分类号：TP311.5　　单位代码：10335

密　级：（无）　　学　　号：21251160

浙江大学

硕士学位论文

中文论文题目：**软硬件可编程 OpenFlow 交换机的研究与实现**

英文论文题目：**The Research and Implementation of All Programmable OpenFlow Switch**

申请人姓名：潘祖龙

指导教师：施青松 副教授

合作导师：

专业学位类别：工程硕士

专业学位领域：软件工程

所在学院：软件学院

论文提交日期 2014 年 4 月 22

软硬件可编程 OpenFLow 交换机的研究与实现

论文作者签名:_____

指导教师签名:_____

论文评阅人 1：_____

评阅人 2：_____

评阅人 3：_____

评阅人 4：_____

评阅人 5：_____

答辩委员会主席：_____

委员 1：_____

委员 2：_____

委员 3：_____

委员 4：_____

委员 5：_____

答辩日期：_____

The Research and Implementation of All Programmable Switch

Author's signature:_____

Supervisor's signature:_____

Thesis reviewer 1：_____

Thesis reviewer 2：_____

Thesis reviewer 3：_____

Thesis reviewer 4：_____

Thesis reviewer 5：_____

Chair：_____
(Committee of oral defence)

Committeeman 1：_____

Committeeman 2：_____

Committeeman 3：_____

Committeeman 4：_____

Committeeman 5：_____

Date of oral defence：_____

浙江大学研究生学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得 **浙江大学** 或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名：　　签字日期：　　年　　月　　日

学位论文版权使用授权书

本学位论文作者完全了解 **浙江大学** 有权保留并向国家有关部门或机构送交本论文的复印件和磁盘，允许论文被查阅和借阅。本人授权 **浙江大学** 可以将学位论文的全部或部分内容编入有关数据库进行检索和传播，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名：　　导师签名：

签字日期：　　年　　月　　日　　签字日期：　　年　　月　　日

摘要

OpenFlow 网络是软件定义网络（Software Define Networking, SDN）的一个子集，它由一个简单的基于流的数据通路交换机和一个远程控制器组成。在实际环境中，OpenFlow 网络常被构建在以太网交换机、路由器或者无线接入点之上。它分离了数据转发平面和控制平面，通过对数据层的抽象和统一，是开发人员能够更加专注于控制层算法的设计和实现。OpenFlow 交换机以流表作为数据转发的核心组件，依靠对数据包头解析和匹配即可支持几乎所有传统协议，为未来网络的发展提供了一种新的决解方案。

本文以 Xilinx 全可编程芯片 Zynq-7000 (以下简称 ZYNQ) 为基础平台，构建了 OpenFlow 交换机系统。该交换机主要由硬件流表和软件软件流表组成，两者之间使用内部 AXI 总线互联，实现高速的总线数据传输。硬件流表在 ZYNQ 可编程逻辑（FPGA）部分构建，一共有两级，使用 OpenFlow1.0 规范实现，第一级流表对 L2 进行匹配，第二级流表对 L3 以上进行匹配，硬件流表的主要功能是实现数据包的线速转发，提高交换机的数据吞吐率；在此基础上，在 ZYNQ ARM 端构建软件流表，它满足 OpenFlow1.3 规范，并可以配置为多级流表，极大的提高了交换机的处理能力，并对硬件流表进行了扩展和补充。在软件端还实现了同控制器的连接通道，数据包和命令通过 OpenFlow 协议在这个通道中进行传输，从而实现了数据层面同控制层面的接口交互。

关键词： OpenFlow，网络虚拟化，软硬件可编程，软件定义网络

Abstract

OpenFlow network, as a subset of software defined networking, is comprised by simple flow-based switches and remote controllers. Under practical conditions, OpenFlow network is often constructed over Ethernet switches, routers and APs. Benefited by separated data plane and control plane, as well as the data plane being unified and abstracted, developers and researchers are capable of concentrating on the control plane algorithm and protocols. Flow tables act as the core components of packet forwarding in an OpenFlow switch. Depending only on packet header matching and manipulation, it can support almost all the existing traditional protocols. This concept is providing a new solution for future networks.

This paper targets on Xilinx All Programmable Zynq-7000 SoC and implements the switch system on this platform. The switch is constituted by hardware and software flow tables and using AXI bus as internal interconnection so as to provide high speed data transfer. Hardware flow table is implemented in FPGA part with two stages in total. It conforms to OpenFlow 1.0 specification. Its first stage matches among L2 fields and the second stage matches among L3 fields and above. Hardware flow table is designed to conduct linear speed forwarding in order to improve the throughput. And beyond hardware, we build software flow tables on Zynq ARM processor, which complies with the OpenFlow 1.3 spec and supports multiple-stage flow table. It prominently enhanced the switch in handling various packets, which is also an extension and replenishment of the hardware part. In addition, datapath to connect with the remote controller is also implemented by software. So that it enables transmission of packets and signaling by OF protocol and thereby establishes an interface between data plane and control plane.

Key Words： OpenFlow, SDN, All Programmable, Virtual netowrk

目录

图目录

表目录

第1章 绪论

1.1 课题背景

如今，我们的生活已经离不开网络，然而，想要对每天都要使用的网络进行革新却变得非常困难。当前的主流网络设备，如以太网交换机、IP 路由器已经变得非常的臃肿，以至于我们很难在对其进行改变或者进行扩展。网络设备厂商和运营商也不会轻易的让研究人员和开发人员在他们的网络中添加一些新功能来对新网络或者新协议进行实验。在这种情况下，OpenFlow 网络在斯坦福大学诞生了，他们的最初目的仅仅是为了帮助校园网络研究人员提供一个真实的实验平台，并期望能够对下一代网络的研究和发展做一点尝试。在这之后，斯坦福大学 Nick Mckeown 教授和他的团队将其推广到了 SDN 层面，这引起了业界和学术界的广泛关注。

在本文之前，已经存在有很多的成熟的 OpenFlow 网络实验平台。比较有名的有 Mininet 虚拟实验环境 [1]以及 NetFPGA 硬件实验平台，这两个平台都由斯坦福大学的研究小组进行维护。当然，还有一些是由网络设备厂商自行维护的， 比如爱立信巴西实验室以及 CPqD 共同维护的 OpenFlow 网络开源项目[2]，本课题在 ARM 端软件交换机以及控制器程序就是对该项目进行了移植。

NetFPGA 是一个基于 Xilinx Virtex 系列芯片的实验平台，一共有两个版本，NetFPGA 1G 和 NetFPGA 10G。NetFPGA 1G 使用的是 Virtex-II Pro 50 芯片，包括 4 个千兆以太网网口，而 NetFPGA 10G 则使用逻辑资源和处理能力相对较强的 Virtex5 芯片，包括 4 个 SFP 万兆以太网网口。这两个版本都是使用 FPGA 逻辑资源来实现的 MAC 软核，并在此基础上建立 OpenFlow 硬件流表，然后通过 PCI/PCIe 总线接口连接到 PC 机中，所有的软件环境都在 PC 机中搭建，本项目在 NetFPGA 之上进行了改进，将交换机的软硬件环境都构建在一块 ZYNQ 芯片之上，其使用更加方便，功能更加丰富[1]。

Xilinx 在 2011 年推出了首款全可编程芯片 ZYNQ，它将双核 Cortex A9 集成到了 FPGA 中，并通过 AMBA 总线在内部将两者进行连接。相对于 NetFPGA 来说，ZYNQ 使用起来就更加方便了，一方面，用户可以在 FPGA 部分对硬件进行

[1] Mininet Website: http://www.mininet.org
[2] http://github.com/CPqd/

编程，并且其逻辑资源也相对丰富；另一方面，用户不需进行复杂总线的设计，就可以通过 ZYNQ 内部自带的高速 AXI 总线就能进行软硬件之间的通信，并能够轻易的在 ARM 端搭建自己的软件环境。基于此，Xilinx 大学计划部（以下简称 XUP）成立了 SDN on ZYNQ 项目组，而我们的工作就是先在 ZYNQ 上搭建一个 OpenFlow 交换机，然后再进行其他方面的扩展。我们的目标很明确，主要有以下几点：

1) 构建一块 SDN Soc 解决方案

2) 降低成本，让 SDN 走进校园，让更多的研究人员参与进来

3) 降低设计迭代周期，快速构建实验平台

在本项目中，本人主要负责基础网络通路搭建、硬件流表结构设计、IP 集成、Linux 驱动设计以及系统集成测试。本项目最大的难点在于硬件流表的设计与实现，FPGA IP 集成以及驱动设计三个方面。本项目从立项到实现，再到测试完成经过 4 个月时间，这其中借助了一些开源项目的成功经验，并立足于 ZYNQ 芯片的全可编程特性，最终成功实现了第一版本全可编程 OpenFlow 交换机。

