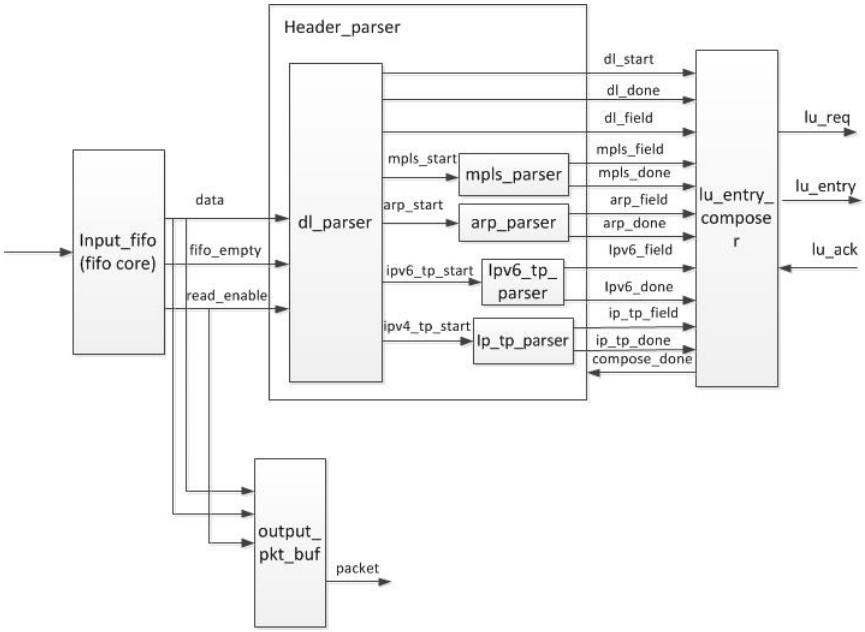

Figure 3.3: Flow Table Entry Composer

3.2.1 Queue block

Input FIFO queue block is a common block for OpenFlow switch architecture to reduce back pressure situation, also used in other modules. The FIFO block and output buffer block generated by FIFO generator IP cores [20] both buffer the incoming packets. The input FIFO block buffers incoming packets for parsing header. While the output buffer block buffers the incoming packets for action processor and also synchronous with parsed header. The buffer size (64×1024) is sufficient to store data until finishing header parsing.

3.2.2 Header parser block

Table 3.1: Match fields

Field	Bits
Match fields	256
Ingress port	8
Ethernet source address	48
Ethernet destination address	48
Ethernet type	16
VLAN id	12
VlAN priority	3
MPLS label	20
MPLS traffic class	3
IPv4 source address	32
IPv4 destination address	32
IPv4 protocol/ARP opcode	8
IPv4 ToS bits	6
IPv6 source address	128
IPv6 destination address	128
Transport source port/ICMP Type	16
Transport destination port/ICMP Code	16

Header parser module extracts L2 header information (dl_parser block) and also L3/L4 header information (ip_tp_parser block, ipv6_tp_parser block, arp_parser, mpls-parser block). Each header field has the exact position in the packet. Thus, the important header fields can be extracted according to their exact positions in Ethernet frame. Table 3.1 shows the header fields that are extracted from the packet in our design according to the match fields described in OpenFlow specification v1.1 [3].

According to the Table 3.1, Ethernet source/destination address, VLAN ID, priority (if VLAN tag) and Ethernet type need to be extracted from L2 header. Figure 3.4 illustrates the structure of Ethernet frame with and without VLAN tag (0x8100) or QinQ tag (0x8a88). Figure 3.5 illustrates the process of getting L2 header fields. When the Ethernet packets (64 bits per clock) come in, Ethernet source/destination addresses are extracted firstly. At the same time, header parsing

signal is sent to lookup entry composer which waits for receiving the extracted fields. If VLAN tag is found in the packet, VLAN ID and VLAN priority are obtained from the packet. Different Ethernet types (see Table 3.2) are detected through if statements. If one of those types is found, the corresponding header fields are extracted further. Otherwise, header parser block stops to parse further.

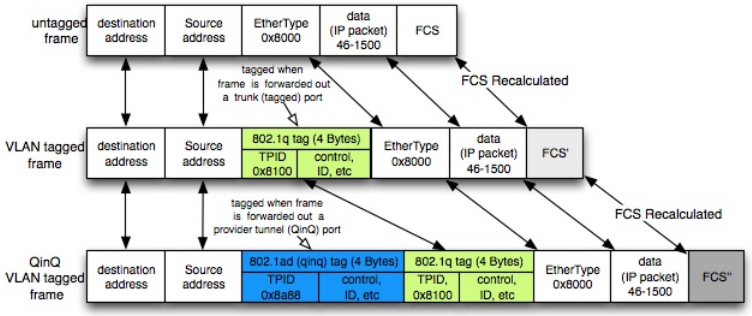

Figure 3.4: Ethernet Packet

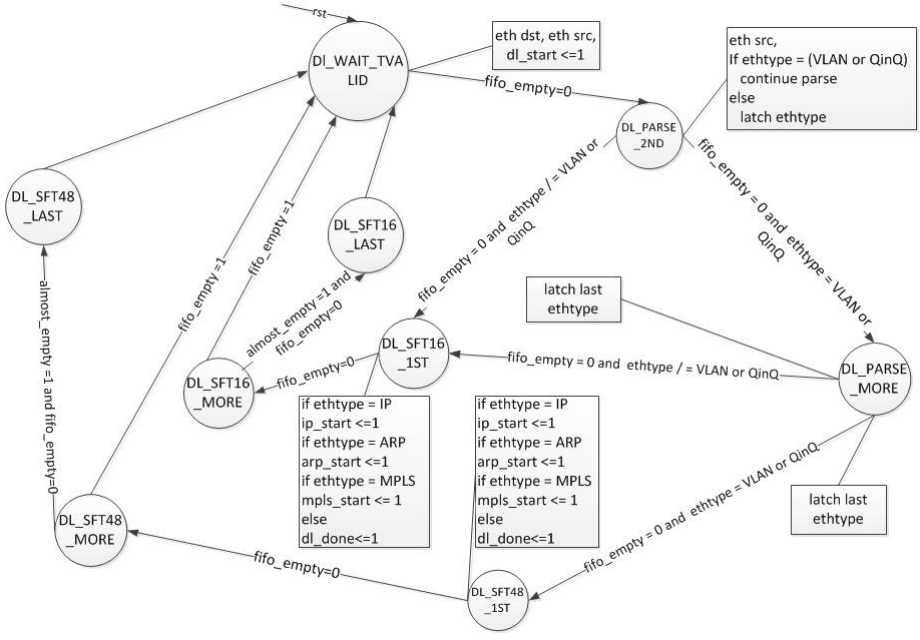

Figure 3.5: L2 parser state machine

Table 3.2: Ethernet type

Ethernet type	IPv4	ARP	MPLS unicast	MPLS multicast
Content	0x0800	0x0806	0x8847	0x8848

If Ethernet type is IPv4, ip_tp_parser starts to work. The structure of IP, ICMP, TCP/UDP and SCTP headers are described in Figure 3.6, Figure 3.7, Figure 3.8 and Figure 3.9 respectively. In L3, IP source/destination address, IPv4 protocol and IPv4 TOS need to be extracted from the IP header. Moreover, source/destination ports (TCP/UDP/SCTP) or ICMP type and ICMP code need to be extracted from L4 header. Figure 3.11 illustrates the procedure of ip_tp_parser. Besides, IP protocol type is also detected through if statements. If IP protocol is TCP, UDP, SCTP or ICMP (see Table 3.3), the packet is parsed further in order to extract the corresponding header fields. Otherwise, the match fields of L4 are put null.

Table 3.3: IP protocol type

IP protocol type	TCP	UDP	SCTP	ICMP
Content	0x06	0x11	0x84	0x01

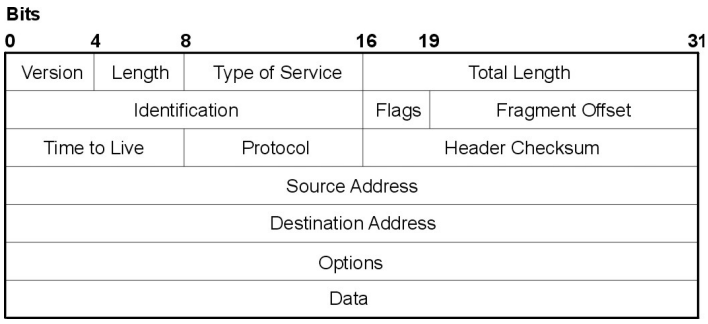

Figure 3.6: IP header

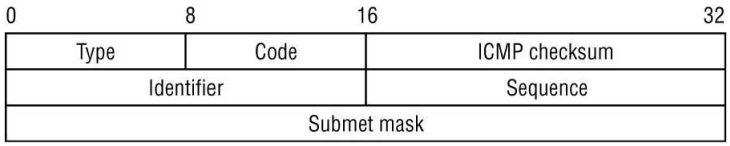

Figure 3.7: ICMP header

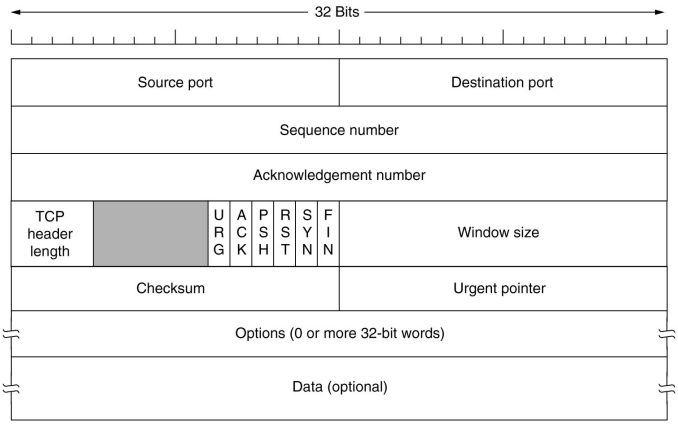

Figure 3.8: TCP header

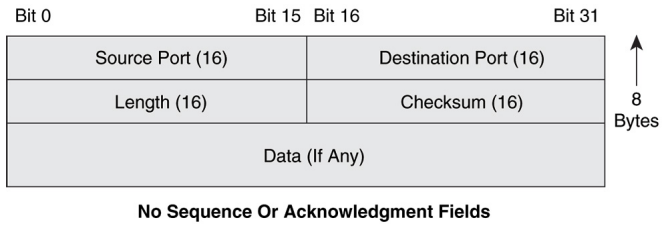

Figure 3.9: UDP header

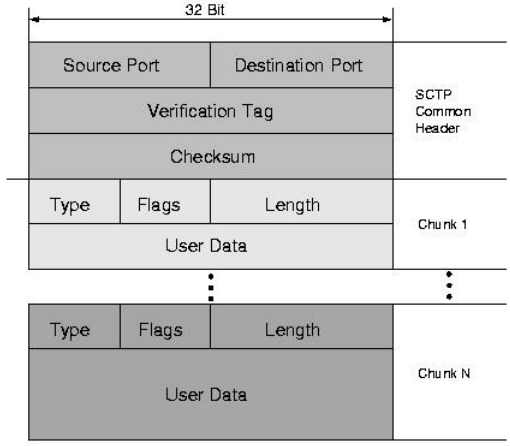

Figure 3.10: SCTP header

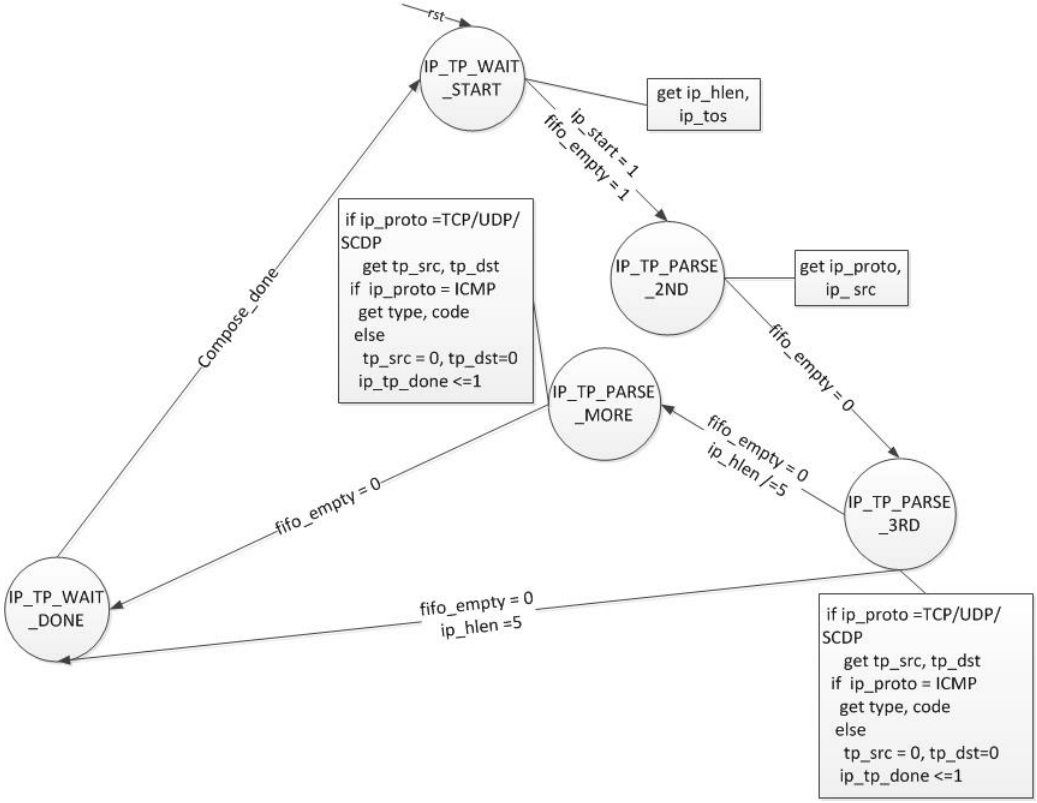

Figure 3.11: L3/L4 (IPv4) parser state machine

If Ethernet type is ARP, arp_parser (Figure 3.13) starts to work. The ARP opcode, sender IP address and target IP address in the ARP header fields (Figure 3.12) are extracted.

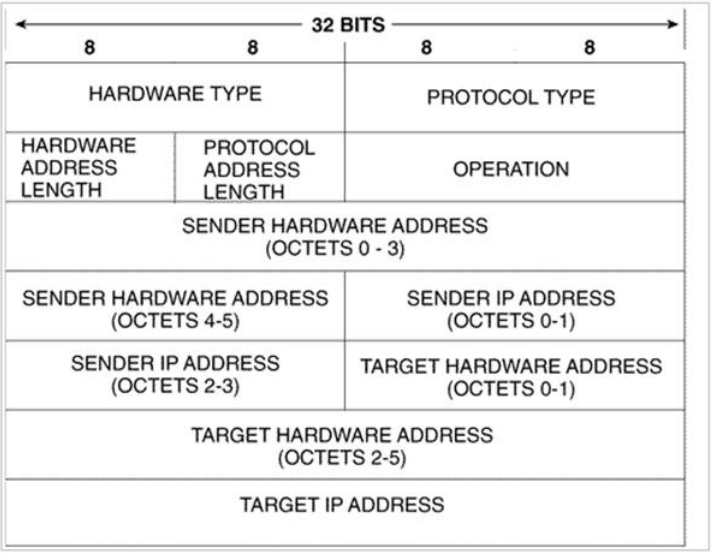

Figure 3.12: ARP header

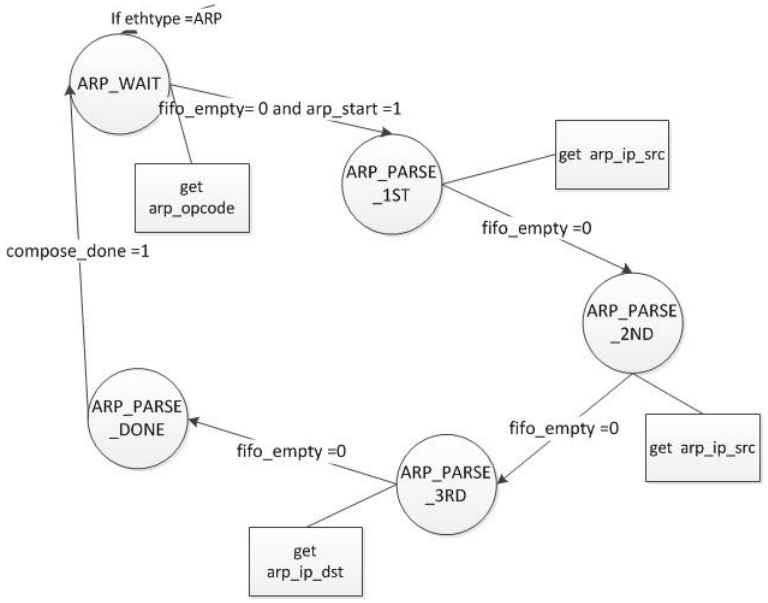

Figure 3.13: ARP parser state machine

It can be seen from Figure 3.14, MPLS label length is 20 bits and MPLS traffic

class is 3 bits in MPLS header fields. If Ethernet type is MPLS, mpls_parser state machine (Figure 3.15) starts to extract MPLS label and MPLS traffic class.

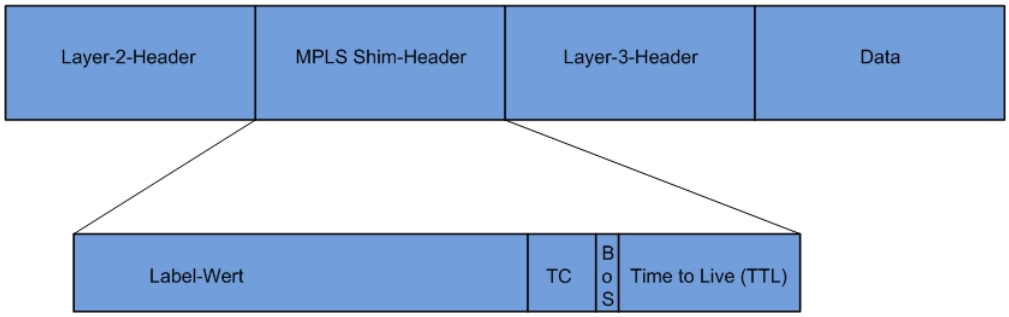

Figure 3.14: MPLS header

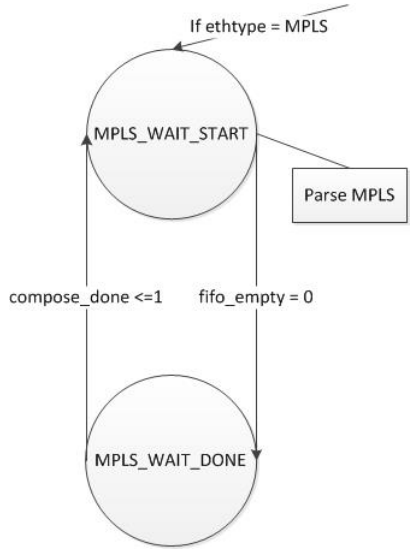

Figure 3.15: MPLS parser state machine

3.2.3 Lookup entry composer

The lookup entry composer block is ready to compose when the header parser block begins to work (dl_start <= '1'). The lookup entry composer block organizes all the parsed fields received from the header parser block into a specific format. All extracted fields have their own specific position in the lookup entry (lu_entry). This

block consists of three state machines shown in Figure 3.16: parsing-status check, request-latch and flow-table module interface. Parsing-status checks state machine is to communicate with the preceding header parser block. Request-latch state machine is used to compose these extracted header fields into lookup entry format. Finally, flow table controller interface state machine is to transfer signals to the following flow table controller module.