1.2 OpenFlow 网络起源与发展

OpenFlow 起源于斯坦福大学的 Clean Slate [3]项目组。该项目的最初目标是要重新定义英特网，以改变现有复杂的臃肿不堪的 TCP/IP 基础网络架构。斯坦福大学教授 Nick McKeown 和他的学生们在 2008 年 ACM SIGCOMM 发表了题为 OpenFlow: Enabling Innovation in Campus Networks 的论文，这篇论文也就现在的 OpenFlow 白皮书的原型，它详细阐述了 OpenFlow 的概念，并列举了 OpenFlow 的很多应用场景。在此基础上，Nick 和他的团队继续提出了 SDN 的概念，这一概念参考操作系统的原理，将所有的网络设备都视为被管理的资源，同时为上层提供一个统一的管理视图和编程接口。这样，用户就可以轻松的开发相应的应用程序，来定义自己的网络拓扑结构，以满足对网络资源的不同需求，而不需要关注复杂的底层物理网络[2]。

开放网络基金会（Open Networking Foundation, ONF）正在领导着 SDN 的标准化发展，并定义了如图 1.1 所示的 SDN 框架模型。SDN 是一种新型的网络架构，它具有高性能、动态可配置、适应能力强等特点，非常适合动态应用程序以及高宽带应用程序的发展。SDN 将网络的控制层面同转发平面进行分离，它对底

[3] http://Cleanslate.stanford.com

层网络架构进行抽象，使得应用程序和网络服务能够最大化对整个网络进行控制。在图 1.1.中，应用层主要包括用户的业务应用程序，它通过定制化的应用程序接口（Application Program Interface，API）同控制层进行交互。控制层实现网络的控制逻辑，通过 OpenFlow 协议同底层网络硬件进行交互。数据层则是由各种网络设备组成，这里的设备是整个网络中的基本元素，它们主要提供转发和交换服务[3]。

围绕着 OpenFlow/SDN 的整个框架，学术界、网络设备厂商、网络运营商以及互联网公司等都对其进行了广泛的关注。到目前，包括 HP、IBM、Cisco、NEC 以及国内的华为和中兴等设备制造厂商都已纷纷加入到 OpenFlow 阵营。在数据层，一些支持 OpenFlow 的网络硬件设备已经面世，如 NetFPGA 等学术型实验平台，华为的 S12700 系列敏捷交换机，思科的 Catalyst 6800 系列交换机等 [4] 。在控制层上，linux 基金会的 Open DayLight [4]，爱立信的 NOX 等做了很多有用的工作。在应用层与网络 API 发展方面，由 Rackspace 和美国国家航空航天局（National Aeronautics and Space Administration, NASA）共同开发的云计算平台 OpenStack 正在被广泛使用[5]，并且大批公司都正在加入到其中。在部署方面，Google 公司已经在其数据中心及其骨干网络中部署了 OpenFlow 网络，并取得了很好的效果[6]。同时一些知名互联网公司如 FaceBook、百度、阿里等都在筹划 SDN 网络的研发和部署。

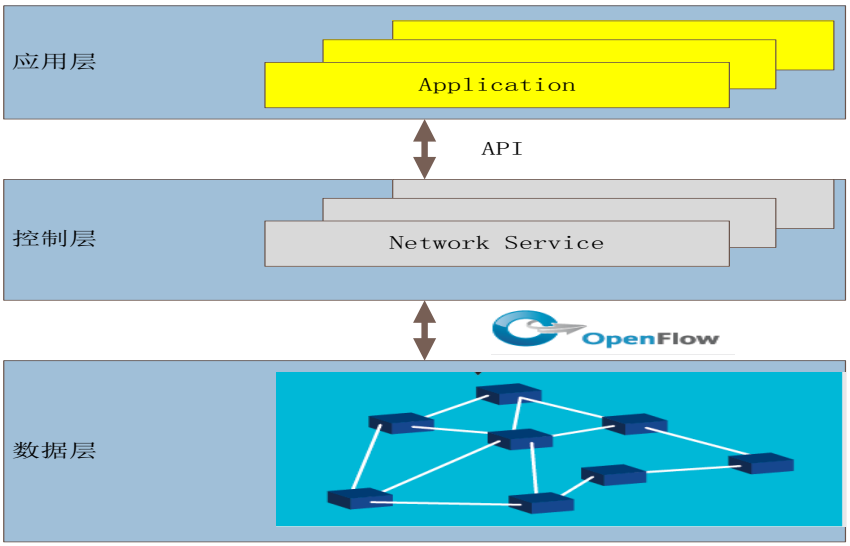

图 1.1 SDN 框架模型

[4] http://www.opendaylight.org

1.3 主要研究内容及步骤

在 SDN 模型中，本文主要以数据转发层为原型来构建 OpenFlow 交换机。这里，我们重点研究 OpenFlow 硬件流表在 ZYNQ 中的实现，这也是整个系统的核心。当我们的控制算法性能被限制后，提高数据包的匹配和转发速度就显得非常的必要了，所以我们选择 FPGA 来加速这一过程，但由于 FPGA 逻辑资源的限制，我们的流表不可能做的很大，所以在硬件部分的流表中存放的只能访问频率较高的流表项，而绝大多数的流表项都是被存放在内存中，并通过 ARM 来进行逻辑处理，也就是我们索要构建的软件交换机。

整个系统构建的主要过程以及遇到的困难如下。

1) 基本网络通路的搭建：在加入 OpenFlow 流表之前，我们需要保证基本的网络能够畅通，这一过程花费了很长的时间，问题集中表现在 FPGA 逻辑资源的分配不合理、系统时序的调整，Linux 驱动的编写上面。
2) 加入软件流表：当基本网络通路搭建完成后，我们首先在 ARM 端移植了一个 OpenFlow 软件流表，并搭建了一些简单的网络来对 OpenFlow 进行测试，这里的主要问题表现在开源项目 OpenFlow 的交叉编译上面，最终我们是通过 QEMU 搭建了一个 ARM 虚拟机来对其进行编译。
3) 加入硬件流表：我们将硬件流表作为单个模块来进行设计，对该模块进行仿真和测试后，才将其加入到整个系统中，这其中遇到的困难主要表现在数据通路的实现、Stream 总线的连接、流表项的设计与实现等。

1.4 本章小结

综上所述，本章主要对课题的背景进行阐述，通过对 SDN 模型的分析，论述了 OpenFlow/SDN 网络的发展前景。最后描述了本课题实现的主要步骤以及在设计的过程中遇到的主要困难点。相对于传统网络协议来说，OpenFlow 从 2008 年到现在才过去仅仅 6 年时间，如今发展依然迅猛，虽然它刚从实验里面走出来，前景还无法估量，但有一点可以肯定，它所带来的新思想将会在很长一段时间内影响人们对构建新网络架构的认识。

第2章 系统整体设计

2.1 OpenFlow 标准和规范

2009 年 11 月，OpenFlow 规范发布了 1.0.0 版本，现在已经更新到了 1.4.0 版本，本课题使用的是 1.3.0 版本，OpenFlow 规范是本课题所讨论的理论基础，在整个系统设计中，我们将围绕着 OpenFlow 标准来进行项目的设计和开发 [7] 。

2.1.1 OpenFlow 组件

OpenFlow 交换机主要包括一个或多个流表、一个组表、以及一个连接外部控制器的安全通道，如图 2.1 所示。

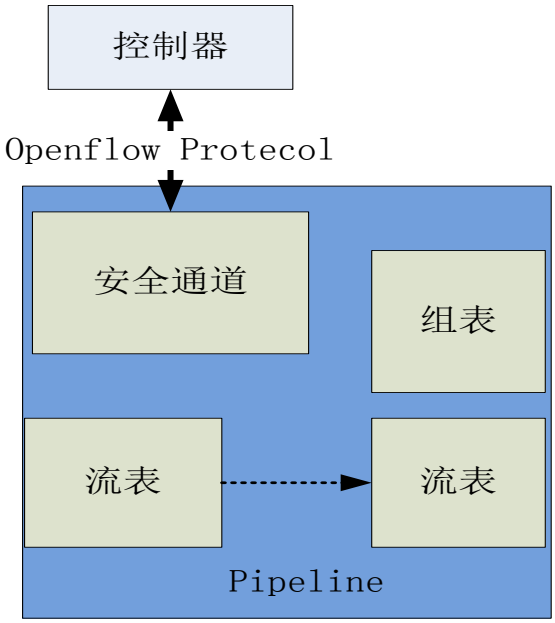

图 2.1 OpenFlow 网络

流表由流表项组成，每一个流表项主要包括匹配域、计数器、和一组命令集，如表 2.1 所示。

表 2.1 流表项定义

匹配域	优先级	计数器	命令集	Timeouts	Cookie

匹配域主要是对数据包进行匹配操作，它包括入口端口、数据包的头部、以及前一个流表所制定的操作元数据；优先级定义了该流表项匹配的优先顺序；计数器主要是用来记录匹配数据包的计数；命令集主要包括 Action 命令以及 Pipeline 的处理命令；Timeouts 定义了该流表项的生存时间；Cookie 主要用在控制器对流表的静态检测以及修改上面。

除了正常的流表项以外，每个流表都必须包含一个 Table-miss 流表项，该流表项规定了匹配失败的数据包默认的 Action。该流表项的优先级最低，以及它将会通配所有的域，从而保证每一个数据包都能够正常分配一个 Action 来与之相对应。

组表主要由组表项组成，组表项的定义如表 2.2 所示。它主要功能对流表项进行分组，并根据组类型来决定执行哪些存储段。

表 2.2 组表项定义

组 ID	组类型	计数器	Action 存储段

控制器通过 OpenFlow 协议可以对流表中的流表项进行添加、删除等操作。同时交换机也是用过这个安全通道与控制器交换信息。在数据通路与 OpenFlow 通道之间的接口都是特定实现的，然而，所有的 OpenFlow 通道信息都必须按照 OpenFlow 协议的格式来进行传输，一般都要使用 TLS（Transport Layer Security，安全传输层协议）加密，不过这里我们通常都是直接使用 TCP/IP 协议来进行通信。

2.1.2 OpenFlow 流水线处理

OpenFlow 流水线主要由一系列流表组成，对 OpenFlow 流水线的处理定义了数据包是如何同流表进行交互的，如图 2.2 所示。

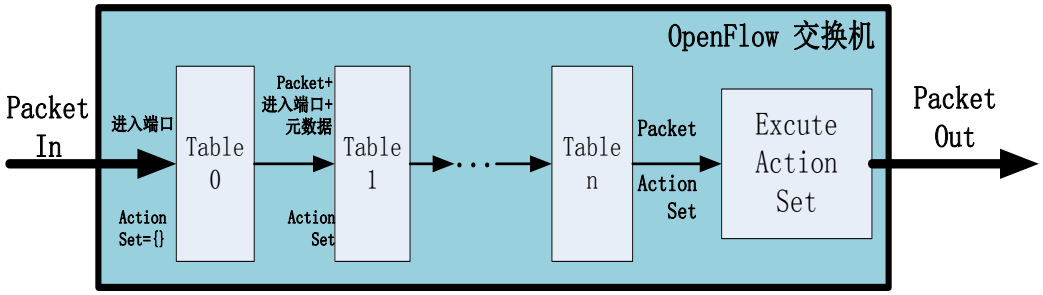

图 2.2 流水线中的多流表进行数据包匹配

流水线处理从第一个流表开始，数据包通常从 Table 0 的的流表项开始匹配，其它的流表将根据第一个流表的匹配输出结果来决定是否会被使用。当数据包在流表中进行匹配时，将首先选择优先级最高的流表项进行匹配，如果匹配成功，则该流表项所设定的一组命令将被执行，这些命令可能会直接对数据包进行转发，也有可能将其转发到其他流表做相应的处理。如果匹配失败，则 Table-miss 流表项中规定的 Action 命令将会被执行，一般会选择丢弃或者流到下一流表进行处理，或者直接转发到控制器进行处理，其处理流程图如图 2.3 所示。

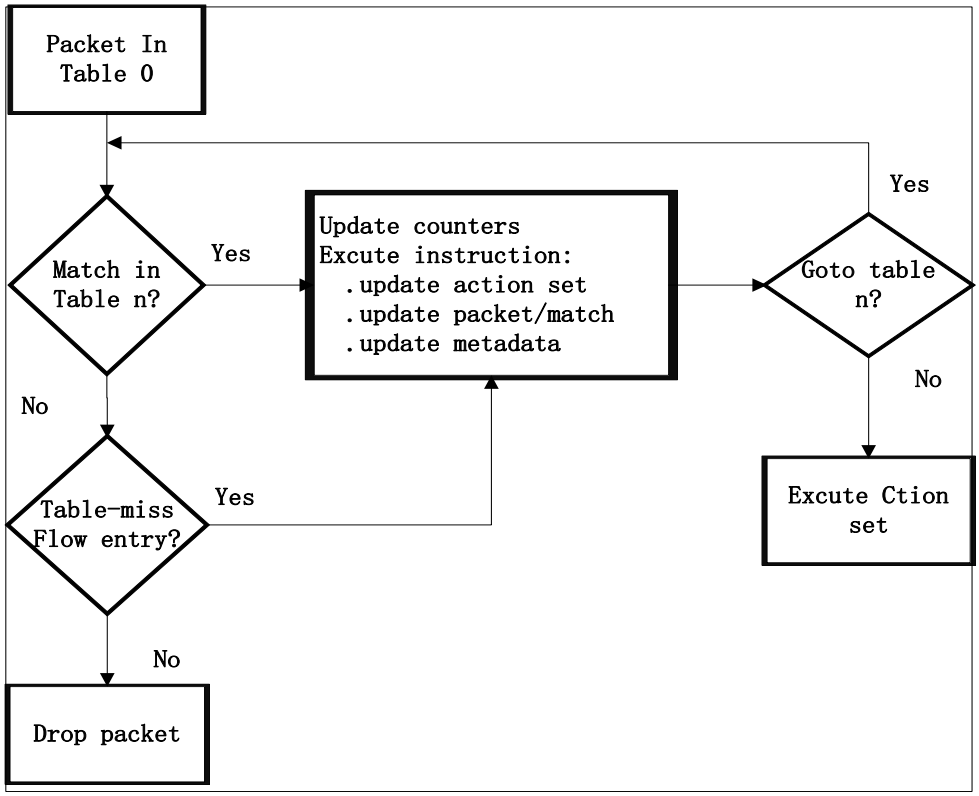

图 2.3 流水线处理流程图

2.1.3 OpenFlow 通信协议

OpenFlow 规范定义了如何与控制器之间进行连接、通讯的消息类型，并规定了如何对这些消息进行加密和进行多连接等等。一共定义了三种消息类型。

1) Controller-to-Switch 消息：该消息在控制器中进行初始化，主要包括 Features、Configuration、Modify-State、Read-State、Packet-out、Barrier、Role-Request 等消息，主要功能是控制器发起对交换机进行状态查询和修改配置等操作。

2) Asynchronous 消息：该消息由交换机发送给控制器，用来通知交换机上发生了异步事件，主要包括 Packet-in、Flow-Removed、Port-Status 以及 Errord 等消息。

3) Symmetric 消息：这个消息主要使用来进行控制器和交换机之间建立连接使用，双方都可以发起这个消息。主要包括 Hello、Echo 和 Experimenter 三种消息。

2.2 ZYNQ 全可编程框架

本课题所实现的软硬件可编程交换机，从根本上来说是由 ZYNQ 的全可编程特性所决定，如图 2.4 为 ZYNQ 的整体结构 [8] 。

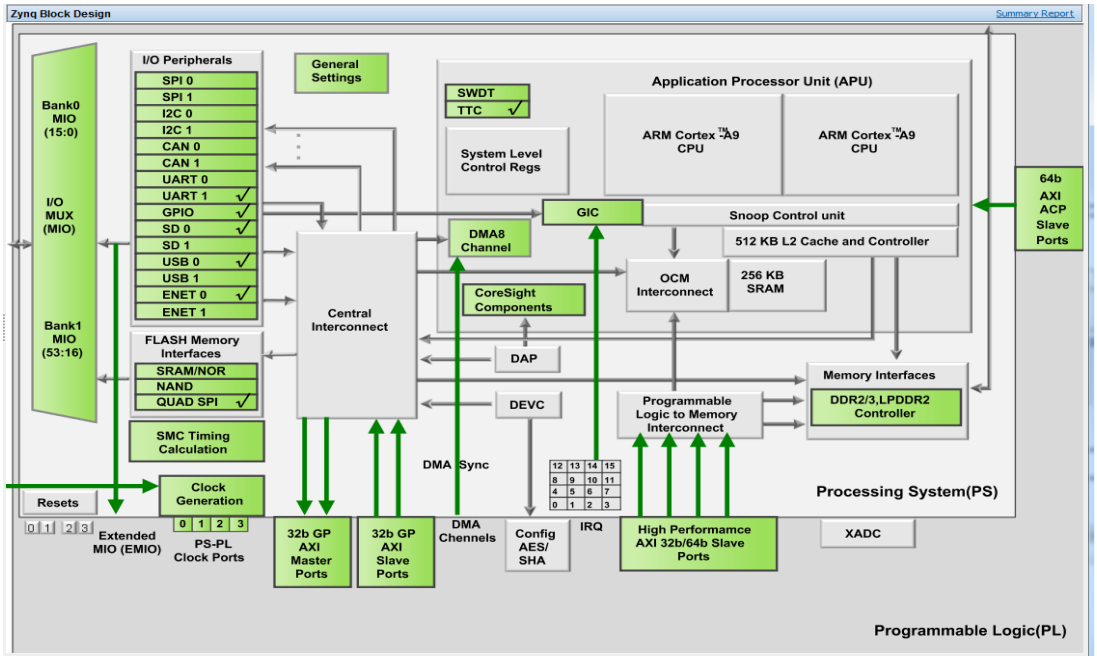

图 2.4 ZYNQ 体系架构

ZYNQ 体系结构主要包含两个部分，分别为处理器系统（Processing System，PS）部分和可编程逻辑（Programmable Logic，PL）部分。PS 部分主要由应用处理单元（Application Processor Unit，APU）、总线控制器，以及一些常用 I/O 外设，如 IIC，USB 等组成。这部分是根据 ARM 体系结构实现的微处理器单元（Micro Processor Unit, MPU），它直接使用硬核嵌入到芯片之中。APU 是 PS 端的核心，它主要由双核 Cortext A9 处理器、NEON 多媒体协处理器、多级 Cache、DMA、中断控制器、以及一个 256K 片上内存（On-Chip Memory, OCM）组成。