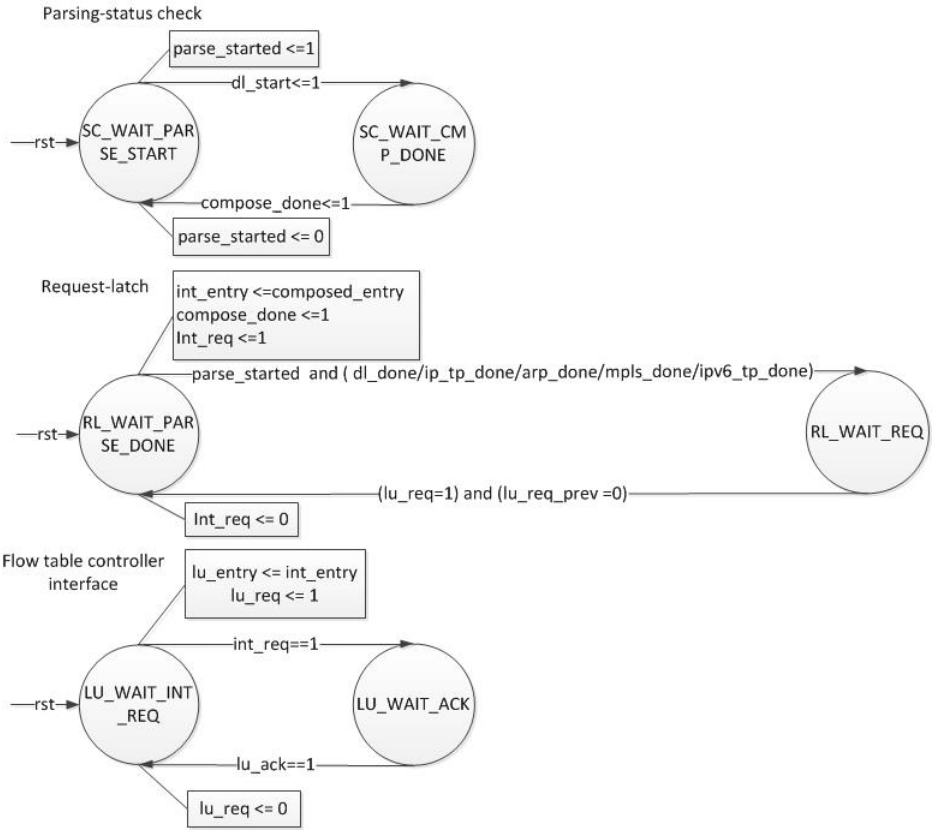

Figure 3.16: Lookup entry composer

The content of the flow entry is different due to the different Ethernet types and IP protocol types. Each extracted field is put into the exact position in the flow entry. The structure of the flow entry these header fields defined in our design is shown in the following algorithms. It can be seen from these algorithms (see Algorithm 3.1, Algorithm 3.2, Algorithm 3.3 and Algorithm 3.4) that the extracted fields are put in the exact positions of the flow entry.

Algorithm 3.1 Ethernet fields (no L3/L4 fields), program in VHDL.

```
if (dl_done='1' and ip_tp_done='0' and arp_don ='0' and mpls_done='0')then
        int_entry <=  src_port        --input port
                    & dl_src          --Ethernet source address
                    & dl_dst          --Ethernet destination address
                    & dl_ethtype      --Ethernet type
                    & dl_vlantag      --Ethernet VLAN
                    & X"00000000"     --IPv4 source address
                    & X"00000000"     --IPv4 destination address
                    & X"00"           --IPv4 protocol type
                    & X"00"           --IPv4 TOS
                    & X"0000"         --transport layer source port
                    & X"0000"         --transport layer destination port
                    & X"00";
        compose_done <= '1';
        int_req_nxt := '1';
        req_latch_state_nxt <= RL_WAIT_REQ;
```

Algorithm 3.2 ARP Ethernet type, program in VHDL.

```
      elsif (dl_done = '1' and arp_done = '1') then
        int_entry <=  src_port        --input port
                    & dl_src          --Ethernet source address
                    & dl_dst          --Ethernet destination address
                    & dl_ethtype      --Ethernet type
                    & dl_vlantag      --Ethernet VLAN
                    & arp_ip_src      --ARP source address
                    & arp_ip_dst      --ARP destination address
                    & arp_opcode      --ARP operation code
                    & X"00"           --IPv4 TOS
                    & X"0000"         --transport layer source port
                    & X"0000"         --transport layer destination port
                    & X"0000000000";
          compose_done <= '1';
          int_req_nxt := '1';
          req_latch_state_nxt <= RL_WAIT_REQ;
```

Algorithm 3.3 MPLS Ethernet type, program in VHDL.

```
elsif (dl_done = '1' and mpls_done = '1') then
 int_entry <=  src_port       --input port
               & dl_src       --Ethernet source address
               & dl_dst       --Ethernet destination address
               & dl_ethtype   --Ethernet type
               & dl_vlantag   --Ethernet VLAN
               & mpls_lable   --MPLS label
               & mpls_tc      --MPLS traffic class
               & X"00000000"
               & X"00000000"
               & X"0000"
               & X"0000"
               & B"0";

  compose_done <= '1';
  int_req_nxt := '1';
  req_latch_state_nxt <= RL_WAIT_REQ;
```

Algorithm 3.4 IPv4 Ethernet type, program in VHDL.

```
elsif (dl_done = '1' and ip_tp_done = '1') then
 int_entry <=  src_port       --input port
               & dl_src       --Ethernet source address
               & dl_dst       --Ethernet destination address
               & dl_ethtype   --Ethernet type
               & dl_vlantag   --Ethernet VLAN
               & ip_src       --IPv4 source address
               & ip_dst       --IPv4 destination address
               & ip_tos       --IPv4 TOS
               & tp_src       --transport layer source port
               & tp_dst       --transport layer destination port
               & X"00";
  compose_done <= '1';
  int_req_nxt := '1';
  req_latch_state_nxt <= RL_WAIT_REQ;
```

3.2.4 Signals

The signals transferred between modules are listed and described here.
(1) Signals to the flow table controller module

lu_entry (3-0) (256 bits, output):

- Flow table entry (256 bits) organized by lookup composer block for matching against flow table
- Latched when a request (lu_req) to flow table controller is active and released when an acknowledgement (lu_ack) from flow table controller is received

lu_req (3-0) (output):

- Lookup request generated by lookup composer block to flow table controller
- Action processor also uses it to start reading packet out of output_pkt_buf
- Active until an acknowledgement (lu_ack) from flow table controller is received

(2) Signals from the flow table controller

lu_ack (3-0) (input):

- Generated by flow table controller module
- The flow table lookup request is accepted but not finished when asserted
- Releasing lu_req and lu_entry

(3) Signals to the action processor module

packet (64 bits, output):

- Sent when FIFO read_enable is asserted that means a matching field is found
- Sent 64 bits per clock

3.2.5 Simulation test

The testbench in VHDL is written to test that the functions of header parser block, lookup entry composer block and the whole flow table entry composer module. The testing packet (1024 bits) is written in the testbench files for the simulation and the packet is generated every 64 bits per clock. The simulation test result is shown in Figure 3.17. Algorithm 3.5 shows the example of the testbench. Figure 3.17 shows that the important header fields are extracted correctly and these fields are composed correctly into lu_entry (see Figure 3.18). Figure 3.19 shows the simulation results of the top module of these two main blocks, which also indicates that this module works correctly.

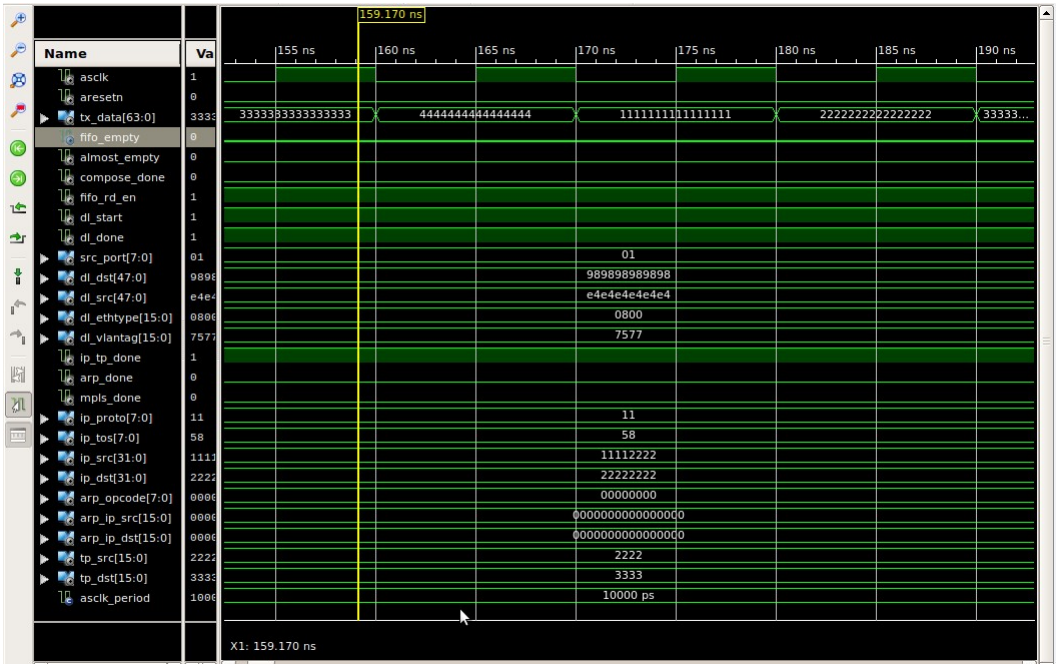

Figure 3.17: Header parser simulation test result

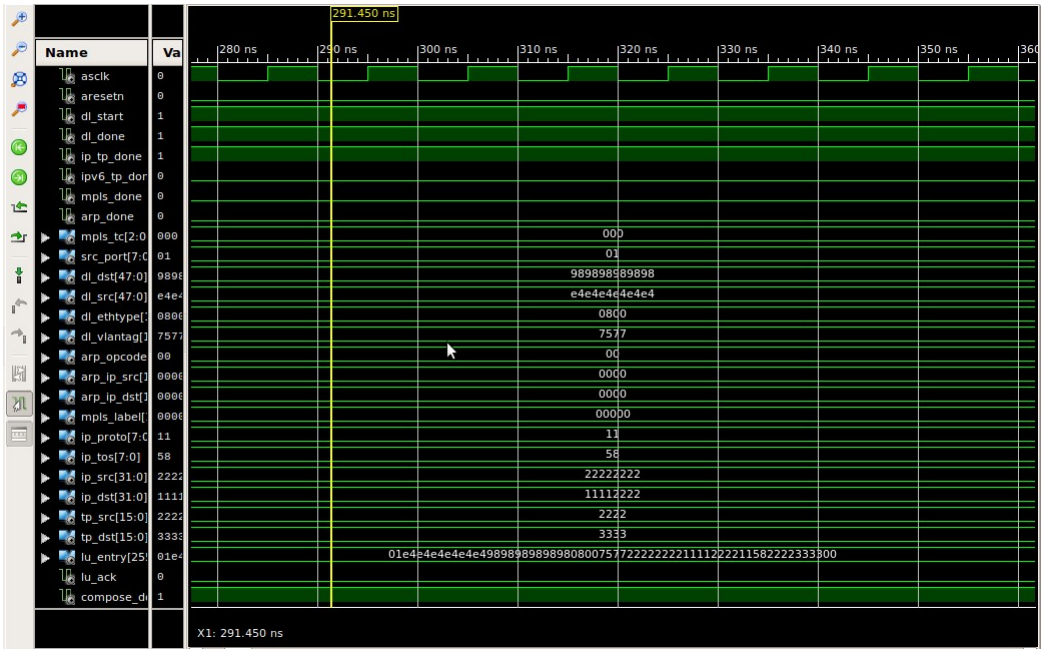

Figure 3.18: Lookup entry composer simulation test result

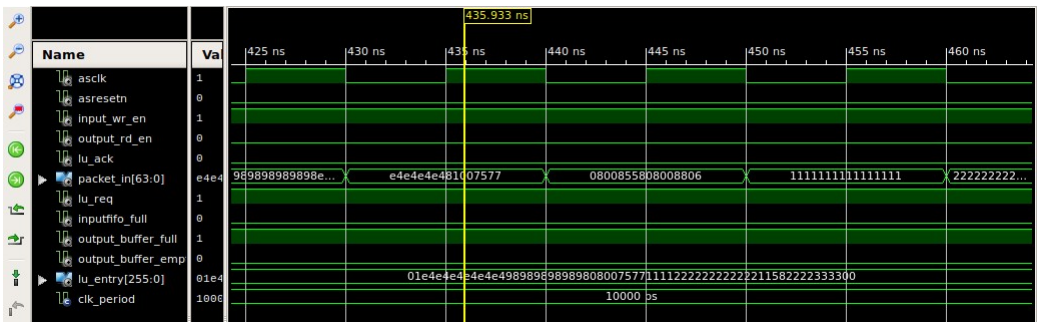

Figure 3.19: Flow entry composer simulation test result

Algorithm 3.5 Header parser testbench, program in VHDL.

```
process
  begin
    wait for 100 ns;
    fifo_empty <='0';
    tx_data  <= X"989898989898e4e4";
    wait for 10 ns;
    tx_data  <= X"e4e4e4e481007577";
    wait for 10 ns;
    tx_data  <= X"0800855808008806";
    wait for 10 ns;
    tx_data  <= X"1111111111111111";
    wait for 10 ns;
    tx_data  <= X"2222222222222222";
    wait for 10 ns;
    tx_data  <= X"3333333333333333";
    wait for 10 ns;
    tx_data  <= X"4444444444444444";
    wait for 10 ns;
    tx_data  <= X"1111111111111111";
    wait for 10 ns;
    tx_data  <= X"2222222222222222";
    wait for 10 ns;
    tx_data  <= X"3333333333333333";
    wait for 10 ns;
    tx_data  <= X"4444444444444444";
    wait for 10 ns;
    tx_data  <= X"5555555555555555";
    wait for 10 ns;
    tx_data  <= X"6666666666666666";
    wait for 10 ns;
    tx_data  <= X"7777777777777777";
    wait for 10 ns;
    tx_data <= X"2222222222222222";
    wait for 10 ns;
    almost_empty <= '1';
    tx_data <= X"3333333333333333";
    wait for 10 ns;
    fifo_empty <='1';
    almost_empty <= '0';
    wait for 10 ns;
  end process;
END;
```

3.3 Flow Table Controller

The lookup entry is looked up in flow table controller module after being parsed and being extracted. This section explains the entire procedure of looking up flow tables and writing flow entries.

3.3.1 Flow table controller module

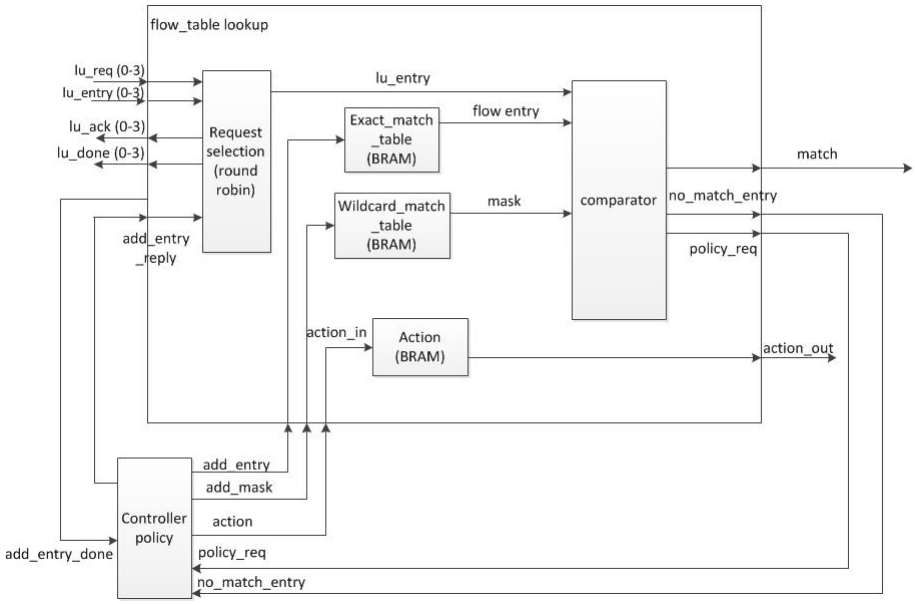

Figure 3.20: Flow table controller module

Figure 3.20 illustrates the main components of the flow table controller module including the request selection, the exact match table, the wildcard match table, the action, the comparator and also the controller policy. Flow table controller module manages the flow tables and handles all the requests (lookup requests from the flow table composer module and the writing request from the controller policy).

The process of looking up flow table and writing the flow entry is described in Figure 3.21. When a port queries if there is a matching field, it consults both exact match table and wildcard table. Round Robin scheduling is used in request selection block in order to schedule all the requests fairly. Exact match table and wildcard match table are flow entries storage and flow masks storage respectively. While action stores the action information. Table 3.4 shows the storage size of two flow tables and the action (16 bits). Moreover, the mask field is defined to include

Ethernet source address and Ethernet destination address in our implementation. Exact match table, wildcard match table and the action are implemented by BRAM [21]. Then the lookup entry (lu_entry) is sent to the comparator block when lookup request (lu_req = '1') is handled. In the comparator block, the received lookup entry is compared with the flow entries read from the exact match table and masks read from the wildcard match table through 'for-loop' statements. If a matching entry is found, the corresponding action is grabbed from the action storage and is also sent to the action processor module. Besides, if it matches both exact match table and wildcard match table, then the action for the exact match table is used. If there is no matching field, a policy request is sent to the controller policy module. The controller policy module is explained further in the following subsection.

Table 3.4: Flow tables and action lists size storage

BRAM	Exact match table	Wildcard match table	Action
width (bits) × depth	256 × 1024	256 × 1024	256 × 1024

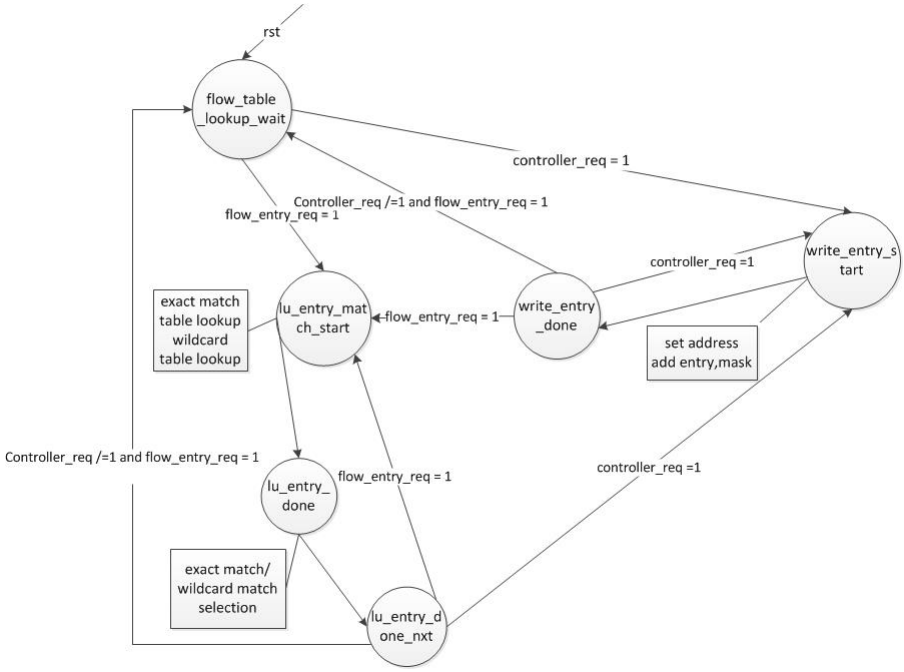

Figure 3.21: Flow table controller state machine

3.3.2 Signals

The signals transferred between modules are listed and described following:
(1) Signals of the flow entry composer module query and respond

lu_req (3-0) (input):

- This port has a lookup query when asserted one and it is set to zero when lu_ack is consumed by flow entry composer

lu_entry (3-0) (256 bits, input):

- It is parsed header information (256 bits) to be matched against two flow tables

lu_ack (3-0) (output):

- Sent from this module when lu_req is accepted and lu_entry is started to be looked up, but the process hasn't been done or action is ready
- It is asserted to one for one-clock period

lu_done (3-0) (output):

- Sent from this module when the lookup process is finished and action is ready
- It is asserted to one also for one-clock period

(2) Signals to the controller policy module

add_entry_reply (input):

- It is same as lu_req and an entry information is ready to be written when asserted to one
- Keep asserting 0 or 1 until add_entry_done is asserted

add_entry (256 bits, input):

- Flow entry to be written

add_mask (256 bits, input):

- Flow mask to be written

action (256 bits, input):

- Action list to be written

no_match_entry (256 bits, output):

- Lookup entry is sent to the controller policy when no matching field is found

policy_req (output):

- Asserted to one when no matching field is found

add_entry_done (output):

- It is same as lu_done and asserted to one when flow entry and flow mask are written successfully

(3) Signals to the action processor module

match (output):

- Notify signal is sent to the action processor to tell it that it is ready to forward the matching packet

action_out (256 bits, output):

- Sent to the action processor when matching field is found

3.3.3 Simulation test

Algorithm 3.6 Flow table lookup testbench example, program in VHDL.

```
stim_proc1: process
begin
wait for 100 ns;
lu_req1<= ‘1’;
lu_entry1<=X"012222222222675467548e31222200000000000000000000000000000000";
lu_req2<= ‘1’;
lu_entry2<=X"02e4e4e4e4e4e4989898989898080075771111222222222222211582222333300";
lu_req3<= ‘1’;
lu_entry3<=X"033333333333337676767676763333000000000000000000000000000000000";
lu_req4<= ‘1’;
lu_entry4<=X"041111111111155555555555511110000000000000000000000000000000000";
add_entry_reply <= ‘1’;
add_entry<=X"02e4e4e4e4e4e4989898989898080075771111222222222222211582222333300";
add_mask<=X"02e4e4e4e4e4e4989898989898000000000000000000000000000000000000000";
wait for 10 ns;
lu_req1 <= ‘0’;
lu_entry1<=X"000000000000000000000000000000000000000000000000000000000000000";
wait for 10 ns;
lu_req2 <= ‘0’;
lu_entry2<=X"000000000000000000000000000000000000000000000000000000000000000";
wait for 10 ns;
lu_req3 <= ‘0’;
lu_entry3<=X"000000000000000000000000000000000000000000000000000000000000000";
wait for 10 ns;
lu_req4 <= ‘0’;
lu_entry4<=X"000000000000000000000000000000000000000000000000000000000000000";
wait for 10 ns;
add_entry_reply <= ‘0’;
wait for 10 ns;
add_entry_reply <= ‘1’;
add_entry<=X"012222222222675467548e31222200000000000000000000000000000000";
add_mask<=X"012222222222675467548e31000000000000000000000000000000000000000";
lu_req2<= ‘1’;
lu_entry2<=X"02e4e4e4e4e4e4989898989898080075771111222222222222211582222333300";
end process;
END;
```

The Algorithm 3.6 illustrates the example of the flow table lookup testbench. In this testbench, four lookup requests with four entries and one write request with the flow entry information (flow entry, mask and action) are generated. This testbench is to test two results in two conditions. One, it is to test the lookup function when the new flow entry enters. If four new requests come in at the same time, one request is handled per clock using Round Robin schedule. It can be seen from Figure 3.22 that policy_req is asserted to one and no_match_entry is sent out orderly when no matching fields (match <= '0000').