外围可编程逻辑部分为 7 系列 FPGA，实现工艺为 28nm。PL 为 PS 部分提供了丰富的可编程逻辑资源和接口资源， 最重要的是这些资源是用户可以定制的，从而对系统的可扩展性提供了保障。对于本课题而言， 我们的绝大数工作都将在 PL 端完成，包括 MAC 软核，DMA 软核，OpenFlow 硬件流表等。

进行软硬件协同设计，我们首要考虑的是如何进行 PS 端同 PL 端的通信问题，ZYNQ 不同于其他的 MPU+FPGA 平台就在于其为用户提供了 AXI 总线接口，它使用户从复杂的总线设计中摆脱出来，从而更加专注系统设计。有三种总线接口可供选择，分别是 AXI GP 接口、AXI HP 接口、以及 AXI ACP 接口。AXI GP 接口提供了低速的通路设计，一般我们用这个接口来对设备寄存器进行配置；AXI HP 接口适合高速的总线设计， 一般使用它来进行大数据的传输，比如网络数据包、实时图像信息等；AXI ACP 接口提供直接的 L2 Cache 访问的功能 [9]。

2.3 系统架构设计

至此，我们开始构思系统整体架构。通过对 OpenFlow 规范的理解以及 ZYNQ 架构分析，最终构建全可编程 OpenFlow 交换机的整体结构如图 2.5 所示。

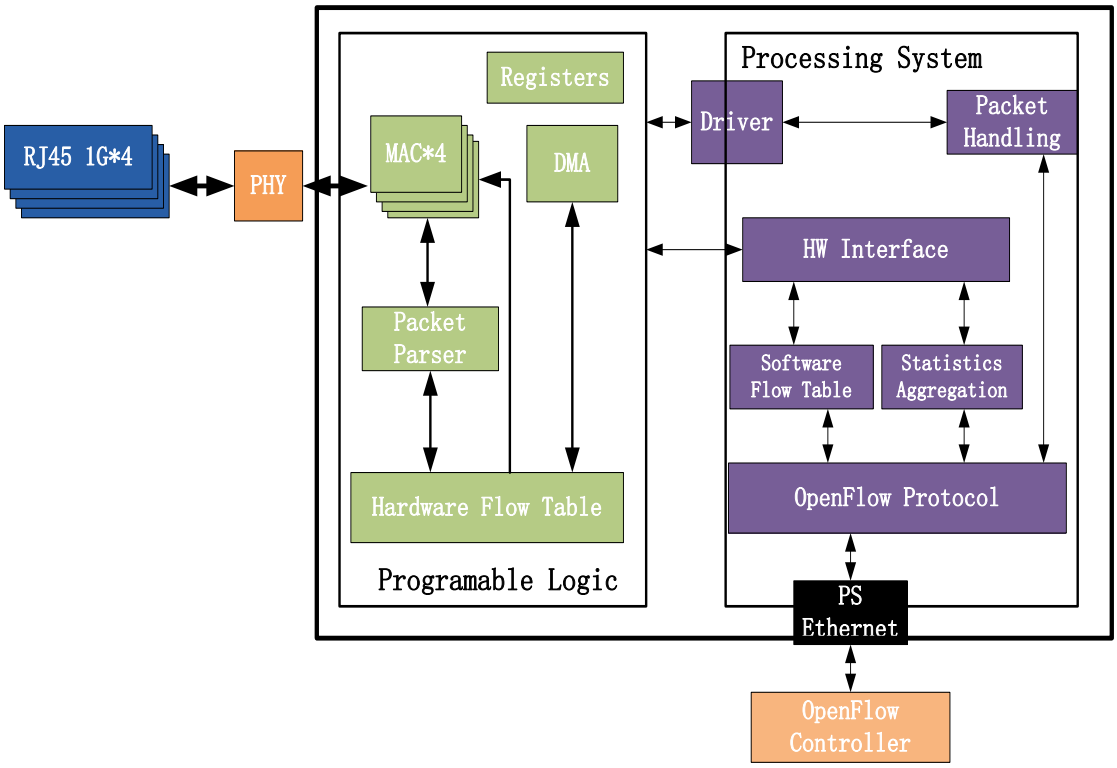

图 2.5 OpenFlow 交换机整体结构

本系统使用 4 个千兆以太网网口接入到 PL 端，数据包首先经过 MAC 软核的处理，之后会按照 OpenFlow 匹配域的头部格式进行解析，并将原有数据包保存到 fifo 队列中。对解析后的数据包进行流表匹配操作，如果匹配成功，则对相应的数据包进行转发，否则按照默认 Action 进行处理，这里我们一般会将其通过 DMA 转到 PS 端进行软件流表匹配，或者直接进行丢包处理。Register 模块主要功能是实现用户接口对流表的配置，查询计数状态以及设置时间戳等功能，它向上层提供了流表的编程接口。

在 PS 端，操作系统通过驱动程序对底层硬件进行抽象，并提供底层网络设备的操作接口，在这个基础上，我们来进行 OpenFlow 软件流表的构建，并按照 OpenFlow 流水线对数据包进行软件处理。外部控制器通过 PS 端的以太网端口同 OpenFlow 交换机进行连接，并按照 OpenFlow 协议规范对软硬件流表进行配置，状态查询等操作。

系统整体的数据通路如图 2.6 所示。使用 HP 端口来进行 DMA 与 DDR3 内存之间的数据传输，通过一个互联控制器，AXI DMA 将它的 scatter-gather、Stream to memory mapped (S2MM) 、以及 memory mapped to stream (MM2S) 接口同 HP 端口连接起来。这个互联控制器的功能是进行数据流的选择以及将 AXI DMA 32 位的数据转换为 HP 端口 64 位的数据 [10]。

OpenFlow 模块在 AXI DMA 与 AXI Ethernet 之间，所实现的数据接口为 AXI Stream 接口，流表操作结束后可以直接从 Ethernet 转发，也可以通过 DMA 将数据传到 PS 端做进一步处理。

AXI Ethernet IP 能够对数据包进行全面的校验，并进行 MAC 层数据处理，这里，它通过一个 32K 的先进先出 (FIFO) 队列来对大数据帧进行传输。该 IP 核通过一个精简千兆媒体独立接口（Reduce Gigabit Media Independent Interface, RGMII）同 PHY 芯片 BCM5464 进行连接，并通过 MDIO 接口对 PHY 进行配置。

系统使用通用 AXI GP Master 端口来管理所有模块的控制寄存器，这个总线类型为 AXI Lite 总线，并通过这个接口来对所有的寄存器进行状态查询，当模块通过 AXI GP Slave 接口连接到这个接口的 Interconnect 时，它将对相应模块分配一个物理地址，这样，我们在操作系统中通过内存映射就能很好呢容易的对底层硬件设备进行控制。特别的，对 PHY 的配置， 我们使用的 Ethernet 的寄存器，当其接收到数据后，会自动启用 MDIO 来对 PHY 进行配置，而我们所要做的就是对相应的寄存器写数据，其他模块也是一样。

BCM5464 集成四组 1000/100/10Base-T 收发器，配置为 RGMII-to Copper 模式，这 4 个接口的状态以及控制寄存器都是完全独立的。它将 RJ45 接口输入的电平信号转换为 RGMII 接口数字信号[5]。

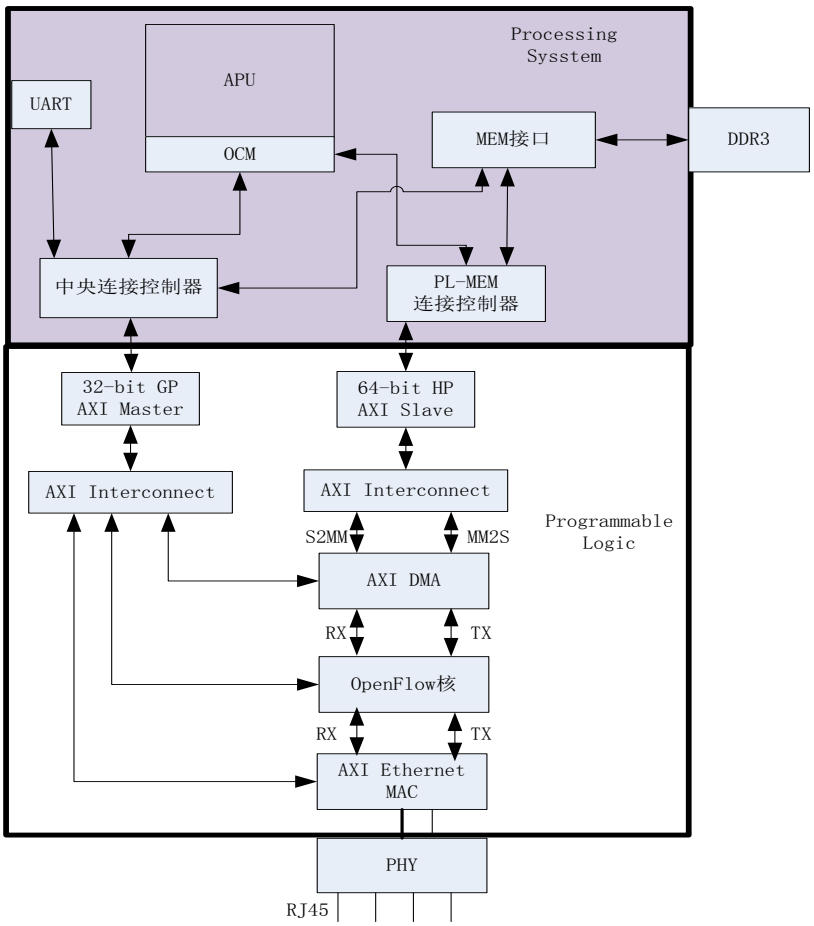

图 2.6 系统数据通路

2.4 本章小节

本文全篇贯穿了软硬件协同设计的思想，本章对 OpenFlow 规范协议进行了简单的介绍，特别对 OpenFlow 流表的流水线处理进行了介绍，这是 OpenFlow 的核心操作。第二节对本系统的基础硬件平台进行了介绍，对 ZYNQ 的全可编程特性有了一个全新的认识，了解了 ZYNQ 的 PS 端与 PL 端的组成元素以及他们之间进行通信的总线类型。

2.3 节在前两节的基础上来对系统的整体框架进行构建，这里给出了一个大体

[5] BCM5465SR 数据手册，保密文档，未公开。

的框架，在下面的章节中，我们会对各个模块展开进行详细的讲解，这章是整篇论文的理论基础。本章最后给出了系统的数据通路，并详细介绍了数据在整个交换机中的流动状态，网络数据包在 FPGA 状态机以及驱动程序的控制下进行流动。

整个系统在内部总线的驱动下进行工作，通过定义不懂的总线类型来实现对不同类型数据的传输管理，有效的利用了 ZYNQ 的资源，最大化系统数据吞吐率。使用 AXI Inerconnect 来对 PL 端的模块进行管理，使得数据能够有效进行传输，总线也能很好的管理数据的流动。

第3章 OpenFlow 硬件流表

3.1 硬件架构整体概述

图 3.1 描述了 OpenFlow 硬件实现的顶层视图，它包括 3 个部分，第一部分为 PHY 层到 MAC 层之间的数据通路，第二部分为 OpenFlow 硬件流表数据通路，第三部分为各模块之间的接口。其中，我们将对对第二部分进行重点论述。

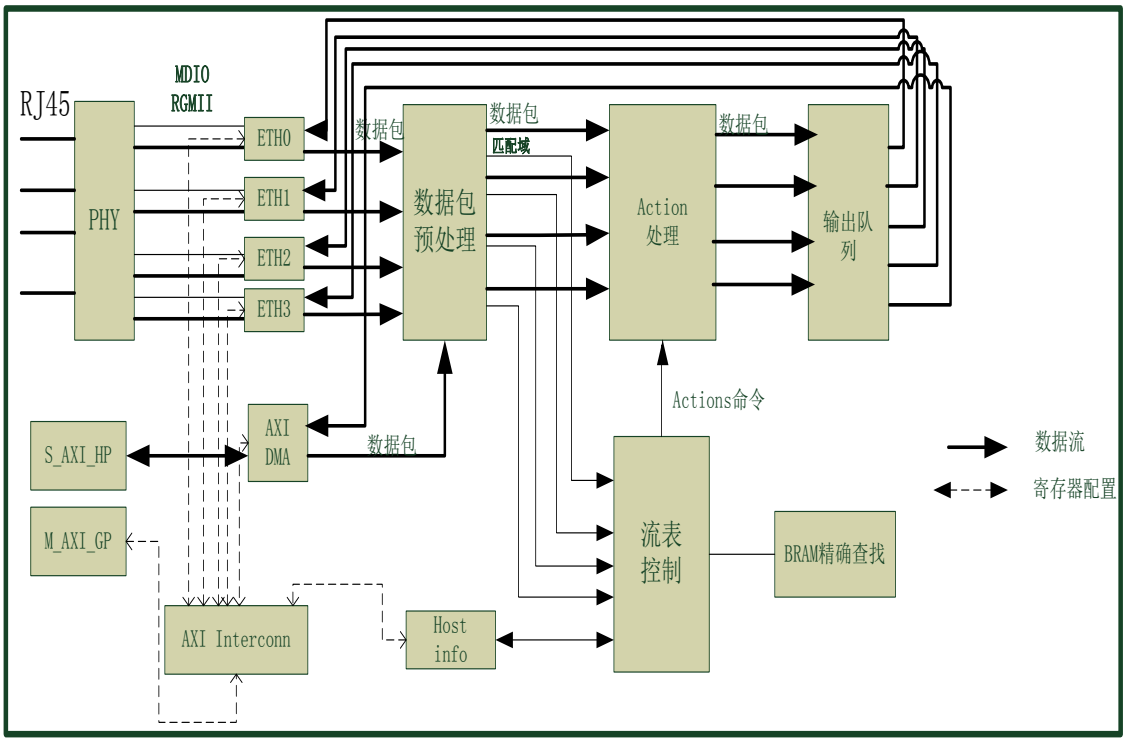

图 3.1 OpenFlow 硬件实现架构

图中的实线部分代表数据流，其中加粗的部分表示数据包的传输通路，它使用的是 AXI Stream 协议进行数据传输，虚线部分主要是对寄存器进行配置，它使用的是 AXI Lite 协议。流表操作模块由数据包预处理、Action 处理、Host info 以及流表控制 4 个小模块组成。输出队列模块主要进行端口数据的转发并同实现了同 AXI DMA 的接口 [11] 。

3.2 OpenFlow 模块构建

OpenFlow 交换机在以太网基础之上进行构建。在本设计中，数据流从 RJ45 接口进入， 经过 PHY 的转换，到 AXI Ethernet 的链路层处理，最后才进入到 OpenFlow 模块。这里 AXI Ethernet IP 核使用的是 Xilinx 官方提供的 IP 核，由于这并不是论文的重点，所以就不再对其进行论述详细介绍。

OpenFlow 模块之间使用 AXI Stream slave/master 接口进行数据包的传输，并使用 AXI Lite 接口来对寄存器进行配置。所有的 AXI Stream 总线共享 125Mhz 系统时钟，AXI Lite 在 75MHz 时钟下进行工作。途中的虚线部分标明了 AXI Lite 总线数据通路，它通过一个 AXI Interconnect 连接将 M_AXI_GP 接口同各模块的 AXI Lite 接口连接，ARM 处理通过这个总线将数据从内存写入到各模块的寄存器中。在设计中的每个模块都应该链接两条数据通路总线上，高速总线主要实现数据包的分析和处理。低带宽的寄存器数据通路总线主要实现各模块寄存器的读写，从而完成用户软件对模块功能和参数的配置。

在输出队列中，本设计使用了 4 个数据通路流水线来对数据包进行转发和传送处理，在 AXI DMA 中启用 4 个通道，使得每个流水线的数据传输不会相互干扰。由于本设计将会对每个数据包进行包头解析，然而包头域可能会发生改变，以及可能会随时增加或删除数据包的标签，所以使用短数据位宽 (64bit) 多通路流水线还是非常必要的，它可以在一个时钟周期内完成多个任务，从而对数据流进行快速的更新。

当前版本硬件流表支持 OpenFlow1.0 规范。只有两级流表，第一个流表对 2 层以下进行匹配，也即对 MAC 地址和端口进行匹配。第二级流表对三层以上进行匹配，也即对 TCP/IP 进行匹配。一条硬件流表项的内容如表 3.1 所示 [12]。

表 3.1 硬件流表一条 entry 内容定义

匹配域 64bits	匹配域 mask 64bits	actions 320bits

3.2.1 OpenFlow 数据包包头解析

包头解析是 OpenFlow 对数据包进行处理的第一个模块，它对输入的数据包进行解析，并对解析后的域进行组织，从而使得其和流表项的匹配域能够匹配。最后它将解析后的位图 (bitmap) 通过握手信号传送到流表控制模块，并直接将原始数据包转发到 Action 处理模块 [13]。

该模块包括两个数据包缓冲区 (input_fifo, output_pkt_buf) ，一个包头解析模

块 (header parser) 以及一个流表项编辑器(entry_composer) 。其基本框图如图 3.2 所示。

输入队列是一个先进先出队列，在其他模块中也会使用到，它的主要作用是对数据包进行缓存与数据同步。数据包首先经过输入 FIFO 队列缓冲，之后被送到 Header Parser 和数据包缓冲模块中进行处理。