Table 3.5: The 'match' value description

match	description
0000	no matching is found
0001	packet sent from the first port matches
0010	packet sent from the second port matches
0100	packet sent from the third port matches
1000	packet sent from the fourth port matches

The description of different 'match' values is shown in Table 3.5. Thus, the simulation results show that the flow table lookup function works correctly. And two, the add_entry_reply is asserted to one and the flow entry information of the second port is generated (add_entry and add_mask) in the testbench for testing the function of the writing flow entry. It can be seen from Figure 3.23 that the same lookup entry (lu_entry2 <= X"02e4e4e4e4e4e4989898989898080075771111222222222222115822223 33300") from the second port is sent again to this module after a few time. And the result of 'match' is '0010' which means it has the matching field. In other words, it indicates that the flow entry information has already been written into flow tables successfully when the previous writing request comes.

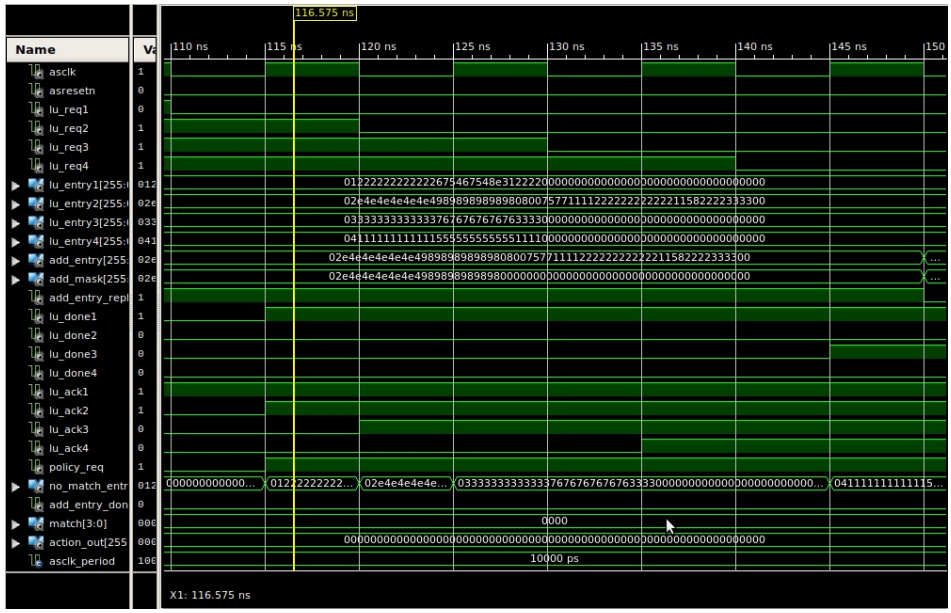

Figure 3.22: Flow table lookup simulation test results

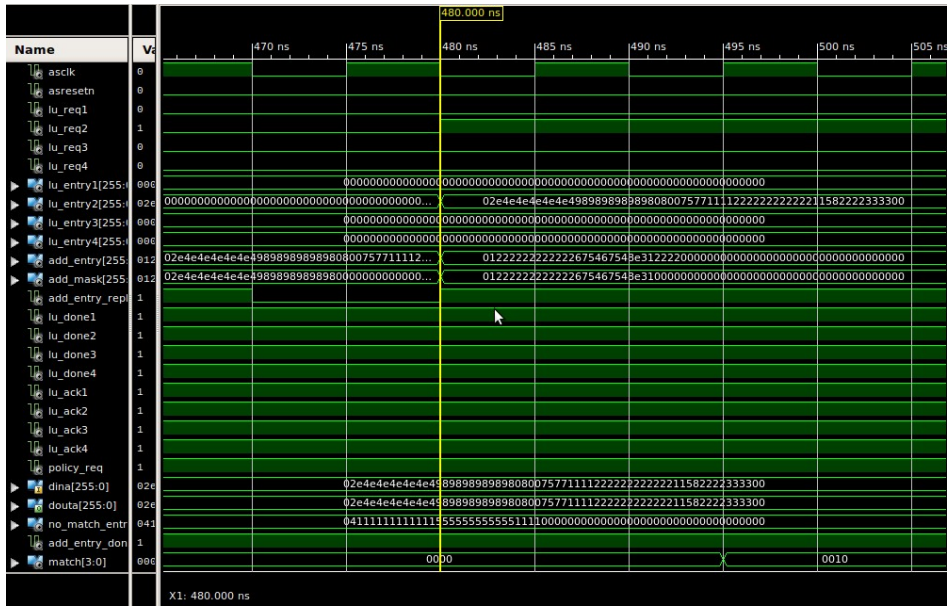

Figure 3.23: Writing flow entry simulation test results

3.4 Action Processor

3.4.1 Action processor module

The role of the action processor module (see Figure 3.24) is to specify forwarding ports and to update header fields and length of the packets according to the OpenFlow switch specification. Due to the limited time, only output forwarding action is executed in our action processor module. Packets are sent to the corresponding port queues referring to the action received from the flow table controller.

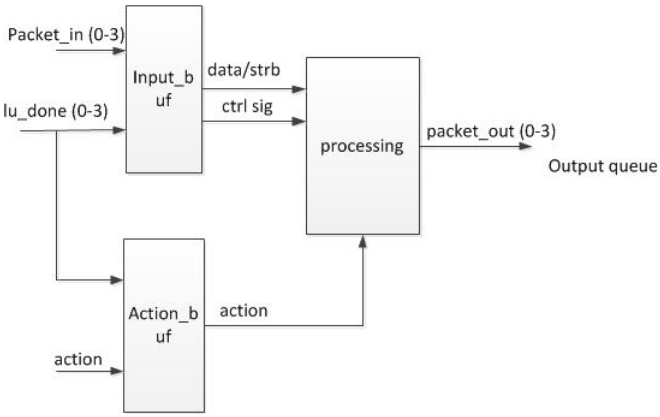

Figure 3.24: Action processor

The action (see Table 3.6) includes the information such as output port, action flag, VLAN ID, etc. Table 3.7 lists these OpenFlow actions. Action flag is to give the exact instructions to execute these actions. The length of action fag is 16 bits and each bit is assigned to an OpenFlow action, and if the value for a field is one, it means this action is expected to be performed.

Table 3.6: Action

Contents	bits
Forward bitmap	16
Action flag	16
VLAN ID	16
VLAN PCP	8
Ethernet source address	48
Ethernet destination address	48
IPv4 source address	32
IPv4 destination address	32
IPv4 TOS	8
Transport layer source port	16
Transport layer destination port	16

Table 3.7: Action flag

Bit	Action
0	Output
1	Set VLAN ID
2	Set VLAN PCP
3	Pop VLAN
4	Set Ethernet source address
5	Set Ethernet destination address
6	Set IPv4 source address
7	Set IPv4 destination address
8	Set IPv4 TOS
9	Set transport layer source port
10	Set transport layer destination port
11	Set IPv4 ECN
12	Push VLAN
13	Set IPv4 TTL
14	Decrement IPv4 TTL
15	Reserved

3.4.2 Signals

(1) Signals from the flow table composer

packet_in (64 bits, input):

- Sent from output packet buffer block of flow entry composer module only when action is valid.
- Sent per 64 bits per clock

(2) Signals from the flow table controller

match (input):

- Sent from flow table controller notify to forward packets

lu_done (input):

- Sent from flow table controller when lookup process is done and action is ready
- Action is gotten and stored until the next lu_done when it is asserted to one.
- Asserted to one for one-clock period

action (256 bits, input):

- Sent from flow table controller
- Valid only when lu_done and match are both asserted to one
- All zero means to drop this packet

(3) Signals to the output queue

packet_out (64 bits, output):

- Packet is sent to the corresponding port queue after action processing
- Sent per 64 bits per clock

3.4.3 Simulation test

Figure 3.25 shows the simulation results of the action processor module. Since only the forwarding action is implemented in this module, this simulation is to check whether the packets are forwarded to the corresponding output port queues. In our design, the forwarding strategy is that the packet from the current port is forwarded to the queue of next port. For example, the packets from the first port are forwarded to the second port, the packets from the second port are forwarded to the third port and the packets from the third port are forwarded to the fourth port. The packets from the fourth port are forwarded to the first port. It can be seen from Figure 3.25 that these packets received from the different ports are forwarded to the corresponding ports correctly.

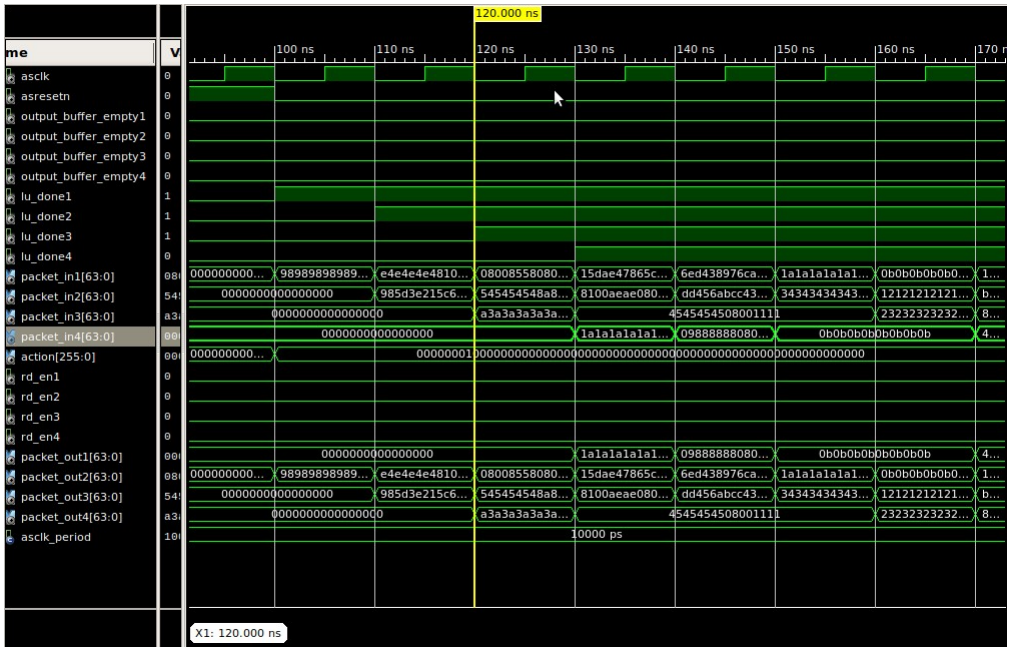

Figure 3.25: Action processor simulation test results

3.5 Controller Policy

It is supposed to install the controller in PC or different FPGA-platforms. However, the controller policy module is done in the same FPGA. If no matching field is found, controller policy module starts to work to make a decision about how to deal with the

unmatched packet. However, this module is just to imitate the controller function, but not implementing the complete functions of the controller. The policy defined in our implementation is to write new flow entry, new mask and new action when no matching happens.

3.5.1 Controller policy module

Figure 3.26 and Figure 3.27 illustrate the controller policy module and the process of writing the new flow entry respectively. It can be seen from the state diagram that the controller policy starts to work when the request signal sent from the flow table controller module is asserted to one. Then '0' bit of action flag in 'action' is set to one, which means to execute the forwarding action. At the same time, new flow entry, mask and forwarding port number are generated and sent to the flow table controller module. The state goes back to the waiting state for the next request when the writing process is done.

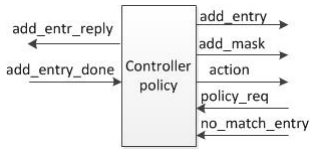

Figure 3.26: Controller policy module

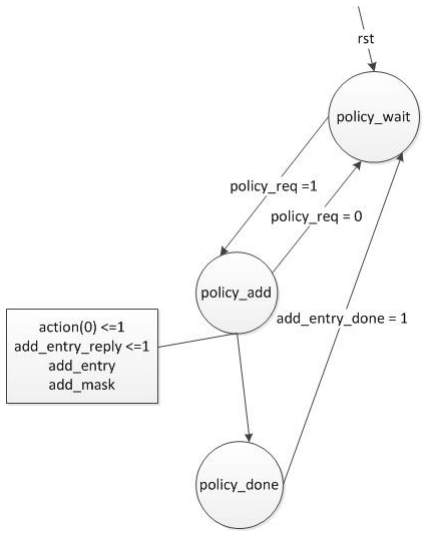

Figure 3.27: Policy state machine

3.5.2 Signals

(1) Signals from the flow table controller module

policy_req (input):

- Sent from the flow table controller module
- Asserted to one when no matching filed is found

no_match_entry (256 bits, input):

- It is unmatched flow match fields (256 bits) sent from flow table controller

add_entry_done (input):

- Sent from the flow table controller module
- Asserted to one when the process of writing flow entry is done

(2) Signals to the flow table controller module

add_entry_reply (input):

- Sent to the flow table controller module
- It means that it wants to write the information when asserted to one and it release a flow entry information (flow entry, mask and action)
- Asserted to one for one-clock period

add_entry (256 bits, input):

- It is the new flow entry and sent to the flow table controller module when add_entry_reply is asserted to one

add_mask (256 bits, input):

- It is the new flow mask and sent to the flow table controller module when add_entry_reply is asserted to one

action (256 bits, output):

- It is the new flow action and sent to the flow table controller module when add_entry_reply is asserted to one

3.5.3 Simulation test

This simulation is to test the function of the controller policy module. The function of this module is to generate the flow entry information (add_entry, add_mask, action) and write them into the flow tables after receiving no matching request (policy_req <= '1'). In addition, only forwarding bit (action(0) <= '1') in action flag is asserted to one. Figure 3.28 shows the simulation results of this module, which indicates that the writing request with the flow entry information are generated correctly.

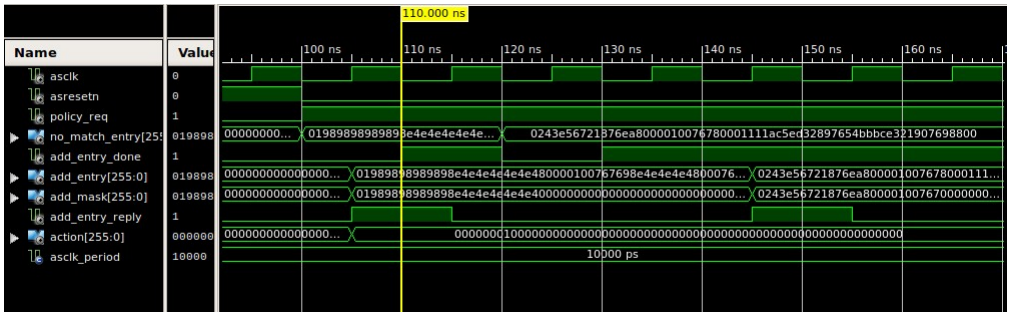

Figure 3.28: Controller policy simulation test result

After implementing the OpenFlow switch implementation, the performance simulation is done to measure the switch service time, sojourn time and controller service time. The following chapter shows the results of performance simulation.

Chapter 4
Performance Simulation

In this chapter, the results of performance simulation are described, specifically the service time (switch and controller) and sojourn time.

4.1 Resources utilization

Table 4.1: Design summary/reports

Device Utilization Summary (estimated values)			
Logic Utilization	Used	Available	Utilization
Number of Slice Registers	5330	301440	1%
Number of Slice LUTs	6870	150720	4%
Number of fully used LUTT-FF pairs	4604	7596	60%
Number of bonded IOBs	522	600	87%
Number of Block RAM/FIFO	31	416	7%
Number of BUFG/BUFGCTRLs	16	32	50%
Clock speed	100 MHz		
Power	275 mW		

Table 4.1 provides the device utilization used in our implementation such as utilized resources, the operational clock speed, the consumed power of the switch etc. The resource utilization is not very high according to Table 4.1. The utilization of slice registers, slice LUTs and Block RAM/FIFO is low. Because our OpenFlow switch design didn't implement the complete functions of OpenFlow siwtch and controller, it is not so complicated compared to current OpenFlow switches (Table 4.2). For example, only forwarding function of switch is implemented and other functions such as updating Ethernet source address, Ethernet destination address haven't

implemented. And only the function of adding flow entry is implemented. Thus, the resource utilization is less compared other two implementation. Besides, OpenFlow switches implemented on NetFPGA-10G and DE4 are designed in high-level hardware language (BSV), while our switch is designed in VHDL that is low-level language.

Table 4.2: Comparison of OpenFlow switch implementations on three FPGA boards

	NetFPGA-10G [15]	DE4 [15]	ML605
Ports	5×5	5×5	1×1
LUTs	24009	11131	6870
Flips Flops	29326	40287	4604
Block RAMs	159	1.1 Mb	31
Clock speed	160 MHz	100 MHz	100 MHz
Power	876 mW	442 mW	275 mW

4.2 Service time and Sojourn time

In order to do the performance simulation of the switch, the packet generator is implemented to generate the packets into OpenFlow switch module. In the performance simulation, performance metrics such as the controller service time, the switch service time and sojourn time are observed and measured. The packet generator is introduced briefly here. It is implemented in the OpenFlow switch testbench file. The packets are generated continuously and periodically (64 bits per 50 ns). Table 4.3 shows the switch service time and sojourn time. It can be concluded that the sojourn time in this case is end-to-end packet delay.

The switch service time (μ_{Switch}) is the time that packets spend in the switch. Because the queue modules are implemented inside the switch, the waiting time is included in the switch service time. The forwarding time for different packet sizes between 64 bytes and 1514 bytes is measured and the mean switch service time μ_{Switch} is estimated based on the results, shown in Table 4.3. In order to measure the time, the method introduced in [19] is used. The method is that the OpenFlow switch needs to forward the packets without the controller interaction [19]. Bursts of one hundred identical packets are generated in the packet generator module. A rule matching these packets is pre-written into the switch.