Header parse 模块对数据包包头做解析，每一个解析模块都是使用一个状态机来进行维护，如图 3.3 所示为 ARP 解析处理的状态机。

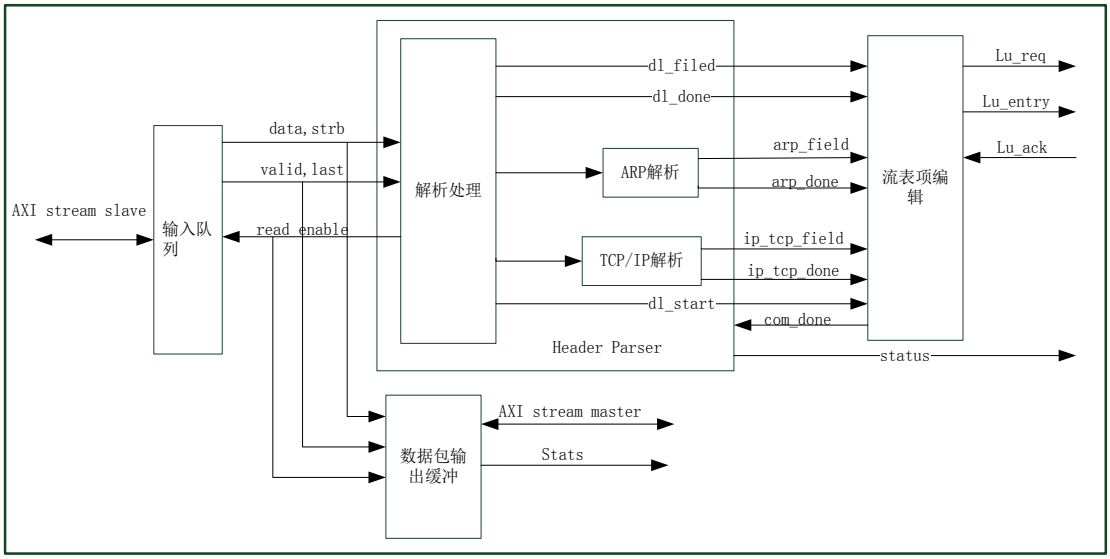

图 3.2 数据预处理模块图

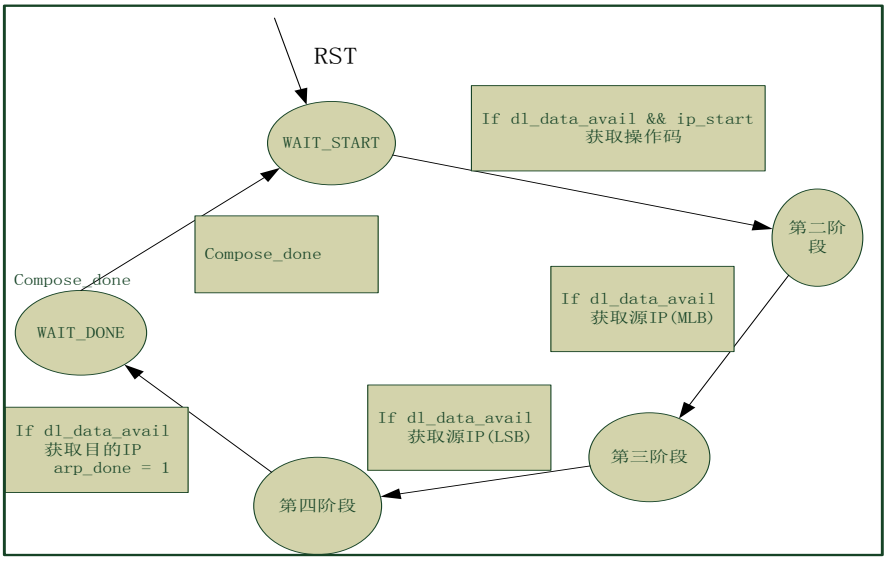

图 3.3 ARP 解析状态机

第一部分解析模块的状态机相对复杂，它主要是完成与输入 FIFO 队列的同步，检测以太网数据包的类型，比根据相应的类型启动相应的状态机。ARP 解析的过程相对简单，当 compose_done 信号有效，ARP 状态恢复，ARP 解析完成。如图 3.4 所示为 IP_TCP 解析状态机。

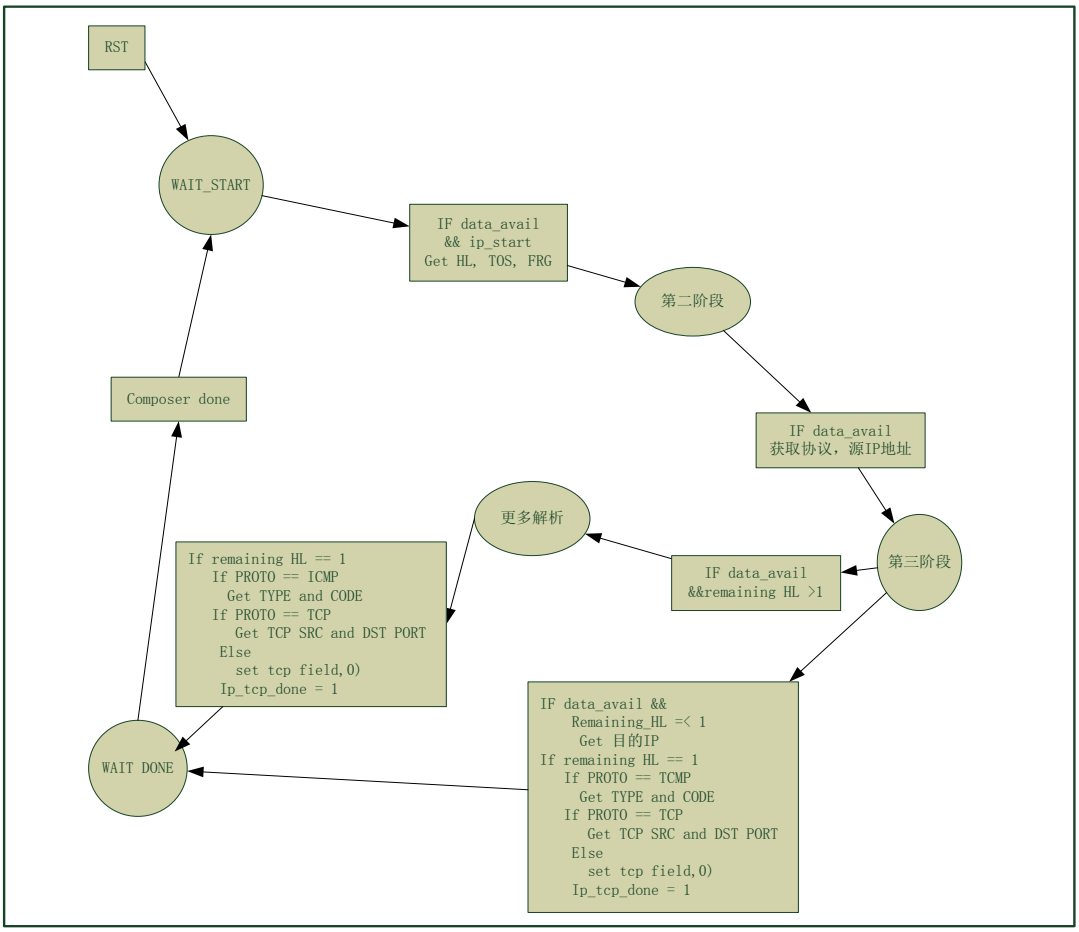

图 3.4 TCP/IP 数据包解析状态

当数据解析完成，流表项编辑模块将会为相关的解析创建一个 bitmap，并产生一个 request 信号向流表控制模块请求进行流表匹配操作，当收到允许信号后，就将创建的 bitmap 转发到流表控制模块中进行处理。此时 Header parser 模块处于等待状态，当 lu_ack 信号有效后，它将清除 request 信号以及刚创建的 bitmap。

数据包输出缓冲模块一直在等待 read enable 信号，当数据包解析完成，read enable 信号有效，从而激活数据包处理模块，之后数据包输出缓冲模块会将数据包通过 AXI Stream 送入到 Action 处理模块，并在数据包传输结束后使 AXI Stream 的 TLAST 信号有效。该缓冲区的深度为 128 字节，能够支持 TCP 数据包带有两

个 VLAN 标签，以及 16 个字节的 IP 选项。

3.2.2 Host info 模块

Host info 模块是 OpenFlow 硬件与 PS 端的交互接口。该模块使用 AXI Lite 总线连接到 PS 端，所有对 OpenFlow 模块的读写控制以及 OpenFlow 硬件模块本身的状态信息都是通过这个模块获取。该模块会暂时存储从软件层面写入的流表项信息，并通过一个握手信号将流表项以 bitmap 的方式传到流表控制模块。

3.2.3 流表控制模块

流表控制模块与流表和所有的请求信息进行交互，它使用一级流水来对流表进行处理，所有的端口通过时间片来共享这个流水线。如图 3.5 为该模块的结构图。

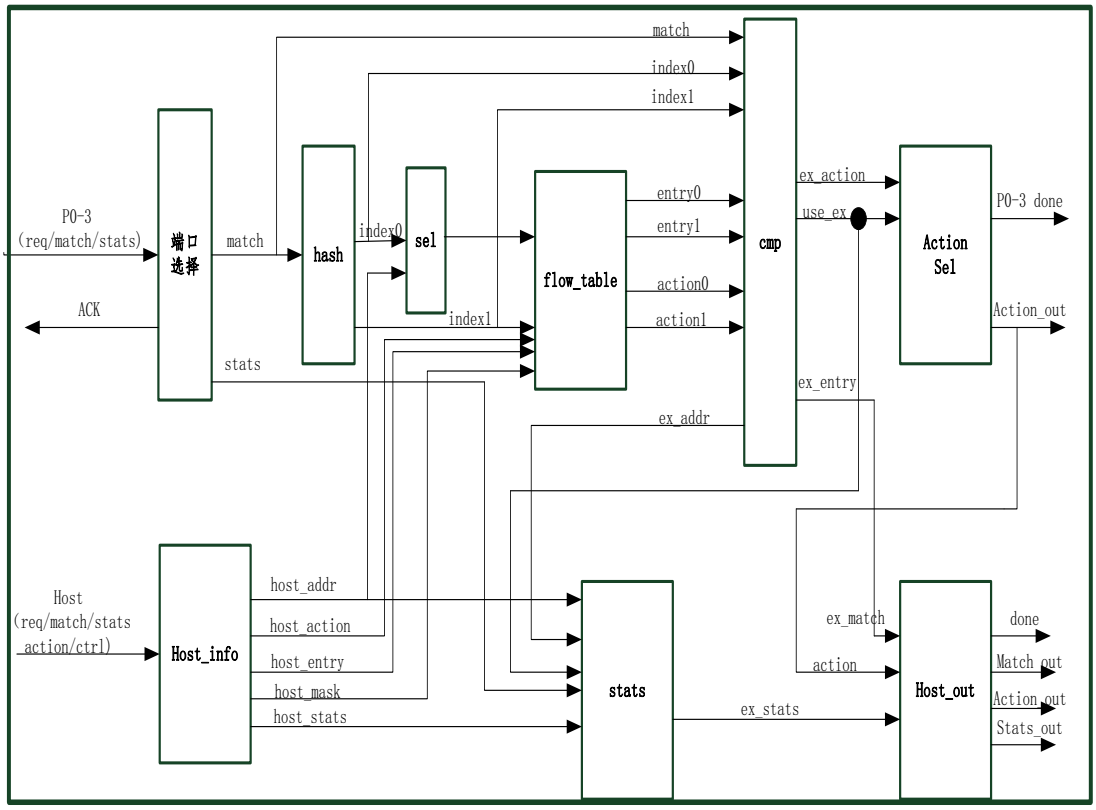

图 3.5 流表控制模块结构图

当有端口向流表控制模块查询是否有流表匹配成功时，控制器首先进行流表的精确查找，通过对端口的 hash 选择以及流表的匹配操作，如果找到一个匹配项，

它会从流表中取出相应的 action 并从源端口送出[14]。

3.2.3.1 接口说明

（1）请求/回应接口：所有的端口使用相同的接口

a) p<0-3>_req:当其中一位置位，说明这个端口上有包头解析查询操作。当 p<x>_ack 有效时，p<x>_req 为 0。
b) p<0-3>_match: 包头解析传过来的 bitmap 信息，它将用来对流表进行匹配查询。
c) p<0-3>_ack: 当 p<x>_match 开始进行以及 request 信号接受到后，由该模块发送的应答信息，但这并不以为着流表处理结束。
d) p<0-3>_done:输出信号，当过程处理结束，该信号有效，并在下一个时钟周期发送到接收端。
e) actions: 输出信号，该信号表示匹配操作后相应的 action，如果匹配失败，所有的 action 都为 0 表示没有匹配项，则选择默认选项将数据博丢弃。该信号将在 p<x>_done 过程结束后才会有效。

（2）Host 接口：当 PS 端主机向流表中写入一条流表项时，写入的数据将会在同以流水线时间里从 Host Info 模块里到达流表控制模块。所以这两个模块的握手接口信息是相同的。

a) host_addr: 寄存器地址信息
b) host_req: 同 p<x>_req 相同,当有效的时候，表示请求写入一条流表项
c) host_ack：同 p<x>_ack 相同，应答信号
d) host_done：同 p<x>_done，如果 host_req 是写操作，则 host_info 模块可以将一条流表项信息写入。
e) host_write: 如果为高，表示 host 想要进行写操作，如果为 0，表示 host 想要进行写操作，这种状态将会维持到 host_done 被置位。
f) host_stats_en: 当 host_write 为 1，可以将 host_stats_en 连同 host_mask_in、host_match_in、host_action_in 一起写入，否则不写。
g) host_match_in: 将要写入的流表项的匹配域。
h) host_mask_in: 将要写入的流表项的 mask。
i) host_action_in:将要写入的流表项的 action。

j) host_match/mask/action_out:将要被读出的流表项信息。

3.2.3.2 流表操作流程介绍

在流表控制模块中，一个流的匹配操作和流表项的注册是在同一个流水线中进行处理。这个流水线分 8 个阶段完成，没个阶段使用一个时钟周期。而对于查询请求则使用 7 个阶段来完成流水线操作，所以请求查询将会在 7 个时钟周期后完成。

第一阶段和第二阶段主要完成端口的选择。端口选择模块接收来自 4 个端口的请求流表查找操作，并使用一个时钟周期来完成端口的选择，并将相应端口的流表项信息传递到下一个阶段，然后再使用一个时钟周期来完向相应的端口发送应答信号 ACK，表明端口选择完成，同时这里的请求还包括了 PS 端对流表项的读写操作，它也被视为另外一个端口。

第三阶段主要完成输入流表项信息的处理，这里使用 hash 算法来对其进行处理。hash 模块会根据送入的流表项信息使用 32 位的循环冗余校验码 (Cyclic Redundancy Check 32, CRC-32)生成两个地址 index0 和 index1，在软件端也是使用相同的机制来选择流表项将要写入的地址。这个阶段对于 host 请求来说他将不会做任何事情，它将会直接将相应的数据传到下一个阶段。

第四阶段和第五阶段主要完成流表的查找操作，flow_table 模块会根据传入的地址将需要进行匹配的流表项传入到下一个匹配模块中。在第一阶段中介绍到，如果请求的是 host，那么 flow_table 将会通过固定端口来对流表项进行读写。

第六阶段主要完成流表的匹配操作，这一阶段将会对上一阶段传过来的两个流表项进行匹配，选择其中最大匹配的流表项传入到下一个阶段，并增加错误计数器，表示另外一个并没有匹配到。如果是 host 请求，则直接将 host 地址转到下一个阶段，而不做匹配操作。匹配使用 CAM 算法进行匹配，该算法采用以空间换时间的方式在一个时钟周期内完成一条流表项的匹配 [15] 。

第七阶段主要将匹配结果返回到到请求模块。这里它会读取当前的状态信息，并将其送入下一个阶段。在 action_sel 模块中，匹配信息将会通过请求端口返回，并发送匹配完成信号 done。同时相应流表项的统计数据也会被读取。

最后一个阶段主要完成流表状态的更新，如果是匹配流表成功，相应的流表统计数据将会被更新，如果是 host 接入，那么相应的数据将会被被更新到流表。跟新的信息包括数据包计数，字节计数，以及当前时间值。

3.2.4 Action 处理模块

这个模块所扮演的角色就是确定端口的转发以及更新数据包头部域和长度，这里的 action 指的是从流表控制模块接收到的 action 指令，其模块结构图如图 3.6 所示。