Table 4.3: Performance simulation results

Packet size (bytes)	Switch service time (μs)	Sojourn time (μs)
64	0.48	0.51
128	0.88	0.93
192	1.3	1.34
256	1.68	1.75
320	2.08	2.12
384	2.58	2.62
448	2.88	2.91
512	3.38	3.41
576	3.70	3.73
640	4.08	4.12
704	4.48	4.51
768	5.00	5.03
832	5.28	5.34
896	5.68	5.71
960	6.08	6.11
1024	6.50	6.54
1088	6.88	6.93
1152	7.28	7.31
1216	7.68	7.71
1280	8.08	8.11
1344	8.48	8.51
1408	8.90	8.93
1536	9.30	9.33

Figure 4.1 plots the simulation results of the switch service time. It can be seen that there is an almost linear increase of the mean switch service time from about 0.48 μs to about 9.3 μs with the increase in payload size.

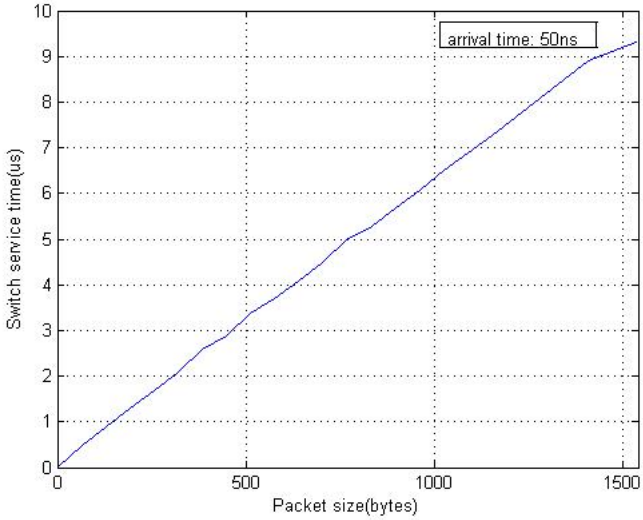

Figure 4.1: Switch service time

The sojourn time ($\mu_{Sojourn}$) consists of the switch service time (μ_{Switch}), the controller service time ($\mu_{Controller}$) and the communication time between the switch and the controller (μ_{S-C}). The sojourn time for different packet sizes between 64 bytes and 1514 bytes is measured and it is estimated the mean sojourn time ($\mu_{Sojourn}$) based on the results, shown in Table 4.3. As the similar method introduced in last section. In order to measure the time, the switch forwards the packets with the controller interaction this time. The mean sojourn time also has almost linear increase as shown in Figure 4.2. It shows the similar linear increasing trend from about 0.51 μs to about 9.33 μs with the increase in payload size. Besides, it is also found that the switch responses to the writing request from the controller with the fixed latency 2 cycles in our implementation. This time is shorter than the the comparison with the result (9 cycles) in [19]. In order to measure the controller service time, 10 new flows of each packet size are inserted to the switch. The arrival rate of these new flows are same with that used in the switch service time and the sojourn time measurements. The controller service time plotted in Figure 4.3 is calculated by the following formula:

$$\mu_{Switch} + \mu_{Controller} + \mu_{S-C} = \mu_{Sojourn}$$

μ_{Switch}: Switch service time
$\mu_{Controller}$: Controller service time

$\mu_{Sojourn}$: Sojourn time
μ_{S-C}: Communication time between switch and controller

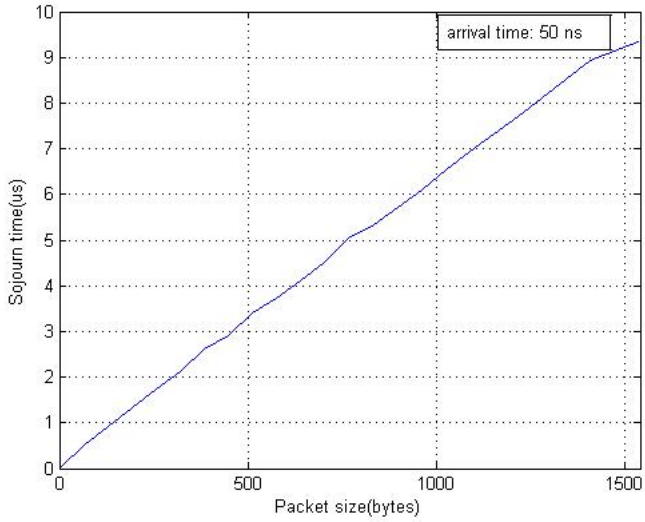

Figure 4.2: Sojourn time

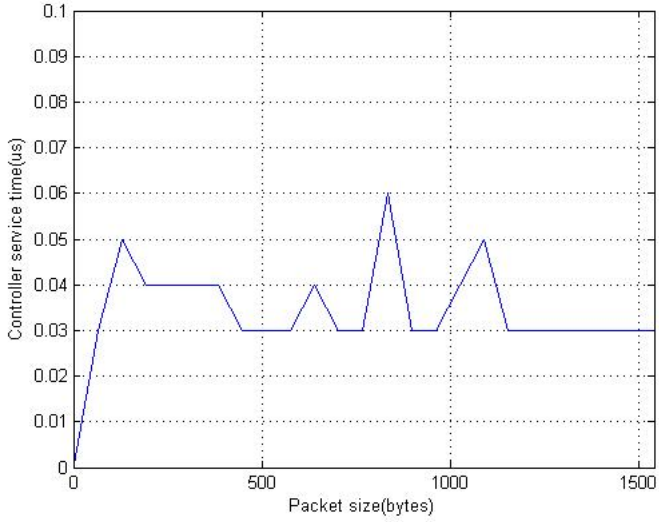

Figure 4.3: Controller service time

It can be seen from Figure 4.3 above that the controller service times of different packet sizes are a little bit variable. The average of the controller service time is 0.045μs.

Chapter 5
Conclusions and Future Work

In this master thesis, the details of OpenFlow system model is described and results of performance simulation are shown. Our goal is to do the performance simulation of OpenFlow system (switch and controller policy). On one hand, it is to implement the OpenFlow system model on the FPGA-based platform. On the other hand, the performance simulation is done in order to measure the sojourn time and the service time (switch and controller). In order to simulate the performace, data plane and control plane are both implemented on our FPFA-platform (Xilinx Virtex6) using ISE design tools. As the switch is for research, not for the market, four mainly components of the OpenFlow switch are implemented in our design, which are the flow entry composer module, the flow table controller module, the action processor module and the controller policy module. Besides, the packets are generated by the packet generator for measuring the performance metrics through the performance simulation test, specifically the switch service time, sojourn time and controller service time. As a major result, it is found that the sojourn time and the switch service time both have an almost linear increase with the increase in payload size. Moreover, the switch responds to the writing request from the controller policy module with the fixed latency of 2 cycles. Thus, it can be concluded that the communication time between the switch and the controller decreases in comparison with another FPGA-based OpenFlow switch, when the controller is also implemented on the FPGA-platform.

It is important to underline that findings above only apply to the study presented in this master thesis and cannot be generalized. Because there are some limitations about the OpenFlow switch implementation, which can prompt to the future work. Firstly, the FPGA-platform used in the OpenFlow switch implementation has only one Ethernet port. Secondly, the whole functions of OpenFlow switch doesn't completely be implemented as well as the entire functions of the controller. For example, only the forwarding action is implemented in OpenFlow switch part and only writing the flow entry is designed in the controller policy module. Finally,

the performance metrics are measured under simulation test environment through generating the packets on the board, not real-time Internet environment.

According to the limitation discussed above, there are lots of to do the future work. The performance metrics (e.g, switch service time, sojourn time and controller service time) of the OpenFlow switch can be measured under the real-life Internet environment in the future, and more performance metrics can be measured such as the lost rate, etc. Also, OpenFlow switch and controller can be implemented on the FPGA-platform with more Ethernet ports.

References

[1] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, (New York, NY, USA), pp. 1–9, ACM, 2008. http://doi.acm.org/10.1145/1477942.1477944.

[2] M. Wielgosz, M. Panggabean, J. Wang, and L. A. Rønningen, "An fpga-based platform for a network atchitecture with delay guarantee," *Journal of Circuits, Systems and Computers*, vol. 22, no. 06, p. 1350045, 2013. http://www.worldscientific.com/doi/abs/10.1142/S021812661350045X.

[3] ONF, "OpenFlow Switch Specification," Dec. 2011. http://goo.gl/tKo6r.

[4] K. Shahmir Shourmasti, "Stochastic switching using openflow," Master's thesis, Norwegian University of Science and Technology, Department of Telematics, 2013.

[5] Xilinx, *Getting Started with the Xilinx Virtex-6 FPGA ML605 Evaluation Kit*, Octorber 2011. http://www.xilinx.com/support/documentation/boards_and_kits/ug533.pdf.

[6] Xilinx, *ML605 Hardware User Guide*, 1.2.1 ed., January 2010. http://www.xilinx.com/support/documentation/boards_and_kits/ug534.pdf.

[7] O. Fundation, "Software-Defined Networking: The New Norm of Networks," 2012. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[8] H. Hata, "A study of requirements for sdn switch platform," in *Intelligent Signal Processing and Communications Systems (ISPACS), 2013 International Symposium on*, pp. 79–84, Nov 2013.

[9] NICIRA, "It's time to virtualize the network," *White Paper*, 2012. http://www.netfos.com.tw/PDF/Nicira/It%20is%20Time%20To%20Virtualize%20the%20Network%20White%20Paper.pdf.

[10] IBM, "Software defined networking," *IBM Systems and TechnologyThought Leadership White Paper*, Octorber 2012. http://ict.unimap.edu.my/images/doc/SDN%20IBM%20WhitePaper.pdf.

[11] NEC, *OpenFlow Feature Guide (IP8800/S3640)*, May 2010. http://support.necam.com/kbtools/sdocs.cfm?id=fcbdcb3e-45fa-4ec4-9311-215bd9ab9f81.

[12] G. Romero de Tejada Muntaner, "Evaluation of openflow controllers," Master's thesis, KTH, School of Information and Communication Technology (ICT), 2012.

[13] Xilinx, *Virtex-6 Family Overview*, January 2012. http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf.

[14] G. Antichi, A. Di Pietro, S. Giordano, G. Procissi, and D. Ficara, "Design and development of an openflow compliant smart gigabit switch," in *Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE*, pp. 1–5, Dec 2011.

[15] A. Khan and N. Dave, "Enabling hardware exploration in software-defined networking: A flexible, portable openflow switch," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, pp. 145–148, April 2013.

[16] V. Tanyingyong, M. Hidell, and P. Sjödin, "Improving pc-based openflow switching performance," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, (New York, NY, USA), pp. 13:1–13:2, ACM, 2010. http://doi.acm.org/10.1145/1872007.1872023, doi = 10.1145/1872007.1872023.

[17] Y. Luo, P. Cascon, E. Murray, and J. Ortega, "Accelerating openflow switching with network processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '09, (New York, NY, USA), pp. 70–71, ACM, 2009. http://doi.acm.org/10.1145/1882486.1882504.

[18] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, "Openflow switching: Data plane performance," in *Communications (ICC), 2010 IEEE International Conference on*, pp. 1–5, May 2010.

[19] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and performance evaluation of an openflow architecture," in *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pp. 1–7, International Teletraffic Congress, 2011. http://dl.acm.org/citation.cfm?id=2043468.2043470.

[20] Xilinx, *LogiCORE IP FIFO Generator v9.2 Product Guide*, July 2012. http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator/v9_2/pg057-fifo-generator.pdf.

[21] Xilinx, *LogiCORE IP Block Memory Generator v7.3 Product Guide*, December 2012. http://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v7_3/pg058-blk-mem-gen.pdf.

[22] Xilinx, *ISE Design Suite 14: Release Notes, Installation, and Licensing.*

Appendix A

OpenFlow Switch Top Level Module

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY openflow_switch IS
GENERIC (
    OPENFLOW_MATCH_SIZE: INTEGER:= 256;
    OPENFLOW_MASK_SIZE: INTEGER:= 256;
    OPENFLOW_ACTION_SIZE: INTEGER:= 256
    );
PORT (
    clk:IN STD_LOGIC;
    reset:IN STD_LOGIC;
    input_wr_en1:IN STD_LOGIC;
    input_wr_en2: IN STD_LOGIC;
    input_wr_en3: IN STD_LOGIC;
    input_wr_en4: IN STD_LOGIC;
    inputfifo_full1: OUT STD_LOGIC;
    inputfifo_full2: OUT STD_LOGIC;
    inputfifo_full3: OUT STD_LOGIC;
    inputfifo_full4: OUT STD_LOGIC;
    inputfifo_empty1: OUT STD_LOGIC;
    inputfifo_empty2: OUT STD_LOGIC;
    inputfifo_empty3: OUT STD_LOGIC;
    inputfifo_empty4: OUT STD_LOGIC;
    packet_in_port1: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_in_port2: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_in_port3: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_in_port4: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_out_port1: OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_out_port2: OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
```

```
    packet_out_port3: OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_out_port4: OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
END openflow_switch;


ARCHITECTURE openflow_switch of openflow_switch IS
.....Pre-processor 1.....
COMPONENT pre_processor
    PORT(
        asclk : IN STD_LOGIC;
        asresetn : IN STD_LOGIC;
        input_wr_en : IN STD_LOGIC;
        output_rd_en : IN STD_LOGIC;
        lu_ack : IN STD_LOGIC;
        packet_in : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        lu_req : INOUT STD_LOGIC;
        lu_entry : OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        outputbuffer_data : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        inputfifo_full : OUT STD_LOGIC;
        output_buffer_full : OUT STD_LOGIC;
        output_buffer_empty : OUT STD_LOGIC
        );
END COMPONENT;
.....Pre-processor 2.....
COMPONENT pre_processor2
    PORT(
        asclk : IN STD_LOGIC;
        asresetn : IN STD_LOGIC;
        input_wr_en : IN STD_LOGIC;
        output_rd_en : IN STD_LOGIC;
        lu_ack : IN STD_LOGIC;
        packet_in : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        lu_req : INOUT STD_LOGIC;
        lu_entry : OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        outputbuffer_data : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        inputfifo_full : OUT STD_LOGIC;
        output_buffer_full : OUT STD_LOGIC;
        output_buffer_empty : OUT STD_LOGIC
        );
```

```
END COMPONENT;
.....Pre-processor 3.....
COMPONENT pre_processor3
    PORT(
        asclk : IN STD_LOGIC;
        asresetn : IN STD_LOGIC;
        input_wr_en : IN STD_LOGIC;
        output_rd_en : IN STD_LOGIC;
        lu_ack : IN STD_LOGIC;
        packet_in : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        lu_req : INOUT STD_LOGIC;
        lu_entry : OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        outputbuffer_data : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        inputfifo_full : OUT STD_LOGIC;
        output_buffer_full : OUT STD_LOGIC;
        output_buffer_empty : OUT STD_LOGIC
        );
END COMPONENT;
.....Pre-processor 4.....
COMPONENT pre_processor4
    PORT(
        asclk : IN STD_LOGIC;
        asresetn : IN STD_LOGIC;
        input_wr_en : IN STD_LOGIC;
        output_rd_en : IN STD_LOGIC;
        lu_ack : IN STD_LOGIC;
        packet_in : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        lu_req : INOUT STD_LOGIC;
        lu_entry : OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        outputbuffer_data : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        inputfifo_full : OUT STD_LOGIC;
        output_buffer_full : OUT STD_LOGIC;
        output_buffer_empty : OUT STD_LOGIC
        );
END COMPONENT;
.....Flow Table Controller.....
COMPONENT flow_table_controller
    PORT(
        asclk : IN STD_LOGIC;
```

```
        asresetn : IN STD_LOGIC;
        lu_req1 : IN STD_LOGIC;
        lu_req2 : IN STD_LOGIC;
        lu_req3 : IN STD_LOGIC;
        lu_req4 : IN STD_LOGIC;
        lu_entry1 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        lu_entry2 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        lu_entry3 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        lu_entry4 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
        lu_done1 : INOUT STD_LOGIC;
        lu_done2 : INOUT STD_LOGIC;
        lu_done3 : INOUT STD_LOGIC;
        lu_done4 : INOUT STD_LOGIC;
        lu_ack1 : OUT STD_LOGIC;
        lu_ack2 : OUT STD_LOGIC;
        lu_ack3 : OUT STD_LOGIC;
        lu_ack4 : OUT STD_LOGIC;
        action: OUT STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 downto
0);
        match : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
        );
END COMPONENT;
.....-Packet forwarding.....
COMPONENT packet_forwarding
    PORT(
        asclk : IN STD_LOGIC;
        asresetn : IN STD_LOGIC;
        match : IN STD_LOGIC_VECTOR (3 DOWNTO 0);
        action:IN STD_LOGIC_VECTOR (OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
        output_buffer_empty1 : IN STD_LOGIC;
        output_buffer_empty2 : IN STD_LOGIC;
        output_buffer_empty3 : IN STD_LOGIC;
        output_buffer_empty4 : IN STD_LOGIC;
        packet_in1 : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        packet_in2 : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        packet_in3 : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
```

```
        packet_in4 : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        lu_done1 : IN STD_LOGIC;
        lu_done2 : IN STD_LOGIC;
        lu_done3 : IN STD_LOGIC;
        lu_done4 : IN STD_LOGIC;
        rd_en1 : OUT STD_LOGIC;
        rd_en2 : OUT STD_LOGIC;
        rd_en3 : OUT STD_LOGIC;
        rd_en4 : OUT STD_LOGIC;
        packet_out1 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        packet_out2 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        packet_out3 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        packet_out4 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
        );
END COMPONENT;
SIGNAL rd_en1_nxt,rd_en2_nxt,rd_en3_nxt,rd_en4_nxt: STD_LOGIC;
SIGNAL lu_req1_nxt,lu_req2_nxt,lu_req3_nxt,lu_req4_nxt: STD_LOGIC;
SIGNAL lu_done1_nxt,lu_done2_nxt,lu_done3_nxt,lu_done4_nxt: STD_LOGIC;
SIGNAL lu_ack1_nxt,lu_ack2_nxt, lu_ack3_nxt, lu_ack4_nxt: STD_LOGIC;
SIGNAL output_buffer_full1,output_buffer_full2, output_buffer_full3,
output_buffer_full4: STD_LOGIC;
SIGNAL output_buffer_empty1_nxt, output_buffer_empty2_nxt,output_buffer_empty3_nxt,out
STD_LOGIC;
SIGNAL match_nxt: STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL lu_entry1_nxt, lu_entry2_nxt, lu_entry3_nxt, lu_entry4_nxt: STD_LOGIC_VECTOR
(OPENFLOW_MATCH_SIZE-1 DOWNTO 0);
SIGNAL outputbuffer_data1,outputbuffer_data2,outputbuffer_data3,outputbuffer_data4:
STD_LOGIC_VECTOR (63 DOWNTO 0);
SIGNAL action_nxt : STD_LOGIC_VECTOR (OPENFLOW_ACTION_SIZE-1
DOWNTO 0);
BEGIN
.....Pre-processor 1.....
Inst_pre_processor: pre_processor PORT MAP(
    asclk => clk,
    asresetn => reset,
    input_wr_en => input_wr_en1,
    output_rd_en => rd_en1_nxt,
    lu_entry => lu_entry1_nxt,
    lu_req => lu_req1_nxt,
    lu_ack => lu_ack1_nxt,
    packet_in => packet_in_port1,
```

```
    outputbuffer_data => outputbuffer_data1,
    inputfifo_full => inputfifo_full1,
    output_buffer_full => output_buffer_full1,
    output_buffer_empty => output_buffer_empty1_nxt
    );
.....Pre-processor 2.....
Inst_pre_processor2: pre_processor2 PORT MAP(
    asclk => clk,
    asresetn => reset,
    input_wr_en => input_wr_en2,
    output_rd_en => rd_en2_nxt,
    lu_entry => lu_entry2_nxt,
    lu_req => lu_req2_nxt,
    lu_ack => lu_ack2_nxt,
    packet_in => packet_in_port2,
    outputbuffer_data => outputbuffer_data2,
    inputfifo_full => inputfifo_full2,
    output_buffer_full => output_buffer_full2,
    output_buffer_empty => output_buffer_empty2_nxt
    );
.....Pre-processor 3.....
Inst_pre_processor3: pre_processor3 PORT MAP(
    asclk => clk,
    asresetn => reset,
    input_wr_en => input_wr_en3,
    output_rd_en => rd_en3_nxt,
    lu_entry => lu_entry3_nxt,
    lu_req => lu_req3_nxt,
    lu_ack => lu_ack3_nxt,
    packet_in => packet_in_port3,
    outputbuffer_data => outputbuffer_data3,
    inputfifo_full => inputfifo_full3,
    output_buffer_full => output_buffer_full3,
    output_buffer_empty => output_buffer_empty3_nxt
    );
.....Pre-processor 4.....
Inst_pre_processor4: pre_processor4 PORT MAP(
    asclk => clk,
    asresetn => reset,
    input_wr_en => input_wr_en4,
    output_rd_en => rd_en4_nxt,
```

```
    lu_entry => lu_entry4_nxt,
    lu_req => lu_req4_nxt,
    lu_ack => lu_ack4_nxt,
    packet_in => packet_in_port4,
    outputbuffer_data => outputbuffer_data4,
    inputfifo_full => inputfifo_full4,
    output_buffer_full => output_buffer_full4,
    output_buffer_empty => output_buffer_empty4_nxt
    );
..... Flow Table Controller .....
Inst_flow_table_controller: flow_table_controller PORT MAP(
    asclk => clk,
    asresetn => reset,
    lu_req1 => lu_req1_nxt,
    lu_req2 => lu_req2_nxt,
    lu_req3 => lu_req3_nxt,
    lu_req4 => lu_req4_nxt,
    lu_entry1 => lu_entry1_nxt,
    lu_entry2 => lu_entry2_nxt,
    lu_entry3 => lu_entry3_nxt,
    lu_entry4 => lu_entry4_nxt,
    lu_done1 => lu_done1_nxt,
    lu_done2 => lu_done2_nxt,
    lu_done3 => lu_done3_nxt,
    lu_done4 => lu_done4_nxt,
    lu_ack1 => lu_ack1_nxt,
    lu_ack2 => lu_ack2_nxt,
    lu_ack3 => lu_ack3_nxt,
    lu_ack4 => lu_ack4_nxt,
    action => action_nxt,
    match => match_nxt
    );
..... Packet Forwarding .....
Inst_packet_forwarding: packet_forwarding PORT MAP(
    asclk => clk,
    asresetn => reset,
    match => match_nxt,
    action => action_nxt,
    lu_done1 => lu_done1_nxt,
    lu_done2 => lu_done2_nxt,
    lu_done3 => lu_done3_nxt,
```

```
    lu_done4 => lu_done4_nxt,
    output_buffer_empty1 => output_buffer_empty1_nxt,
    output_buffer_empty2 => output_buffer_empty2_nxt,
    output_buffer_empty3 => output_buffer_empty3_nxt,
    output_buffer_empty4 => output_buffer_empty4_nxt,
    rd_en1 => rd_en1_nxt,
    rd_en2 => rd_en2_nxt,
    rd_en3 => rd_en3_nxt,
    rd_en4 => rd_en4_nxt,
    packet_in1 => outputbuffer_data1,
    packet_in2 => outputbuffer_data2,
    packet_in3 => outputbuffer_data3,
    packet_in4 => outputbuffer_data4,
    packet_out1 => packet_out_port1,
    packet_out2 => packet_out_port2,
    packet_out3 => packet_out_port3,
    packet_out4 => packet_out_port4
    );
END openflow_switch;
```

Appendix B
Pre-processor Module

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY pre_processor is
GENERIC (
    OPENFLOW_MATCH_SIZE : integer := 256
    );
PORT (
    asclk : IN STD_LOGIC;
    asresetn : IN STD_LOGIC;
    input_wr_en: IN STD_LOGIC;
    output_rd_en: IN STD_LOGIC;
    lu_entry : OUT std_logic_vector (OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_req: INOUT STD_LOGIC;
    lu_ack : IN STD_LOGIC;
    packet_in: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    outputbuffer_data: OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
    inputfifo_full : OUT STD_LOGIC;
    inputfifo_empty : OUT STD_LOGIC;
    output_buffer_full: OUT STD_LOGIC;
    output_buffer_empty: OUT STD_LOGIC
    );
END pre_processor;
ARCHITECTURE pre_processor of pre_processor is
.....FIFO-input QUEUE.....
COMPONENT input_fifo_exdes
    PORT (
        clk : IN STD_LOGIC;
        rst : IN STD_LOGIC;
```

```
        din : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        wr_en : IN STD_LOGIC;
        rd_en : IN STD_LOGIC;
        dout : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        full : OUT STD_LOGIC;
        empty : OUT STD_LOGIC;
        almost_empty : OUT STD_LOGIC
        );
END COMPONENT;
.....Output Buffer.....
COMPONENT output_pkt_buffer_exdes
    PORT(
        CLK : IN STD_LOGIC;
        RST : IN STD_LOGIC;
        WR_EN : IN STD_LOGIC;
        RD_EN : IN STD_LOGIC;
        DIN : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
        DOUT : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
        FULL : OUT STD_LOGIC;
        EMPTY : OUT STD_LOGIC
        );
END COMPONENT;
.....Header Parser.....
COMPONENT header_parser
    PORT(
        asclk : IN STD_LOGIC;
        aresetn : IN STD_LOGIC;
        tx_data : IN STD_LOGIC_VECTOR(63 DOWNTO 0);
        fifo_empty : IN STD_LOGIC;
        almost_empty : IN STD_LOGIC;
        compose_done : IN STD_LOGIC;
        fifo_rd_en : OUT STD_LOGIC;
        dl_start : OUT STD_LOGIC;
        dl_done : OUT STD_LOGIC;
        src_port : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        dl_dst : OUT STD_LOGIC_VECTOR (47 DOWNTO 0);
        dl_src : OUT STD_LOGIC_VECTOR (47 DOWNTO 0);
        dl_ethtype : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        dl_vlantag : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        ip_tp_done : OUT STD_LOGIC;
        ipv6_tp_done : OUT STD_LOGIC;
```

```
        arp_done : OUT STD_LOGIC;
        mpls_done : OUT STD_LOGIC;
        ip_proto : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        ip_tos : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        ip_src : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        ip_dst : OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
        arp_opcode : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        arp_ip_src : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        arp_ip_dst : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        mpls_label : OUT STD_LOGIC_VECTOR (19 DOWNTO 0);
        mpls_tc : OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
        ipv6_src : OUT STD_LOGIC_VECTOR (127 DOWNTO 0);
        ipv6_dst : OUT STD_LOGIC_VECTOR (127 DOWNTO 0);
        tp_src : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        tp_dst : OUT STD_LOGIC_VECTOR (15 DOWNTO 0)
        );
END COMPONENT;
.............................. Lookup Entry Composer ..............................
COMPONENT lu_entry_composer
    PORT(
        asclk : IN STD_LOGIC;
        aresetn : IN STD_LOGIC;
        dl_start : IN STD_LOGIC;
        dl_done : IN STD_LOGIC;
        src_port : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        dl_dst : IN STD_LOGIC_VECTOR (47 DOWNTO 0);
        dl_src : IN STD_LOGIC_VECTOR (47 DOWNTO 0);
        dl_ethtype : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        dl_vlantag : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        ip_tp_done : IN STD_LOGIC;
        ipv6_tp_done : IN STD_LOGIC;
        arp_done : IN STD_LOGIC;
        mpls_done : IN STD_LOGIC;
        ip_proto : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        ip_tos : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        ip_src : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        ip_dst : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
        arp_opcode : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
        arp_ip_src : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        arp_ip_dst : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        mpls_label : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
```

```
        mpls_tc : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
        ipv6_src : IN STD_LOGIC_VECTOR (127 DOWNTO 0);
        ipv6_dst : IN STD_LOGIC_VECTOR (127 DOWNTO 0);
        tp_src : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        tp_dst : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        lu_ack : IN STD_LOGIC;
        compose_done : INOUT STD_LOGIC;
        lu_req : INOUT STD_LOGIC;
        lu_entry : OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0)
        );
END COMPONENT;


–input signals to pre_processor
SIGNAL rd_en_nxt, fifo_empty_nxt, almost_empty_nxt: STD_LOGIC;
–signals between pre_processor and lu_composer
SIGNAL dl_start_nxt, dl_done_nxt,ip_tp_done_nxt,arp_done_nxt,mpls_done_nxt,
ipv6_tp_done_nxt, compose_done_nxt : STD_LOGIC;
SIGNAL src_port_nxt: STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL ip_proto_nxt,ip_tos_nxt, arp_opcode_nxt: STD_LOGIC_VECTOR (7
DOWNTO 0);
SIGNAL dl_dst_nxt, dl_src_nxt: STD_LOGIC_VECTOR (47 DOWNTO 0);
SIGNAL dl_ethtype_nxt, dl_vlantag_nxt,arp_ip_src_nxt,arp_ip_dst_nxt, tp_src_nxt,
tp_dst_nxt: STD_LOGIC_VECTOR (15 DOWNTO 0);
SIGNAL ip_src_nxt, ip_dst_nxt: STD_LOGIC_VECTOR (31 DOWNTO 0);
SIGNAL ipv6_src_nxt, ipv6_dst_nxt: STD_LOGIC_VECTOR (127 DOWNTO
0);
SIGNAL tx_data_nxt : STD_LOGIC_VECTOR (63 DOWNTO 0);
SIGNAL mpls_label_nxt : STD_LOGIC_VECTOR (19 DOWNTO 0);
SIGNAL mpls_tc_nxt : STD_LOGIC_VECTOR (2 DOWNTO 0);


BEGIN
.....FIFO-input QUEUE.....
Inst_input_fifo_exdes : input_fifo_exdes PORT MAP (
    clk => asclk,
    rst => asresetn,
    din => packet_in,
    wr_en => input_wr_en,
    rd_en => rd_en_nxt,
        dout => tx_data_nxt,
```

```
        full => inputfifo_full,
        empty => fifo_empty_nxt,
        almost_empty => almost_empty_nxt
        );
.....Header Parser.....
Inst_header_parser: header_parser PORT MAP(
    asclk => asclk,
    aresetn => asresetn,
    tx_data => tx_data_nxt,
    fifo_empty => fifo_empty_nxt,
    almost_empty => almost_empty_nxt,
    fifo_rd_en => rd_en_nxt,
    dl_start => dl_start_nxt,
    dl_done => dl_done_nxt,
    src_port =>src_port_nxt,
    dl_dst => dl_dst_nxt,
    dl_src => dl_src_nxt,
    dl_ethtype => dl_ethtype_nxt,
    dl_vlantag => dl_vlantag_nxt,
    ip_tp_done => ip_tp_done_nxt,
    arp_done => arp_done_nxt,
    mpls_done => mpls_done_nxt,
    ipv6_tp_done => ipv6_tp_done_nxt,
    ip_proto => ip_proto_nxt,
    ip_tos => ip_tos_nxt,
    ip_src => ip_src_nxt,
    ip_dst => ip_dst_nxt,
    arp_opcode => arp_opcode_nxt,
    arp_ip_src => arp_ip_src_nxt,
    arp_ip_dst => arp_ip_dst_nxt,
    mpls_label => mpls_label_nxt,
    mpls_tc => mpls_tc_nxt,
    ipv6_src => ipv6_src_nxt,
    ipv6_dst => ipv6_dst_nxt,
    tp_src => tp_src_nxt,
    tp_dst => tp_dst_nxt,
    compose_done => compose_done_nxt
    );
.....Lookup Entry Composer.....
Inst_lu_entry_composer: lu_entry_composer PORT MAP(
    asclk => asclk,
```

```
    aresetn => asresetn,
    dl_start => dl_start_nxt,
    dl_done => dl_done_nxt,
    src_port => src_port_nxt,
    dl_dst => dl_dst_nxt,
    dl_src => dl_src_nxt,
    dl_ethtype => dl_ethtype_nxt,
    dl_vlantag => dl_vlantag_nxt,
    ip_tp_done => ip_tp_done_nxt,
    arp_done => arp_done_nxt,
    mpls_done => mpls_done_nxt,
    ipv6_tp_done => ipv6_tp_done_nxt,
    ip_proto => ip_proto_nxt,
    ip_tos => ip_tos_nxt,
    ip_src => ip_src_nxt,
    ip_dst => ip_dst_nxt,
    arp_opcode => arp_opcode_nxt,
    arp_ip_src => arp_ip_src_nxt,
    arp_ip_dst => arp_ip_dst_nxt,
    mpls_label => mpls_label_nxt,
    mpls_tc => mpls_tc_nxt,
    ipv6_src => ipv6_src_nxt,
    ipv6_dst => ipv6_dst_nxt,
    tp_src => tp_src_nxt,
    tp_dst => tp_dst_nxt,
    lu_ack => lu_ack,
    compose_done => compose_done_nxt,
    lu_entry => lu_entry,
    lu_req => lu_req
    );
.....Output Buffer.....
Inst_output_pkt_buffer_exdes: output_pkt_buffer_exdes PORT MAP (
    clk => asclk,
    rst => asresetn,
    din => packet_in,
    wr_en => input_wr_en,
    rd_en => output_rd_en,
    dout => outputbuffer_data,
    full => output_buffer_full,
    empty => output_buffer_empty
    );
```

```
END pre_processor;
```

Appendix C
Header Parser Block

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
ENTITY header_parser IS
GENERIC (
    C_AXIS_DATA_WIDTH: INTEGER :=64;
    TYPE_VLAN: STD_LOGIC_VECTOR (15 DOWNTO 0):= X"8100";
    TYPE_VLAN_QINQ: STD_LOGIC_VECTOR (15 DOWNTO 0):= X"88a8";
    TYPE_IP: STD_LOGIC_VECTOR (15 DOWNTO 0):= X"0800";
    TYPE_IPV6: STD_LOGIC_VECTOR (15 DOWNTO 0):= X"86dd";
    TYPE_ARP: STD_LOGIC_VECTOR (15 DOWNTO 0):= X"0806";
    TYPE_MPLS: STD_LOGIC_VECTOR (15 DOWNTO 0):= X"8847";
    TYPE_MPLS_MU: STD_LOGIC_VECTOR( 15 DOWNTO 0):= X"8848"
    );
PORT (
    asclk: IN STD_LOGIC;
    aresetn: IN STD_LOGIC;
    tx_data: IN STD_LOGIC_VECTOR (C_AXIS_DATA_WIDTH-1 DOWNTO
0);
    fifo_empty: IN STD_LOGIC;
    almost_empty: IN STD_LOGIC;
    fifo_rd_en: OUT STD_LOGIC;
    dl_start: OUT STD_LOGIC;
    dl_done: OUT STD_LOGIC;
    src_port: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    dl_dst: OUT STD_LOGIC_VECTOR (47 DOWNTO 0);
    dl_src: OUT STD_LOGIC_VECTOR (47 DOWNTO 0);
    dl_ethtype: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    dl_vlantag: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    ip_tp_done: OUT STD_LOGIC;
```

```
    ipv6_tp_done: OUT STD_LOGIC;
    arp_done: OUT STD_LOGIC;
    mpls_done: OUT STD_LOGIC;
    ip_proto: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    ip_tos: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    ip_src: OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    ip_dst: OUT STD_LOGIC_VECTOR (31 DOWNTO 0);
    arp_opcode: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
    arp_ip_src: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    arp_ip_dst: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    mpls_label: OUT STD_LOGIC_VECTOR (19 DOWNTO 0);
    mpls_tc: OUT STD_LOGIC_VECTOR (2 DOWNTO 0);
    ipv6_src: OUT STD_LOGIC_VECTOR (127 DOWNTO 0);
    ipv6_dst: OUT STD_LOGIC_VECTOR (127 DOWNTO 0);
    tp_src: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    tp_dst: OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
    compose_done: IN STD_LOGIC
    );
END header_parser;


ARCHITECTURE header_parser of header_parser IS
TYPE dt_td_type is (DT_RD_1ST, DT_RD_REST, DT_RD_WAIT);
TYPE parse_type is ( DL_WAIT_TVALID, DL_PARSE_2ND, DL_SFT16_1ST,
DL_SFT48_1ST, DL_PARSE_MORE, IP_TP_PARSE_16_1ST,IP_TP_PARSE_16_2ND,
IP_TP_PARSE_16_3RD, IPV6_TP_PARSE_16_1ST,IPV6_TP_PARSE_16_2ND,
IPV6_TP_PARSE_16_3RD, IPV6_TP_PARSE_16_4TH, IPV6_TP_PARSE_16_5TH,
IP_TP_PARSE_48_1ST, IP_TP_PARSE_48_2ND, IP_TP_PARSE_48_3RD,
IPV6_TP_PARSE_48_1ST, IPV6_TP_PARSE_48_2ND, IPV6_TP_PARSE_48_3RD,
IPV6_TP_PARSE_48_4TH, ARP_PARSE_16, ARP_PARSE_48, DL_SFT_MORE,
DL_SFT_LAST);


SIGNAL dt_rd_state, dt_rd_state_nxt :dt_td_type;
SIGNAL parse_state, parse_state_nxt: parse_type;


BEGIN
.....Data Reading Process.....
PROCESS(asclk, aresetn, fifo_empty, dt_rd_state)
BEGIN
    IF (aresetn = '1') THEN
```

```
        fifo_rd_en <= '0';
        dt_rd_state <= DT_RD_1ST;
    ELSIF (asclk'event and asclk = '1') THEN
        dt_rd_state <= dt_rd_state_nxt;
    END IF;
    CASE dt_rd_state IS
        WHEN DT_RD_1ST =>
            IF (fifo_empty ='0') THEN
                fifo_rd_en <= '1';
                dt_rd_state_nxt <= DT_RD_REST;
            END IF;
        WHEN DT_RD_REST =>
            IF (fifo_empty ='0') THEN
                fifo_rd_en <= '1';
                IF (almost_empty ='1') THEN
                    dt_rd_state_nxt <= DT_RD_WAIT;
                END IF;
            END IF;
        WHEN DT_RD_WAIT =>
            dt_rd_state_nxt <= DT_RD_1ST;
    END CASE;
END PROCESS;
.....L2 Parser Process.....
packet_parsing: PROCESS(asclk, aresetn,parse_state)
VARIABLE ip_hlen_nxt :STD_LOGIC_VECTOR(3 DOWNTO 0);
VARIABLE ip_proto_nxt,ipv6_proto_nxt : STD_LOGIC_VECTOR(7 DOWNTO
0);
VARIABLE dl_ethtype_nxt: STD_LOGIC_VECTOR (15 DOWNTO 0);
BEGIN
    IF (aresetn = '1') THEN
        src_port(7 DOWNTO 0)<= (others =>'0');
        dl_dst(47 DOWNTO 0) <= (others =>'0');
        dl_src(47 DOWNTO 0)<= (others =>'0');
        dl_ethtype(15 DOWNTO 0) <= (others =>'0');
        dl_start <='0';
        dl_done <= '0';
        dl_vlantag <= (others =>'0');
        ip_proto (7 DOWNTO 0)<= (others =>'0');
        ip_tos (7 DOWNTO 0)<= (others =>'0');
        ip_src(31 DOWNTO 0)<= (others =>'0');
        ip_dst(31 DOWNTO 0)<= (others =>'0');
```

```
        ipv6_src(127 DOWNTO 0)<= (others =>'0');
        ipv6_dst(127 DOWNTO 0)<= (others =>'0');
        arp_ip_src <= (others =>'0');
        arp_ip_dst <= (others =>'0');
        arp_opcode <= (others =>'0');
        mpls_label <= (others =>'0');
        mpls_tc <= (others =>'0');
        tp_src(15 DOWNTO 0)<= (others =>'0');
        tp_dst(15 DOWNTO 0)<= (others =>'0');
        ip_tp_done <= '0';
        ipv6_tp_done <= '0';
        arp_done <= '0';
        mpls_done <= '0';
        parse_state <= DL_WAIT_TVALID;
    ELSIF (asclk'event and asclk ='1') THEN
        parse_state<=parse_state_nxt;
    END IF;
    CASE parse_state IS
        WHEN DL_WAIT_TVALID =>
            IF (fifo_empty = '0') THEN
                src_port <= X"01";
..... Get Ethernet destination and source .....
                dl_dst(47 DOWNTO 0) <= tx_data(63 DOWNTO 16);
                dl_src(47 DOWNTO 32)<= tx_data(15 DOWNTO 0);
                dl_start <= '1';
                parse_state_nxt <= DL_PARSE_2ND;
            ELSE
                parse_state_nxt <= DL_WAIT_TVALID;
            END IF;
        WHEN DL_PARSE_2ND =>
            IF (fifo_empty = '0') THEN
                dl_src(31 DOWNTO 0)<= tx_data (63 DOWNTO 32);
                IF (tx_data(31 DOWNTO 16) = TYPE_VLAN_QINQ or tx_data(31
DOWNTO 16) = TYPE_VLAN ) THEN
..... Get Ethernet Type and VLAN Tag .....
                    dl_vlantag <= tx_data(15 DOWNTO 0);
                    parse_state_nxt <= DL_PARSE_MORE;
                ELSE
                    dl_ethtype(15 DOWNTO 0) <= tx_data(31 DOWNTO 16);
                    dl_ethtype_nxt(15 DOWNTO 0) := tx_data(31 DOWNTO 16);
                    mpls_tc <= tx_data(6 DOWNTO 4);
```

```
                    mpls_label(19 DOWNTO 16)<= tx_data(3 DOWNTO 0);
                    ip_hlen_nxt:= tx_data(11 DOWNTO 8);
                    ip_tos <= tx_data(7 DOWNTO 0);
                    parse_state_nxt <= DL_SFT16_1ST;
                    END IF;
                ELSE
                    parse_state_nxt <= DL_WAIT_TVALID;
            END IF;
        WHEN DL_PARSE_MORE =>
            IF (fifo_empty = '0') THEN
                IF ( tx_data(63 DOWNTO 48) = TYPE_VLAN_QINQ or tx_data(63
DOWNTO 48) = TYPE_VLAN) THEN
                    dl_vlantag <= tx_data(47 DOWNTO 32);
                    dl_ethtype <= tx_data(31 DOWNTO 16);
                    dl_ethtype_nxt := tx_data(31 DOWNTO 16);
                    ip_hlen_nxt:= tx_data(11 DOWNTO 8);
                    ip_tos <= tx_data(7 DOWNTO 0);
                    mpls_tc <= tx_data(6 DOWNTO 4);
                    mpls_label(19 downto 16)<= tx_data(3 DOWNTO 0);
                    parse_state_nxt <= DL_SFT16_1ST;
                ELSE
                    dl_ethtype <= tx_data(63 DOWNTO 48);
                    dl_ethtype_nxt := tx_data(63 DOWNTO 48);
                    ip_hlen_nxt:= tx_data(43 DOWNTO 40);
                    ip_tos <= tx_data(39 DOWNTO 32);
                    mpls_tc <= tx_data(38 DOWNTO 36);
                    mpls_label(19 DOWNTO 0) <= tx_data(35 DOWNTO 16);
                    parse_state_nxt <= DL_SFT48_1ST;
                END IF;
            ELSE
                parse_state_nxt <= DL_WAIT_TVALID;
        END IF;
        WHEN DL_SFT16_1ST =>
            IF (fifo_empty = '0') THEN
                IF (dl_ethtype_nxt = TYPE_IP) THEN
..... Get IP Protocol, SRC, DST .....
                    ip_proto(7 DOWNTO 0) <= tx_data (7 DOWNTO 0);
                    ip_proto_nxt(7 DOWNTO 0) := tx_data (7 DOWNTO 0);
                    parse_state_nxt <= IP_TP_PARSE_16_1ST;
                ELSIF (dl_ethtype_nxt = TYPE_IPV6) THEN
                    ip_proto (7 DOWNTO 0) <= tx_data(31 DOWNTO 24);
```

```
                ip_proto_nxt (7 DOWNTO 0) := tx_data(31 DOWNTO 24);
                ipv6_src (127 DOWNTO 112) <= tx_data (15 DOWNTO 0);
                parse_state_nxt <= IPV6_TP_PARSE_16_1ST;
.....Get ARP opcode.....
            ELSIF (dl_ethtype_nxt = TYPE_ARP) then
                arp_opcode <= tx_data (23 DOWNTO 16);
                parse_state_nxt <= ARP_PARSE_16;
.....Get MPLS label.....
            ELSIF (dl_ethtype_nxt = TYPE_MPLS or dl_ethtype_nxt = TYPE_MPLS_MU)
THEN
                mpls_label (15 DOWNTO 0)<= tx_data (63 DOWNTO 48);
                dl_done <='1';
                mpls_done <='1';
                parse_state_nxt <= DL_SFT_MORE;
            ELSE
                dl_done <='1';
                ip_tp_done <= '0';
                parse_state_nxt <= DL_SFT_MORE;
            END IF;
        END IF;
    WHEN ARP_PARSE_16 =>
        IF (fifo_empty = '0') THEN
.....Get ARP SRC,DST.....
            arp_ip_src <= tx_data (47 DOWNTO 32);
            arp_ip_dst <= tx_data (15 DOWNTO 0);
            dl_done <='1';
            arp_done <= '1';
            parse_state_nxt <= DL_SFT_MORE;
        END IF;
    WHEN IP_TP_PARSE_16_1ST =>
        IF (fifo_empty = '0') then
            ip_src (31 downto 0) <= tx_data(47 DOWNTO 16);
            ip_dst (31 downto 16) <= tx_data(15 DOWNTO 0);
            parse_state_nxt <= IP_TP_PARSE_16_2ND;
        END IF;
    WHEN IPV6_TP_PARSE_16_1ST =>
        IF (fifo_empty = '0') then
            ipv6_src (111 DOWNTO 48)<= tx_data (63 DOWNTO 0);
            parse_state_nxt <= IPV6_TP_PARSE_16_2ND;
        END IF;
    WHEN IP_TP_PARSE_16_2ND =>
```

```
            IF (fifo_empty = '0') THEN
                ip_dst(15 DOWNTO 0) <= tx_data(63 DOWNTO 48);
                IF (ip_proto_nxt = X"06" or ip_proto_nxt = X"11" or ip_proto_nxt
= X"84") THEN
                    IF (ip_hlen_nxt = B"0101") THEN
 ..... Get L4 header information .....
                        tp_src(15 DOWNTO 0) <= tx_data(47 DOWNTO 32);
                        tp_dst(15 DOWNTO 0) <= tx_data(31 DOWNTO 16);
                        dl_done <='1';
                        ip_tp_done <= '1';
                        parse_state_nxt <= DL_SFT_MORE;
                    ELSE
                        tp_src(15 DOWNTO 0) <= tx_data (15 DOWNTO 0);
                        parse_state_nxt <= IP_TP_PARSE_16_3RD;
                    END IF;
                ELSIF (ip_proto_nxt = X"01") then
                    IF (ip_hlen_nxt = B"0101") then
 ..... Get L4 header information .....
                        tp_src (15 DOWNTO 0) <= X"00" & tx_data (47 DOWNTO
40);
                        tp_dst (15 DOWNTO 0) <= X"00" & tx_data (39 DOWNTO
32);
                        dl_done <='1';
                        ip_tp_done <= '1';
                        parse_state_nxt <= DL_SFT_MORE;
                    ELSE
 ..... Get L4 header information .....
                        tp_src(15 downto 0) <= X"00" & tx_data(15 DOWNTO 8);
                        tp_dst(15 downto 0) <= X"00" & tx_data(7 DOWNTO 0);
                        dl_done <='1';
                        ip_tp_done <= '1';
                        parse_state_nxt <= DL_SFT_MORE;
                    END IF;
                ELSE
                    tp_src <= X"0000";
                    tp_dst <= X"0000";
                    dl_done <='1';
                    ip_tp_done <='1';
                    parse_state_nxt <= DL_SFT_MORE;
                END IF;
            END IF;
```

```
WHEN IPV6_TP_PARSE_16_2ND =>
    IF (fifo_empty = '0') THEN
        ipv6_src (47 DOWNTO 0) <= tx_data(63 DOWNTO 16);
        ipv6_dst (127 DOWNTO 112) <= tx_data(15 DOWNTO 0);
        parse_state_nxt <= IPV6_TP_PARSE_16_3RD;
    END IF;
WHEN IP_TP_PARSE_16_3RD =>
    IF (fifo_empty = '0') THEN
        tp_dst (15 DOWNTO 0) <= tx_data (63 downto 48);
        dl_done <='1';
        ip_tp_done <= '1';
        parse_state_nxt <= DL_SFT_MORE;
    END IF;
WHEN IPV6_TP_PARSE_16_3RD =>
    IF (fifo_empty = '0') then
        ipv6_dst (111 DOWNTO 48) <= tx_data(63 DOWNTO 0);
        parse_state_nxt <= IPV6_TP_PARSE_16_4TH;
    END IF;
WHEN IPV6_TP_PARSE_16_4TH =>
    IF (fifo_empty = '0') then
        ipv6_dst(47 DOWNTO 0) <= tx_data(63 DOWNTO 16);
        IF (ip_proto_nxt = X"06" or ip_proto_nxt = X"11") THEN
            tp_src(15 DOWNTO 0) <= tx_data(15 DOWNTO 0);
            parse_state_nxt <= IPV6_TP_PARSE_16_5TH;
        ELSE
            tp_src <= X"0000";
            tp_dst <= X"0000";
            dl_done <='1';
            ipv6_tp_done <='1';
            parse_state_nxt <= DL_SFT_MORE;
        END IF;
    END IF;
WHEN IPV6_TP_PARSE_16_5TH =>
    tp_dst (15 downto 0) <= tx_data(63 DOWNTO 48);
    dl_done <='1';
    ipv6_tp_done <='1';
    parse_state_nxt <= DL_SFT_MORE;
WHEN DL_SFT48_1ST =>
    IF (fifo_empty = '0') THEN
        IF (dl_ethtype_nxt = TYPE_IP) THEN
            ip_proto_nxt := tx_data (39 DOWNTO 32);
```

```
                ip_proto(7 DOWNTO 0) <= tx_data (39 DOWNTO 32);
                ip_src(31 DOWNTO 16) <= tx_data(15 DOWNTO 0);
                parse_state_nxt <= IP_TP_PARSE_48_1ST;
            ELSIF (dl_ethtype_nxt = TYPE_IPV6) THEN
                ip_proto_nxt := tx_data(63 DOWNTO 56);
                ip_proto(7 DOWNTO 0) <= tx_data(63 DOWNTO 56);
                ipv6_src(127 DOWNTO 80) <= tx_data(47 DOWNTO 0);
                parse_state_nxt <= IPV6_TP_PARSE_48_1ST;
            ELSIF (dl_ethtype_nxt = TYPE_ARP) then
                arp_opcode <= tx_data(53 DOWNTO 46);
                arp_ip_src <= tx_data(31 DOWNTO 16);
                parse_state_nxt <= ARP_PARSE_48;
            ELSIF (dl_ethtype_nxt = TYPE_MPLS or dl_ethtype_nxt = TYPE_MPLS_MU)
            THEN
                dl_done <='1';
                mpls_done <='1';
                parse_state_nxt <= DL_SFT_MORE;
            ELSE
                dl_done <='1';
                ip_tp_done <= '0';
                parse_state_nxt <= DL_SFT_MORE;
            END IF;
        END IF;
    WHEN ARP_PARSE_48 =>
        IF (fifo_empty = '0') THEN
            arp_ip_dst <= tx_data(63 DOWNTO 48);
            dl_done <='1';
            arp_done <= '1';
            parse_state_nxt <= DL_SFT_MORE;
        END IF;
    WHEN IP_TP_PARSE_48_1ST =>
        IF (fifo_empty = '0') then
            ip_src(15 DOWNTO 0) <= tx_data(63 DOWNTO 48);
            ip_dst(31 DOWNTO 0) <= tx_data(47 DOWNTO 16);
            IF (ip_proto_nxt = X"06" or ip_proto_nxt = X"11" or ip_proto_nxt
= X"84") THEN
                IF (ip_hlen_nxt = B"0101") THEN
                    tp_src(15 DOWNTO 0) <= tx_data(15 DOWNTO 0);
                    parse_state_nxt <= IP_TP_PARSE_48_2ND;
                ELSE
                    parse_state_nxt <= IP_TP_PARSE_48_3RD;
```

```
                END IF;
            ELSIF (ip_proto_nxt = X"01") then
                IF (ip_hlen_nxt = B"0101") then
                    tp_src(15 DOWNTO 0) <= X"00" & tx_data(15 DOWNTO
8);
                    tp_dst(15 DOWNTO 0) <= X"00" & tx_data(7 DOWNTO 0);
                    dl_done <='1';
                    ip_tp_done <= '1';
                ELSE
                    parse_state_nxt <= IP_TP_PARSE_48_3RD;
                END IF;
            ELSE
                tp_src<= X"0000";
                tp_dst <= X"0000";
                dl_done <='1';
                ip_tp_done<='1';
                parse_state_nxt <= DL_SFT_MORE;
            END IF;
        END IFf;
    WHEN IPV6_TP_PARSE_48_1ST =>
        IF (fifo_empty = '0') THEN
            ipv6_src (79 DOWNTO 16) <= tx_data(63 DOWNTO 0);
            parse_state_nxt <= IPV6_TP_PARSE_48_2ND;
        END IF;
    WHEN IP_TP_PARSE_48_2ND =>
        IF (fifo_empty = '0') THEN
            tp_dst(15 DOWNTO 0) <= tx_data(63 DOWNTO 48);
            dl_done <='1';
            ip_tp_done <='1';
            parse_state_nxt <= DL_SFT_MORE;
        END IF;
    WHEN IPV6_TP_PARSE_48_2ND =>
        IF (fifo_empty = '0') then
            ipv6_src(15 DOWNTO 0) <= tx_data(63 DOWNTO 48);
            ipv6_dst(127 DOWNTO 80) <= tx_data(47 DOWNTO 0);
            parse_state_nxt <= IPV6_TP_PARSE_48_3RD;
        END IF;
    WHEN IP_TP_PARSE_48_3RD =>
        IF (fifo_empty = '0') THEN
            IF (ip_proto_nxt = X"06" or ip_proto_nxt = X"11" or ip_proto_nxt
= X"84") THEN
```

```
            tp_src(15 DOWNTO 0) <= tx_data(47 DOWNTO 32);
            tp_dst(15 DOWNTO 0) <= tx_data(31 DOWNTO 16);
            dl_done <='1';
            ip_tp_done <= '1';
            parse_state_nxt <= DL_SFT_MORE;
        ELSIF (ip_proto_nxt = X"01") then
            tp_src(15 DOWNTO 0) <= X"00" & tx_data(47 DOWNTO 40);
            tp_dst(15 DOWNTO 0) <= X"00" & tx_data(39 DOWNTO 32);
            dl_done <='1';
            ip_tp_done <= '1';
            parse_state_nxt <= DL_SFT_MORE;
        END IF;
    END IF;
WHEN IPV6_TP_PARSE_48_3RD =>
    IF (fifo_empty = '0') THEN
        ipv6_dst (79 DOWNTO 16) <= tx_data (63 DOWNTO 0);
        parse_state_nxt <= IPV6_TP_PARSE_48_4TH;
    END IF;
WHEN IPV6_TP_PARSE_48_4TH =>
    IF (fifo_empty = '0') THEN
        ipv6_dst (15 DOWNTO 0) <= tx_data (63 DOWNTO 48);
        IF (ip_proto_nxt = X"06" or ip_proto_nxt = X"11") THEN
            tp_src (15 DOWNTO 0) <= tx_data (47 DOWNTO 32);
            tp_dst (15 DOWNTO 0) <= tx_data (31 DOWNTO 16);
            dl_done <='1';
            ipv6_tp_done <='1';
            parse_state_nxt <= DL_SFT_MORE;
         ELSE
            tp_src <= X"0000";
            tp_dst <= X"0000";
            dl_done <='1';
            ipv6_tp_done <='1';
            parse_state_nxt <= DL_SFT_MORE;
        END IF;
    END IF;
WHEN DL_SFT_MORE =>
    IF (fifo_empty = '0' and almost_empty = '1') THEN
        parse_state_nxt <= DL_SFT_LAST;
    ELSIF (fifo_empty = '0' and almost_empty = '0') THEN
        parse_state <= DL_SFT_MORE;
    ELSIF (fifo_empty = '1') then
```

```
                parse_state_nxt <= DL_WAIT_TVALID;
            END IF;
        WHEN DL_SFT_LAST =>
            IF (compose_done = '1') THEN
                parse_state_nxt <= DL_WAIT_TVALID;
            ELSE
                parse_state_nxt <= DL_SFT_LAST;
            END IF;
        END CASE;
END process packet_parsing;
END header_parser;
```

Appendix D
Lookup Entry Composer Block

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY lu_entry_composer IS
GENERIC(
    OPENFLOW_MATCH_SIZE : INTEGER := 256
    );
PORT (      asclk : IN STD_LOGIC;
    aresetn : IN STD_LOGIC;
    dl_start : IN STD_LOGIC;
    dl_done : IN STD_LOGIC;
    src_port : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    dl_dst : IN STD_LOGIC_VECTOR (47 DOWNTO 0);
    dl_src : IN STD_LOGIC_VECTOR (47 DOWNTO 0);
    dl_ethtype : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    dl_vlantag : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    ip_tp_done : IN STD_LOGIC;
    ipv6_tp_done : IN STD_LOGIC;
    mpls_done: IN STD_LOGIC;
    arp_done : IN STD_LOGIC;
    arp_opcode: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    arp_ip_src: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    arp_ip_dst: IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    ip_proto : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    ip_tos :IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    ip_src : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    ip_dst : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
    ipv6_src: IN STD_LOGIC_VECTOR (127 DOWNTO 0);
    ipv6_dst: IN STD_LOGIC_VECTOR (127 DOWNTO 0);
    mpls_label : IN STD_LOGIC_VECTOR (19 DOWNTO 0);
```

```
    mpls_tc : IN STD_LOGIC_VECTOR (2 DOWNTO 0);
    tp_src : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    tp_dst : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
    lu_ack : IN STD_LOGIC;
    compose_done : INOUT STD_LOGIC;
    lu_entry : OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
    lu_req : INOUT STD_LOGIC);
END lu_entry_composer;


ARCHITECTURE lu_entry_composer of lu_entry_composer IS
.....parsing-status check state machine.....
TYPE sc_type IS (SC_WAIT_PARSE_START, SC_WAIT_CMP_DONE);
SIGNAL sc_state, sc_state_nxt : sc_type;
SIGNAL parse_started: STD_LOGIC;
.....request latch state machine.....
TYPE req_latch_type is (RL_WAIT_PARSE_DONE, RL_WAIT_REQ);
SIGNAL req_latch_state, req_latch_state_nxt :req_latch_type;
SIGNAL parse_result : STD_LOGIC_VECTOR (1 DOWNTO 0);
SIGNAL int_entry : STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
SIGNAL lu_req_prev : STD_LOGIC;
SIGNAL int_req, lu_req_pre : STD_LOGIC;
.....flowtable interface state machine.....
TYPE lookup_inf_type is (LU_WAIT_INT_REQ, LU_WAIT_ACK);
SIGNAL lookup_inf_state, lookup_inf_state_nxt : lookup_inf_type;


BEGIN
.....parsing status check.....
PROCESS (aresetn, asclk, dl_start,compose_done,sc_state, sc_state_nxt)
BEGIN
    IF (aresetn = '1') THEN
        parse_started <= '0';
        sc_state <= SC_WAIT_PARSE_START;
    ELSIF (asclk'event and asclk = '1') THEN
        sc_state <= sc_state_nxt;
    END IF;
    CASE sc_state IS
        WHEN SC_WAIT_PARSE_START =>
            IF (dl_start = '1') THEN
```

```
                parse_started <= '1';
                sc_state_nxt <= SC_WAIT_CMP_DONE;
            ELSE
                sc_state_nxt <= SC_WAIT_PARSE_START;
            END IF;
    WHEN SC_WAIT_CMP_DONE =>
        IF (compose_done = '1') THEN
            parse_started <= '0';
            sc_state_nxt <= SC_WAIT_PARSE_START;
        ELSE
            sc_state_nxt <= SC_WAIT_CMP_DONE;
        END IF;
    END CASE;
END PROCESS;


..... Request-latch state machine .....
..... Only one state machine will reply during one packet parsing process .....
PROCESS (asclk, aresetn)
BEGIN
    IF (aresetn = '1') THEN
        lu_req_prev <= '0';
    ELSIF (asclk'event and asclk = '1') THEN
        lu_req_prev <= lu_req;
    END IF;
END PROCESS;
PROCESS (asclk, aresetn, dl_done,ip_tp_done,lu_req,req_latch_state, req_latch_state_nxt,
parse_started, lu_req_prev)
VARIABLE int_req_nxt:STD_LOGIC;
BEGIN
    IF (aresetn = '1') THEN
        int_req <= '0';
        int_entry <= (others =>'0');
        compose_done <= '0';
        req_latch_state <= RL_WAIT_PARSE_DONE;
    ELSIF (asclk'event and asclk = '1') THEN
        int_req <= int_req_nxt;
        req_latch_state <= req_latch_state_nxt;
    END IF;
    CASE req_latch_state IS
        WHEN RL_WAIT_PARSE_DONE =>
            IF (parse_started ='1') THEN
```

```
IF (dl_done = '1' and ip_tp_done = '0') THEN
    int_entry<=
        src_port
        & dl_src
        & dl_dst
        & dl_ethtype
        & dl_vlantag
        & X"00000000" –ipv4_src
        & X"00000000" –ipv4_dst
        & X"00" –ipv4_proto
        & X"00"–ipv4_tos
        & X"0000" –tp_src
        & X"0000" –tp_dst
        & X"00";–pad
    compose_done <= '1';
    int_req_nxt := '1';
    req_latch_state_nxt <= RL_WAIT_REQ;
ELSIF (dl_done = '1' and arp_done = '1') then
    int_entry <=
        src_port
        & dl_src
        & dl_dst
        & dl_ethtype
        & dl_vlantag
        & arp_ip_src –arp_src
        & arp_ip_dst –arp_dst
        & arp_opcode –arp
        & X"00"–ipv4_tos
        & X"0000" –tp_src
        & X"0000" –tp_dst
        & X"0000000000";–pad
    compose_done <= '1';
    int_req_nxt := '1';
    req_latch_state_nxt <= RL_WAIT_REQ;
ELSIF (dl_done = '1' and mpls_done = '1') then
    int_entry <=
        src_port
        & dl_src
        & dl_dst
        & dl_ethtype
        & dl_vlantag
```

```
                    & mpls_label
                    & mpls_tc
                    & X"00000000" –ipv4_src
                    & X"00000000" –ipv4_dst
                    & X"0000" –tp_src
                    & X"0000"–tp_dst
                    & B"0";
                compose_done <= '1';
                int_req_nxt := '1';
                req_latch_state_nxt <= RL_WAIT_REQ;
            ELSIF (dl_done = '1' and ip_tp_done = '1') then
                int_entry <=
                    src_port
                    & dl_src
                    & dl_dst
                    & dl_ethtype
                    & dl_vlantag
                    & ip_src –ipv4_src
                    & ip_dst –ipv4_dst
                    & ip_proto –ipv4_proto
                    & ip_tos–ipv4_tos
                    & tp_src –tp_src
                    & tp_dst–tp_dst
                    & X"00";
                compose_done <= '1';
                int_req_nxt := '1';
                req_latch_state_nxt <= RL_WAIT_REQ;
            END IF;
        ELSE
            req_latch_state_nxt <= RL_WAIT_PARSE_DONE;
        END IF;
    WHEN RL_WAIT_REQ =>
        IF (lu_req = '1' and lu_req_prev = '0') THEN
            int_req_nxt := '0';
            req_latch_state_nxt <= RL_WAIT_PARSE_DONE;
        ELSE
            req_latch_state_nxt <= RL_WAIT_REQ;
        END IF;
END CASE;
END PROCESS;
..... Flow_table module Interface Process .....
```

```
PROCESS (aresetn, asclk, lookup_inf_state, lu_ack, int_req)
BEGIN
    IF (aresetn = '1') THEN
        lu_req <= '0';
        lu_entry <= (others =>'0');
        lookup_inf_state <= LU_WAIT_INT_REQ;
    ELSIF (asclk'event and asclk = '1') THEN
        lookup_inf_state <= lookup_inf_state_nxt;
    END IF;
    CASE lookup_inf_state is
        WHEN LU_WAIT_INT_REQ =>
            IF (int_req = '1') THEN
                lu_req <= '1';
                lu_entry <= int_entry;
                lookup_inf_state_nxt <= LU_WAIT_ACK;
            ELSE
                lookup_inf_state_nxt <= LU_WAIT_INT_REQ;
            END IF;
        WHEN LU_WAIT_ACK =>
            IF (lu_ack = '1') THEN
                lu_req <= '0';
                lookup_inf_state_nxt <= LU_WAIT_INT_REQ;
            ELSE
                lookup_inf_state_nxt <= LU_WAIT_ACK;
            END IF;
    END CASE;
END PROCESS;
END lu_entry_composer;
```

Appendix E

Flow Table Controller Top Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY flow_table_controller IS
GENERIC ( OPENFLOW_MATCH_SIZE: INTEGER:= 256;
OPENFLOW_MASK_SIZE : INTEGER:= 256;
OPENFLOW_ACTION_SIZE :INTEGER:= 256
);
PORT ( asclk : IN STD_LOGIC;
    asresetn: IN STD_LOGIC;
    lu_req1 : IN STD_LOGIC;
    lu_req2 : IN STD_LOGIC;
    lu_req3 : IN STD_LOGIC;
    lu_req4 : IN STD_LOGIC;
    lu_entry1 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry2 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry3 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry4 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_done1 : INOUT STD_LOGIC;
    lu_done2 : INOUT STD_LOGIC;
    lu_done3 : INOUT STD_LOGIC;
    lu_done4 : INOUT STD_LOGIC;
    lu_ack1 : OUT STD_LOGIC;
    lu_ack2 : OUT STD_LOGIC;
    lu_ack3 : OUT STD_LOGIC;
    lu_ack4 : OUT STD_LOGIC;
```

```
    action: OUT STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    match : OUT STD_LOGIC_VECTOR (3 downto 0)      );
END flow_table_controller;


ARCHITECTURE flow_table_controller of flow_table_controller IS
SIGNAL add_entry_int, no_match_entry_int:std_logic_vector(OPENFLOW_MATCH_SIZE-
1 DOWNTO 0);
SIGNAL add_mask_int: std_logic_vector(OPENFLOW_MASK_SIZE-1 DOWNTO
0);
SIGNAL policy_req_int, add_entry_reply_int, add_entry_done_int :std_logic;
SIGNAL action_in_int :std_logic_vector(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);


.....Flow Table Lookup.....
COMPONENT ft_lookup
PORT(
    asclk : IN STD_LOGIC;
    asresetn : IN STD_LOGIC;
    lu_req1 : IN STD_LOGIC;
    lu_req2 : IN STD_LOGIC;
    lu_req3 : IN STD_LOGIC;
    lu_req4 : IN STD_LOGIC;
    lu_entry1 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry2 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry3 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry4 : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    add_entry : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    add_mask : IN STD_LOGIC_VECTOR(OPENFLOW_MASK_SIZE-1 DOWNTO
0);
    add_entry_reply : IN STD_LOGIC;
    lu_done1 : INOUT STD_LOGIC;
    lu_done2 : INOUT STD_LOGIC;
    lu_done3 : INOUT STD_LOGIC;
    lu_done4 : INOUT STD_LOGIC;
```

```
    lu_ack1 : OUT STD_LOGIC;
    lu_ack2 : OUT STD_LOGIC;
    lu_ack3 : OUT STD_LOGIC;
    lu_ack4 : OUT STD_LOGIC;
    policy_req : OUT STD_LOGIC;
    action_in : IN STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    action_out: OUT STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1
DOWNTO 0);
    no_match_entry : OUT STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-
1 DOWNTO 0);
    add_entry_done : OUT STD_LOGIC;
    match : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END COMPONENT;
.............................Controller Policy.............................
COMPONENT policy
PORT(
    asclk : IN STD_LOGIC;
    asresetn : IN STD_LOGIC;
    policy_req : IN STD_LOGIC;
    no_match_entry : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
    add_entry_done : IN STD_LOGIC;
    add_entry : OUT STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
    add_mask: OUT STD_LOGIC_VECTOR(OPENFLOW_MASK_SIZE-1 DOWNTO
0);
    action:out STD_LOGIC_VECTOR (OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    add_entry_reply : OUT STD_LOGIC      );
END COMPONENT;
BEGIN
Inst_ft_lookup: ft_lookup PORT MAP(
    asclk => asclk,
    asresetn => asresetn,
    lu_req1 => lu_req1,
    lu_req2 => lu_req2,
    lu_req3 => lu_req3,
    lu_req4 => lu_req4,
    lu_entry1 => lu_entry1,
```

```
    lu_entry2 => lu_entry2,
    lu_entry3 => lu_entry3,
    lu_entry4 => lu_entry4,
    add_entry => add_entry_int,
    add_mask => add_mask_int,
    lu_done1 => lu_done1,
    lu_done2 => lu_done2,
    lu_done3 => lu_done3,
    lu_done4 => lu_done4,
    lu_ack1 => lu_ack1,
    lu_ack2 => lu_ack2,
    lu_ack3 => lu_ack3,
    lu_ack4 => lu_ack4,
    policy_req => policy_req_int,
    action_in => action_in_int,
    action_out=> action,
    no_match_entry => no_match_entry_int,
    add_entry_reply => add_entry_reply_int,
    add_entry_done => add_entry_done_int,
    match => match
    );
Inst_policy: policy PORT MAP(
    asclk => asclk,
    asresetn => asresetn,
    policy_req => policy_req_int,
    no_match_entry => no_match_entry_int,
    add_entry => add_entry_int,
    add_mask => add_mask_int,
    action => action_in_int,
    add_entry_done => add_entry_done_int,
    add_entry_reply => add_entry_reply_int
    );
END flow_table_controller;
```

Appendix F
Flow Table Lookup Block

```
library IEEE; use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_unsigned.ALL;


ENTITY ft_lookup IS
GENERIC ( OPENFLOW_MATCH_SIZE: INTEGER:= 256;
    OPENFLOW_MASK_SIZE : INTEGER:= 256;
    OPENFLOW_ACTION_SIZE: INTEGER:= 256
    );
PORT ( asclk : IN STD_LOGIC;
    asresetn : IN STD_LOGIC;
    lu_req1 : IN STD_LOGIC;
    lu_req2 : IN STD_LOGIC;
    lu_req3 : IN STD_LOGIC;
    lu_req4 : IN STD_LOGIC;
    lu_entry1 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry2 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry3 : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    lu_entry4: IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    add_entry : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
DOWNTO 0);
    add_mask : IN STD_LOGIC_VECTOR (OPENFLOW_MASK_SIZE-1 DOWNTO
0);
    lu_done1 : INOUT STD_LOGIC;
    lu_done2 : INOUT STD_LOGIC;
    lu_done3 : INOUT STD_LOGIC;
```

```
    lu_done4 : INOUT STD_LOGIC;
    lu_ack1 : OUT STD_LOGIC;
    lu_ack2 : OUT STD_LOGIC;
    lu_ack3 : OUT STD_LOGIC;
    lu_ack4 : OUT STD_LOGIC;
    policy_req : OUT STD_LOGIC;
    action_in: IN STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    action_out: OUT STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1
DOWNTO 0);
    no_match_entry: OUT STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-
1 DOWNTO 0);
    add_entry_reply : IN STD_LOGIC;
    add_entry_done: OUT STD_LOGIC;
    match : OUT STD_LOGIC_VECTOR (3 DOWNTO 0)
    );
END ft_lookup;


ARCHITECTURE ft_lookup of ft_lookup is
.....Exact Match Table.....
COMPONENT exact_match1_exdes
PORT(
    RSTA : IN STD_LOGIC;
    WEA : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    ADDRA : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    DINA : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    CLKA : IN STD_LOGIC;
    RSTB : IN STD_LOGIC;
    WEB : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    ADDRB : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    DINB : IN STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    CLKB : IN STD_LOGIC;
    DOUTA : OUT STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
    DOUTB : OUT STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0)
    );
END COMPONENT;
.....Wildcard Match Table.....
```

```
COMPONENT wildcard_match1_exdes
PORT(
    RSTA : IN STD_LOGIC;
    WEA : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    ADDRA : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    DINA : IN STD_LOGIC_VECTOR(OPENFLOW_MASK_SIZE-1 DOWNTO
0);
    CLKA : IN STD_LOGIC;
    RSTB : IN STD_LOGIC;
    WEB : IN STD_LOGIC_VECTOR(0 to 0);
    ADDRB : IN STD_LOGIC_VECTOR(9 downto 0);
    DINB : IN STD_LOGIC_VECTOR(OPENFLOW_MASK_SIZE-1 DOWNTO
0);
    CLKB : IN STD_LOGIC;
    DOUTA : OUT STD_LOGIC_VECTOR(OPENFLOW_MASK_SIZE-1 DOWNTO
0);
    DOUTB : OUT STD_LOGIC_VECTOR(OPENFLOW_MASK_SIZE-1 DOWNTO
0)
    );
END COMPONENT;
.....Action Storage.....
COMPONENT action
PORT (
    clka : IN STD_LOGIC;
    rsta : IN STD_LOGIC;
    wea : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addra : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    dina : IN STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    douta : OUT STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    clkb : IN STD_LOGIC;
    rstb : IN STD_LOGIC;
    web : IN STD_LOGIC_VECTOR(0 DOWNTO 0);
    addrb : IN STD_LOGIC_VECTOR(9 DOWNTO 0);
    dinb : IN STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    doutb : OUT STD_LOGIC_VECTOR(OPENFLOW_ACTION_SIZE-1 DOWNTO
0)
    );
END COMPONENT;
```

```
TYPE request_type IS (req_idle, req1, req2,req3,req4,req5);
TYPE flow_table_lookup_type is (flow_table_lookup_wait, write_entry_start,write_entry_done,
lu_entry_match_start,lu_entry_match_done, lu_entry_match_done_nxt);
SIGNAL flow_table_lookup_state, flow_table_lookup_state_nxt : flow_table_lookup_type;
SIGNAL request_state, request_state_nxt:request_type;
SIGNAL lu_entry: STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO
0);
SIGNAL exact_match_dina ,exact_match_dinb,exact_match_doutb,exact_match_douta:
STD_LOGIC_VECTOR(OPENFLOW_MATCH_SIZE-1 DOWNTO 0);
SIGNAL action_dina, action_dinb, action_doutb, action_douta:STD_LOGIC_VECTOR(OPENFLO
1 DOWNTO 0);
signal wildcard_match_dina, wildcard_match_douta,wildcard_match_doutb: STD_LOGIC_VECT
1 DOWNTO 0);
SIGNAL exact_match_wea,wildcard_match_wea, action_wea: STD_LOGIC_VECTOR
(0 DOWNTO 0);
SIGNAL exact_match_addrb,exact_match_addra,wildcard_match_addra,wildcard_match_addrb,a
STD_LOGIC_VECTOR(9 DOWNTO 0);
SIGNAL flow_entry_req, controller_req:STD_LOGIC;
SIGNAL lu_done1_int,lu_done2_int,lu_done3_int,lu_done4_int:STD_LOGIC;
SIGNAL req_num:STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN


.....Request Selection (Round Robin).....
PROCESS(asclk, asresetn, lu_done1_int, lu_done2_int, lu_done3_int, lu_done4_int,
lu_req1, lu_req2, lu_req3, lu_req4, add_entry_reply, request_state)
BEGIN
    IF (asresetn = '1' ) THEN
        flow_entry_req <= '0';
        controller_req <= '0';
        lu_ack1 <= '0';
        lu_ack2 <= '0';
        lu_ack3 <= '0';
        lu_ack4 <= '0';
        lu_entry <= (others =>'0');
        req_num <= (others =>'0');
        request_state <= req_idle;
    ELSIF (asclk'event and asclk = '1') THEN
        request_state <= request_state_nxt;
    END IF;
```

```
CASE request_state IS
    WHEN req_idle =>
        IF (lu_req1 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack1 <= '1';
            lu_entry <= lu_entry1;
            req_num <= B"0001";
            request_state_nxt <= req1;
        ELSIF (lu_req2 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack2 <= '1';
            lu_entry <= lu_entry2;
            req_num <= B"0010";
            request_state_nxt <= req2;
        ELSIF (lu_req3 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack3 <= '1';
            lu_entry <= lu_entry3;
            req_num <= B"0100";
            request_state_nxt <= req3;
        ELSIF (lu_req4 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack4 <= '1';
            lu_entry <= lu_entry4;
            req_num <= B"1000";
            request_state_nxt <= req4;
        ELSIF (add_entry_reply = '1') THEN
            controller_req <= '1';
            flow_entry_req <= '0';
            req_num <= B"0000";
            request_state_nxt <= req5;
        ELSE
            request_state_nxt <= req_idle;
        END IF;
    WHEN req1 =>
        IF (lu_req2 = '1' and lu_done1_int = '1') THEN
            flow_entry_req <= '1';
```

```
            controller_req <= '0';
            lu_ack2 <= '1';
            lu_entry <= lu_entry2;
            req_num <= B"0010";
            request_state_nxt <= req2;
        ELSIF (lu_req3 = '1' and lu_done1_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack3 <= '1';
            lu_entry <= lu_entry3;
            req_num <= B"0100";
            request_state_nxt <= req3;
        ELSIF (lu_req4 = '1' and lu_done1_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack4 <= '1';
            lu_entry <= lu_entry4;
            req_num <= B"1000";
            request_state_nxt <= req4;
        ELSIF (add_entry_reply = '1' and lu_done1_int = '1') THEN
            controller_req <= '1';
            flow_entry_req <= '0';
            req_num <= B"0000";
            request_state_nxt <= req5;
        ELSIF (lu_req1 = '1' and lu_done1_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack1 <= '1';
            lu_entry <= lu_entry1;
            req_num <= B"0001";
            request_state_nxt <= req1;
        ELSE
            request_state_nxt <= req_idle;
        END IF;
    WHEN req2 =>
        IF (lu_req3 = '1' and lu_done2_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack3 <= '1';
            lu_entry <= lu_entry3;
            req_num <= B"0100";
```

```
            request_state_nxt <= req3;
        ELSIF (lu_req4 = '1' and lu_done2_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack4 <= '1';
            lu_entry <= lu_entry4;
            req_num <= B"1000";
            request_state_nxt <= req4;
        ELSIF (add_entry_reply = '1' and lu_done2_int = '1') THEN
            controller_req <= '1';
            flow_entry_req <= '0';
            req_num <= B"0000";
            request_state_nxt <= req5;
        ELSIF (lu_req1 = '1' and lu_done2_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack1 <= '1';
            lu_entry <= lu_entry1;
            req_num <= B"0001";
            request_state_nxt <= req1;
        ELSIF (lu_req2 = '1' and lu_done2_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack2 <= '1';
            lu_entry <= lu_entry2;
            req_num <= B"0010";
            request_state_nxt <= req2;
        ELSE
            request_state_nxt <= req_idle;
        END IF;
    WHEN req3 =>
        IF (lu_req4 = '1' and lu_done3_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack4 <= '1';
            lu_entry <= lu_entry4;
            req_num <= B"1000";
            request_state_nxt <= req4;
        ELSIF (add_entry_reply = '1' and lu_done3_int = '1') THEN
            controller_req <= '1';
            flow_entry_req <= '0';
```

```
            req_num <= B"0000";
            request_state_nxt <= req5;
        ELSIF (lu_req1 = '1' and lu_done3_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack1 <= '1';
            lu_entry <= lu_entry1;
            req_num <= B"0001";
            request_state_nxt <= req1;
        ELSIF (lu_req2 = '1' and lu_done3_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack2 <= '1';
            lu_entry <= lu_entry2;
            req_num <= B"0010";
            request_state_nxt <= req2;
        ELSIF (lu_req3 = '1' and lu_done3_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack3 <= '1';
            lu_entry <= lu_entry3;
            req_num <= B"0100";
            request_state_nxt <= req3;
        ELSE
        request_state_nxt <= req_idle;
        END IF;
    WHEN req4 =>
        IF (add_entry_reply = '1' and lu_done4_int = '1') THEN
            controller_req <= '1';
            flow_entry_req <= '0';
            req_num <= B"0000";
            request_state_nxt <= req5;
        ELSIF (lu_req1 = '1' and lu_done4_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack1 <= '1';
            lu_entry <= lu_entry1;
            req_num <= B"0001";
            request_state_nxt <= req1;
        ELSIF (lu_req2 = '1' and lu_done4_int = '1') THEN
            flow_entry_req <= '1';
```

```
            controller_req <= '0';
            lu_ack2 <= '1';
            lu_entry <= lu_entry2;
            req_num <= B"0010";
            request_state_nxt <= req2;
        ELSIF (lu_req3 = '1' and lu_done4_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack3 <= '1';
            lu_entry <= lu_entry3;
            req_num <= B"0100";
            request_state_nxt <= req3;
        ELSIF (lu_req4 = '1' and lu_done4_int = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack4 <= '1';
            lu_entry <= lu_entry4;
            req_num <= B"1000";
            request_state_nxt <= req4;
        ELSE
            request_state_nxt <= req_idle;
        END IF;
    WHEN req5 =>
        IF (lu_req1 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack1 <= '1';
            lu_entry <= lu_entry1;
            req_num <= B"0001";
            request_state_nxt <= req1;
        ELSIF (lu_req2 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack2 <= '1';
            lu_entry <= lu_entry2;
            req_num <= B"0010";
            request_state_nxt <= req2;
        ELSIF (lu_req3 = '1') THEN
            flow_entry_req <= '1';
            controller_req <= '0';
            lu_ack3 <= '1';
```

```
                lu_entry <= lu_entry3;
                req_num <= B"0100";
                request_state_nxt <= req3;
            ELSIF (lu_req4 = '1') THEN
                flow_entry_req <= '1';
                controller_req <= '0';
                lu_ack4 <= '1';
                lu_entry <= lu_entry4;
                req_num <= B"1000";
                request_state_nxt <= req4;
            ELSIF (add_entry_reply = '1') THEN
                controller_req <= '1';
                flow_entry_req <= '0';
                req_num <= B"0000";
                request_state_nxt <= req5;
            ELSE
                request_state_nxt <= req_idle;
            END IF;
    END CASE;
END PROCESS;


.....Write Flow Entry Process.....
PROCESS (asclk,asresetn,controller_req,flow_entry_req,flow_table_lookup_state )
VARIABLE exact_match_addra_nxt,wildcard_match_addra_nxt,action_addra_nxt:
STD_LOGIC_VECTOR(9 DOWNTO 0);
VARIABLE exact_match_nxt,wildcard_match_nxt: STD_LOGIC;
VARIABLE lu_entry_nxt, exact_match_douta_nxt,exact_match_doutb_nxt: STD_LOGIC_VEC
(OPENFLOW_MATCH_SIZE-1 DOWNTO 0);
VARIABLE wildcard_match_douta_nxt,wildcard_match_doutb_nxt: STD_LOGIC_VECTOR(OF
1 DOWNTO 0);
VARIABLE req_num_nxt : STD_LOGIC_VECTOR(3 DOWNTO 0);
BEGIN
    IF (asresetn = '1') THEN
.....Write signals.....
        exact_match_wea <= (others =>'0');
        wildcard_match_wea <= (others =>'0');
        action_wea <= (others =>'0');
        add_entry_done <= '0';
        exact_match_addra <= (others =>'0');
        wildcard_match_addra <= (others =>'0');
        action_addra <= (others =>'0');
```

```
        exact_match_dina <= (others =>‘0’);
        wildcard_match_dina <= (others =>‘0’);
        action_dina <= (others =>‘0’);
        exact_match_addra_nxt := (others =>‘0’);
        wildcard_match_addra_nxt := (others =>‘0’);
        action_addra_nxt := (others =>‘0’);
.....Read signals.....
        exact_match_addrb <= (others =>‘0’);
        exact_match_doutb <= (others =>‘0’);
        wildcard_match_addrb <= (others =>‘0’);
        wildcard_match_doutb <= (others =>‘0’);
        action_addrb <= (others =>‘0’);
        action_doutb <= (others =>‘0’);
        lu_done1 <= ‘0’;
        lu_done2 <= ‘0’;
        lu_done3 <= ‘0’;
        lu_done4 <= ‘0’;
        match <= (others =>‘0’);
        policy_req <= ‘0’;
        no_match_entry <= (others =>‘0’);
        action_out <= (others =>‘0’);
        flow_table_lookup_state <= flow_table_lookup_wait;
    ELSIF (asclk’event and asclk = ‘1’) THEN
        flow_table_lookup_state <= flow_table_lookup_state_nxt;
    END IF;
    CASE flow_table_lookup_state IS
        WHEN flow_table_lookup_wait =>
            IF (controller_req = ‘1’) THEN
                req_num_nxt := req_num;
                add_entry_done <= ‘0’;
                flow_table_lookup_state_nxt <= write_entry_start;
            ELSIF (flow_entry_req = ‘1’) THEN
                req_num_nxt := req_num;
                flow_table_lookup_state_nxt <= lu_entry_match_start;
            ELSE
                flow_table_lookup_state_nxt <=flow_table_lookup_wait;
            END IF;
.....Write Flow Entry Process.....
        WHEN write_entry_start =>
            exact_match_wea <= B"1";
            wildcard_match_wea <= B"1";
```

```
            action_wea <= B"1";
            exact_match_dina <= add_entry;
            wildcard_match_dina <= add_mask;
            action_dina <= action_in;
            wildcard_match_addra <= wildcard_match_addra_nxt;
            wildcard_match_addra_nxt := wildcard_match_addra_nxt + "1";
            exact_match_addra <= exact_match_addra_nxt;
            exact_match_addra_nxt := exact_match_addra_nxt + "1";
            action_addra <= action_addra_nxt;
            action_addra_nxt := action_addra_nxt + "1";
            IF (exact_match_addra_nxt = B"1111111111" or exact_match_addra =
B"1111111111") THEN
                exact_match_addra_nxt := (others =>'0');
                exact_match_addra <= (others =>'0');
            END IF;
            IF (wildcard_match_addra_nxt = B"1111111111" or wildcard_match_addra
= B"1111111111") THEN
                wildcard_match_addra_nxt := (others =>'0');
                wildcard_match_addra <= (others =>'0');
            END IF;
            IF (action_addra_nxt = B"1111111111" or action_addra = B"1111111111")
THEN
                action_addra_nxt := (others =>'0');
                action_addra <= (others =>'0');
            END IF;
                flow_table_lookup_state_nxt <= write_entry_done;
        WHEN write_entry_done =>
            add_entry_done <= '1';
            action_out <= action_in;
            IF (flow_entry_req = '1') THEN
                req_num_nxt := req_num;
                flow_table_lookup_state_nxt <=lu_entry_match_start;
            ELSIF (controller_req = '1') THEN
                req_num_nxt := req_num;
                flow_table_lookup_state_nxt <= write_entry_start;
            ELSE
                flow_table_lookup_state_nxt <=flow_table_lookup_wait;
            END IF;
..... Flow Table Lookup Process .....
        WHEN lu_entry_match_start =>
            exact_match_wea <= B"0";
```

```
            wildcard_match_wea <= B"0";
            lu_entry_nxt := lu_entry;
            FOR i IN 0 TO 1023 LOOP
                exact_match_douta_nxt:= exact_match_douta;
                IF (lu_entry_nxt = exact_match_douta_nxt) THEN
                    exact_match_nxt:= '1';
                ELSE
                    exact_match_nxt:= '0';
                END IF;
                exact_match_addrb<= exact_match_addrb + "1";
            END LOOP ;
            FOR i IN 0 TO 1023 LOOP
                wildcard_match_douta_nxt:= wildcard_match_douta;
                IF(lu_entry_nxt(255 DOWNTO 152) = wildcard_match_douta_nxt(255
DOWNTO 152)) THEN
                    wildcard_match_nxt:= '1';
            EXIT;
                ELSE
                    wildcard_match_nxt:= '0';
                END IF;
                    wildcard_match_addrb<= wildcard_match_addrb + "1";
            END LOOP ;
            flow_table_lookup_state_nxt <= lu_entry_match_done;
        WHEN lu_entry_match_done =>
            IF(req_num_nxt = B"0001") THEN
                IF (exact_match_nxt='1' and wildcard_match_nxt = '1') THEN
                    match <= B"0001";
                    policy_req <= '0';
                ELSIF (exact_match_nxt='0' and wildcard_match_nxt = '1') THEN
                    match <= B"0001";
                    policy_req <= '0';
                ELSIF (exact_match_nxt='0' and wildcard_match_nxt = '0') THEN
                    match<= B"0000";
                    policy_req <= '1';
                    no_match_entry <= lu_entry_nxt;
                END IF;
            lu_done1_int <= '1';
            lu_done1 <= '1';
            ELSIF (req_num_nxt = B"0010") THEN
                IF (exact_match_nxt='1' and wildcard_match_nxt = '1') THEN
                    match <= B"0010";
```

```
                policy_req <= '0';
            ELSIF (exact_match_nxt= '0' and wildcard_match_nxt = '1') THEN
                match <= B"0010";
                policy_req <= '0';
            ELSIF (exact_match_nxt= '0' and wildcard_match_nxt = '0') THEN
                match <= B"0000";
                policy_req <= '1';
                no_match_entry <= lu_entry_nxt;
            END IF;
                lu_done2_int <= '1';
                lu_done2 <= '1';
        ELSIF (req_num_nxt = B"0100") THEN
            IF (exact_match_nxt= '1' and wildcard_match_nxt = '1') THEN
                match <= B"0100";
                policy_req <= '0';
            ELSIF (exact_match_nxt='0' and wildcard_match_nxt = '1') THEN
                match <= B"0100";
                policy_req <= '0';
            ELSIF (exact_match_nxt= '0' and wildcard_match_nxt = '0') THEN
                match <= B"0000";
                policy_req <= '1';
                no_match_entry <= lu_entry_nxt;
            END IF;
            lu_done3_int <= '1';
            lu_done3 <= '1';
        ELSIF (req_num_nxt = B"1000") THEN
            IF (exact_match_nxt= '1' and wildcard_match_nxt = '1') THEN
                match <= B"1000";
                policy_req <= '0';
            ELSIF (exact_match_nxt='0' and wildcard_match_nxt = '1') THEN
                match <= B"1000";
                policy_req <= '0';
            ELSIF (exact_match_nxt= '0' and wildcard_match_nxt = '0') THEN
                match <= B"0000";
                policy_req <= '1';
                no_match_entry <= lu_entry_nxt;
            END IF;
            lu_done4_int <= '1';
            lu_done4 <= '1';
        END IF;
    flow_table_lookup_state_nxt <= lu_entry_match_done_nxt;
```

```
        WHEN lu_entry_match_done_nxt=>
            IF (controller_req = '1') THEN
                req_num_nxt := req_num;
                flow_table_lookup_state_nxt <= write_entry_start;
            ELSIF (flow_entry_req = '1') THEN
                req_num_nxt := req_num;
                flow_table_lookup_state_nxt <=lu_entry_match_start;
            ELSE
                flow_table_lookup_state_nxt <=flow_table_lookup_wait;
            END IF;
    END CASE;
END PROCESS;
.....Exact Match Table.....
Inst_exact_match1_exdes: exact_match1_exdes PORT MAP(
    RSTA => asresetn,
    WEA => exact_match_wea,
    ADDRA => exact_match_addra,
    DINA => exact_match_dina,
    DOUTA => exact_match_douta,
    CLKA => asclk,
    RSTB => asresetn,
    WEB => B"0",
    ADDRB => exact_match_addrb,
    DINB => X"00000000000000000000000000000000000000000000000000000000000000000000",
    DOUTB => exact_match_doutb,
    CLKB => asclk
    );
.....Wildcard Match Table.....
Inst_wildcard_match1_exdes: wildcard_match1_exdes PORT MAP(
    RSTA => asresetn,
    WEA => wildcard_match_wea,
    ADDRA => wildcard_match_addra,
    DINA => wildcard_match_dina,
    DOUTA => wildcard_match_douta,
    CLKA => asclk,
    RSTB => asresetn,
    WEB => B"0",
    ADDRB => wildcard_match_addrb,
    DINB => X"00000000000000000000000000000000000000000000000000000000000000000000",
    DOUTB => wildcard_match_doutb,
    CLKB => asclk
```

```
);
.....Action Storage.....
Inst_action : action PORT MAP (
    clka => asclk,
    rsta => asresetn,
    wea => action_wea,
    addra => action_addra,
    dina => action_dina,
    douta => action_douta,
    clkb => asclk,
    rstb => asresetn,
    web => B"0",
    addrb => action_addrb,
    dinb => X"00000000000000000000000000000000000000000000000000000000000000000000000000000000",
    doutb => action_doutb
    );
END ft_lookup;
```

Appendix G
Controller Policy Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
ENTITY policy IS
GENERIC ( OPENFLOW_MATCH_SIZE: integer:= 256;
    OPENFLOW_MASK_SIZE: integer:=256;
    OPENFLOW_ACTION_SIZE: integer:=256
    );
Port (
    asclk: IN STD_LOGIC;
    asresetn: IN STD_LOGIC;
    policy_req : IN STD_LOGIC;
    no_match_entry : IN STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-
1 downto 0);
    add_entry: OUT STD_LOGIC_VECTOR (OPENFLOW_MATCH_SIZE-1
downto 0);
    add_mask: OUT STD_LOGIC_VECTOR (OPENFLOW_MASK_SIZE-1 downto
0);
    action: OUT STD_LOGIC_VECTOR (OPENFLOW_ACTION_SIZE-1 downto
0);
    add_entry_done : IN STD_LOGIC;
    add_entry_reply : OUT STD_LOGIC);
END policy;


ARCHITECTURE policy of policy IS
TYPE policy_type IS (policy_wait, policy_add, policy_done);
SIGNAL policy_state, policy_state_nxt: policy_type;
BEGIN
PROCESS (asclk,asresetn, policy_req, add_entry_done,policy_state )
BEGIN
```

```
    IF (asresetn = '1') THEN
        add_entry_reply <='0';
        add_entry(OPENFLOW_MATCH_SIZE-1 downto 0) <= (others =>'0');
        add_mask (OPENFLOW_MASK_SIZE-1 downto 0)<=(others =>'0');
        action <=(others =>'0');
        policy_state <= policy_wait;
    ELSIF (asclk'event and asclk = '1') THEN
        policy_state <= policy_state_nxt;
    END IF;
    CASE policy_state IS
        WHEN policy_wait =>
            IF (policy_req = '1') THEN
                policy_state_nxt <= policy_add;
            ELSE
                policy_state_nxt <= policy_wait;
            END IF;
        WHEN policy_add =>
            action(224) <= '1';
            add_entry_reply <= '1';
            add_mask(255 DOWNTO 152) <= no_match_entry(255 downto 152);
            policy_state_nxt <= policy_done;
        WHEN policy_done =>
            IF (add_entry_done = '1') THEN
                add_entry_reply <= '0';
                policy_state_nxt <= policy_wait;
            ELSE
                policy_state_nxt <= policy_done;
            END IF;
        END CASE;
END PROCESS;
END policy;
```

Appendix H
Packet Forwarding Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;


ENTITY packet_forwarding IS
GENERIC (
    OPENFLOW_ACTION_SIZE: INTEGER :=256
    );
PORT ( asclk :IN STD_LOGIC;
    asresetn: IN STD_LOGIC;
    lu_done1: IN STD_LOGIC;
    lu_done2: IN STD_LOGIC;
    lu_done3: IN STD_LOGIC;
    lu_done4: IN STD_LOGIC;
    match: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
    output_buffer_empty1: IN STD_LOGIC;
    output_buffer_empty2: IN STD_LOGIC;
    output_buffer_empty3: IN STD_LOGIC;
    output_buffer_empty4: IN STD_LOGIC;
    rd_en1: OUT STD_LOGIC;
    rd_en2: OUT STD_LOGIC;
    rd_en3: OUT STD_LOGIC;
    rd_en4: OUT STD_LOGIC;
    action:IN STD_LOGIC_VECTOR (OPENFLOW_ACTION_SIZE-1 DOWNTO
0);
    packet_in1: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_in2: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_in3: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_in4: IN STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_out1 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
```

```
    packet_out2 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_out3 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0);
    packet_out4 : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
END packet_forwarding;


ARCHITECTURE packet_forwarding of packet_forwarding IS
TYPE forwarding_type IS (forwarding_start, forwarding_1, forwarding_2, forward-
ing_3, forwarding_4);
SIGNAL forwarding_state, forwarding_state_nxt: forwarding_type;


BEGIN
PROCESS (asclk,asresetn, match, lu_done1,lu_done2, lu_done3, lu_done4, out-
put_buffer_empty1, output_buffer_empty2, output_buffe_empty3,output_buffer_empty4)
BEGIN
    IF (asresetn = '1') THEN
        packet_out1 <= (others =>'0');
        packet_out2 <= (others =>'0');
        packet_out3 <= (others =>'0');
        packet_out4 <= (others =>'0');
        rd_en1 <= '0';
        rd_en2 <= '0';
        rd_en3 <= '0';
        rd_en4 <= '0';
        forwarding_state <= forwarding_start;
    ELSIF (asclk'event and asclk = '1') THEN
        forwarding_state <= forwarding_state_nxt;
    END IF;
    CASE forwarding_state IS
        WHEN forwarding_start =>
            IF (output_buffer_empty1 ='0' and lu_done1 = '1' and match = B"0001")
THEN
                rd_en1 <= '1';
                packet_out2 <= packet_in1;
                forwarding_state_nxt <= forwarding_1;
            ELSIF (output_buffer_empty2 ='0' and lu_done2 = '1' and match =
B"0010") THEN
                rd_en2 <= '1';
                packet_out3 <= packet_in2;
                forwarding_state_nxt <= forwarding_2;
```

```
            ELSIF (output_buffer_empty3 ='0' and lu_done3 = '1' and match =
B"0100") THEN
                rd_en3 <= '1';
                packet_out4 <= packet_in3;
                forwarding_state_nxt <= forwarding_3;
            ELSIF (output_buffer_empty4 ='0' and lu_done4 = '1' and match =
B"1000") THEN
                rd_en4 <= '1';
                packet_out1 <= packet_in4;
                forwarding_state_nxt <= forwarding_4;
            ELSIF ( action(224)= '1' and lu_done1 = '1' and match =B"0000") THEN
                rd_en1 <= '1';
                packet_out2 <= packet_in1;
                forwarding_state_nxt <= forwarding_1;
            ELSIF (action(224)= '1' and lu_done2 = '1' and match =B"0000") THEN
                rd_en2 <= '1';
                packet_out3 <= packet_in2;
                forwarding_state_nxt <= forwarding_2;
            ELSIF (action(224)= '1' and lu_done3 = '1' and match =B"0000") THEN
                rd_en3 <= '1';
                packet_out4 <= packet_in3;
                forwarding_state_nxt <= forwarding_3;
            ELSIF (action(224)= '1' and lu_done4 = '1' and match =B"0000") THEN
                rd_en4 <= '1';
                packet_out1 <= packet_in4;
                forwarding_state_nxt <= forwarding_4;
            ELSE
                forwarding_state_nxt <= forwarding_start;
        END IF;
        WHEN forwarding_1 =>
            IF (output_buffer_empty1 = '0') THEN
                rd_en1 <= '1';
                packet_out2 <= packet_in1;
                forwarding_state_nxt <= forwarding_1;
            ELSE
                rd_en1 <= '0';
                forwarding_state_nxt <= forwarding_start;
            END IF;
        WHEN forwarding_2 =>
            IF (output_buffer_empty2 = '0') THEN
                rd_en2 <= '1';
```

```
            packet_out3 <= packet_in2;
            forwarding_state_nxt <= forwarding_2;
        ELSE
            rd_en2 <= '0';
            forwarding_state_nxt <= forwarding_start;
        END IF;
    WHEN forwarding_3 =>
        IF (output_buffer_empty3 = '0') then
            packet_out4 <= packet_in3;
            rd_en3 <= '1';
            forwarding_state_nxt <= forwarding_3;
        ELSE
            rd_en3 <= '0';
            forwarding_state_nxt <= forwarding_start;
        END IF;
    WHEN forwarding_4 =>
        IF (output_buffer_empty4 = '0') THEN
            rd_en4 <= '1';
            packet_out1 <= packet_in4;
            forwarding_state_nxt <= forwarding_4;
        ELSE
            rd_en4 <= '0';
            forwarding_state_nxt <= forwarding_start;
        END IF;
    END CASE;
END PROCESS;
END packet_forwarding;
```