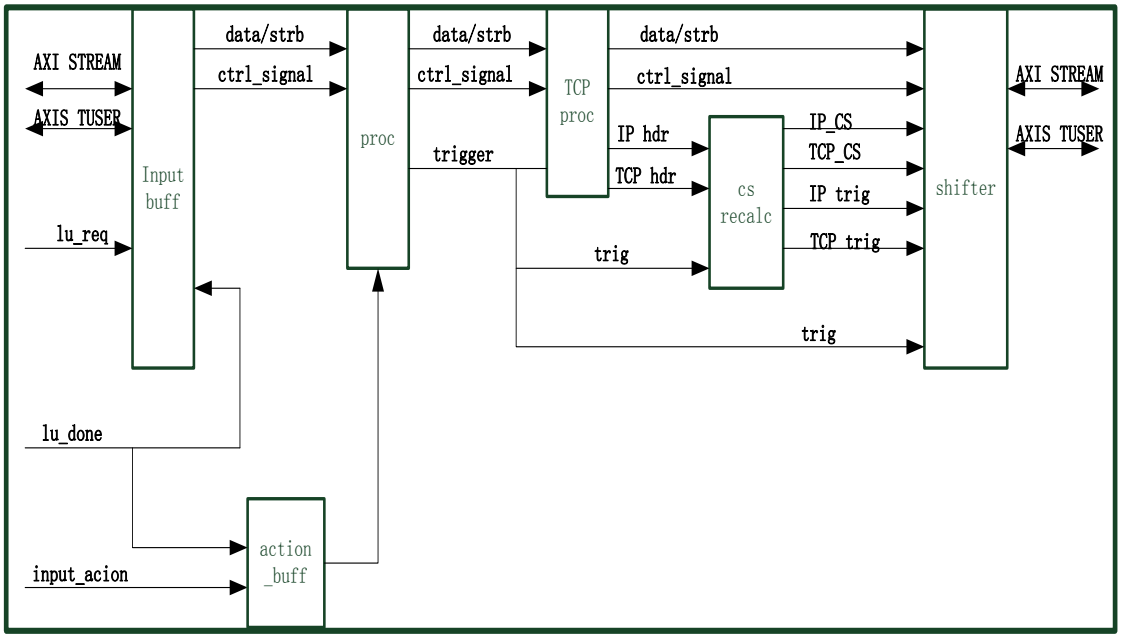

图 3.6 Action 处理模块结构图

当 lu_req 有效时，表明前面的数据包预处理模块有数据包需要传进来，这时 Action 处理模块发送读信号将数据包从前面的模块中取出，并放入输入缓冲 FIFO 队列中，该模块会在第一个时钟周期内将数据锁存，当 lu_done 被置位后，说明流表控制模块已经处理完成，这时，action buffer 将 action 锁存，并发送 read 信号给 proc 模块。proc 模块在收到 read 信号后，将 Input buffer 中的数据取出，并开始进行处理 [16] 。

处理的过程主要是将 TUSER 中的转发端口数据进行更新，之后对数据包进行逐层次的处理，包括对第二层中的域进行更新，添加或删除 VLAN 标签，修改 3 层或 4 层的校验等，最终对数据包的长度进行更新。

TCP proc 模块主要是将 action 中的信息更新到相应的位图中，这里主要对 IP、TOS、TCP 端口等进行修改。当然， 数据包可能含有 L2 的标签，所以应该分不同的状态来对数据包进行处理。

cs_recalc 模块的主要功能是对 IP 数据包进行重新校验，这里根据 RFC1624，

来对数据包进行校验，新校验和为：

NewCheckSum = ~(~OldChecksum + (~OldData1 + NewData1)
+…+(~OldDataN + NewDataN))

Shifter 模块主要是对数据包进行缓存，但前提是 TCP/IP 的校验和被重写以及数据包被移动了 5 次，否则将丢弃数据包，并清除 TUSER 中 dst_port 选项，这样，在输出队列模块中将找不到端口进行数据包的转发。

3.3 IPcore 整体集成

前面我们介绍了整个硬件工程的框架，并对每个模块的功能结构进行了描述。下面我们将使用 Xilinx Vivado 集成开发环境来对整个工程进行集成实现。

3.3.1 Vivado 设计套件

Vivado 是一个集成开发环境，他主要功能是进行 Xilinx FPGA 芯片的设计与开发，并对在 Xilinx 全可编程芯片中进行嵌入式设计提供了很好的支持，尤其对 ZYNQ 做了很多的优化设计 [17] 。

在 Vivado 中，我们使用 IP Integrator (IPI) 来对系统进行设计。这里的 IP 是 Intellectual Property 的简称，它是一段具有特定电路功能描述语言程序。使用 IP 设计 FPGA 系统，引用非常的方便，并且能够对其进行配置，以改变基本元件的某些功能。在 Vivado 中可以很容易引用别人设计的 IP， 并能够轻易的构建属于自己的 IP，Xilinx 官方也提供了很多丰富的 IP 核，包括本设计中使用到的 AXI Ethernet、 AXI DMA 以及其他的一些 IP 核， 我们只需要拿到该 IP 核的许可证就能轻易的将其添加到自己的系统中，当然，很多的 IP 核是免费的。

IP 另外一个很大的优点在于，用户可以不用关心该 IP 内部的实现细节，而是站在系统结构的层面上来对系统进行设计。Vivado IPI 以 IP 为基本元素，通过可视化界面来对 IP 进行操作，包括对 IP 进行配置，以及 IP 核之间的端口连接和总线连接等，简化了以往 FPGA 设计的难度，并能够快速直观的构建系统。

对于 ZYNQ 芯片，Vivado 提供了很多的支持，包括内部时钟的自动约束、PS 端自动化设置、DDR 总线连接与约束、AXI 总线的自动匹配与连接等。这些功能为本设计提供了不小的帮助。

连同 Vivado 一起安装的还有 Xilixn Software Development Kit (SDK)，当我们在 Vivado 中完成硬件工程后，一般都要将其导入到 SDK 中进行测试，在 SDK 中

我们可以通过相应硬件工程的板级支持包来对硬件设计进行测试，这里的测试主要是对双核 Cortex A9 进行编程，控制总线来对 PL 端的硬件的寄存器进行读写，从而实现 PS 端与 PL 端的交互。

为了在 PS 端启动 Linux 操作系统，我们还需要在 SDK 中制作启动镜像和 linux 设备树文件，通过 QSPI 或者 SD 卡来启动 linux 操作系统。

3.3.2 系统 IP 集成

如图 3.7 为 Vivado 系统集成图，图中已经对主要的模块进行了说明。图中使用了 4 个 AXI Ethernet IP 核，该 IP 核由 Xilinx 官方提供，主要处理 MAC 层数据；OpenFlow 硬件模块是使用 Vivado 自己创建的用户 IP，主要完成流表的匹配以及转发端口的选择；AXI DMA 配置为 4 通道，来完成 4 个接口的网络数据包传送。其他的一些模块主要是 AXI Interconnect 模块，主要完成数据格式转换和数据通路的选择。

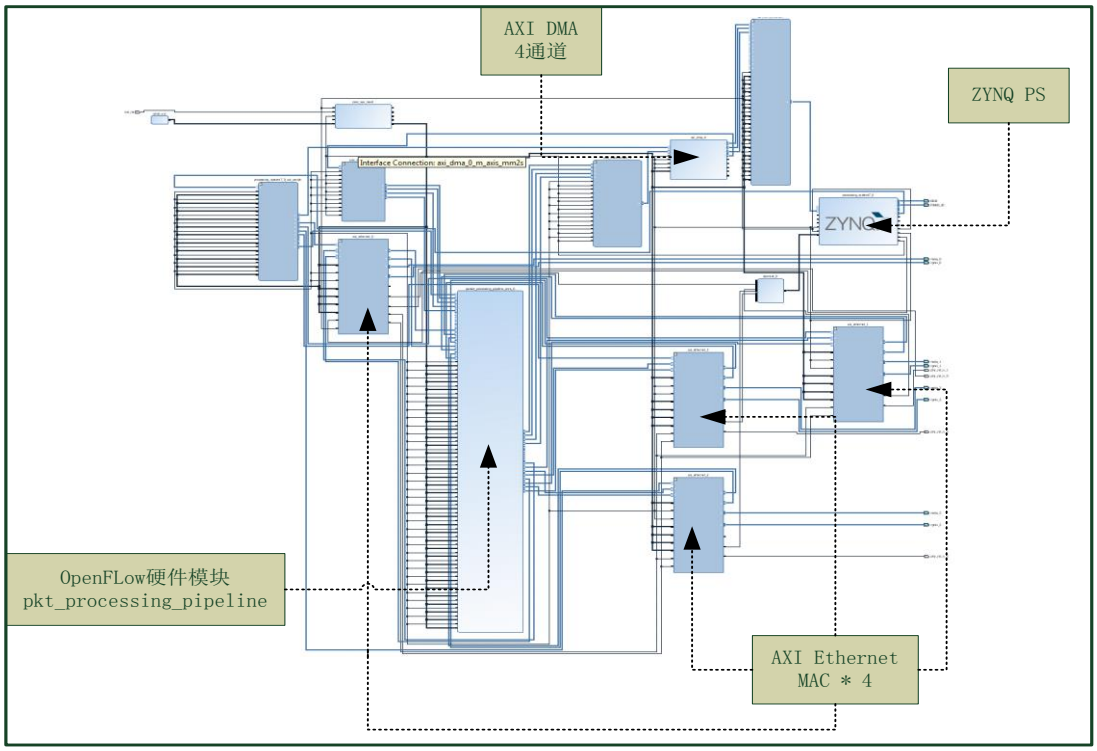

图 3.7 IPI 系统集成

系统使用 PS 端的 3 个时钟资源来提供时钟脉冲，如图 3.8 为 PS 端时钟配置选项。FLCK_CLK0 为 125MHz,主要提供 AXI Stream 的时钟需求，FLCK_CLK1

为 75MHz,主要提供 AXI lite 的时钟，FCLK_CLK0 为 200MHz 主要提供的是 AXI Ethernet 参考时钟。

为了节约系统资源,采用一个 AXI DMA 4 通道来处理数据的传输,通过 AXIS Interconnect 来做数据通道选择，如图 3.9 所示为 DMA 的读通道，对于写通道也是使用相同的方法将 DMA 与 OPenFlow 的 Stream 总线接口进行连接，在驱动层进行相应的驱动 [18] 。

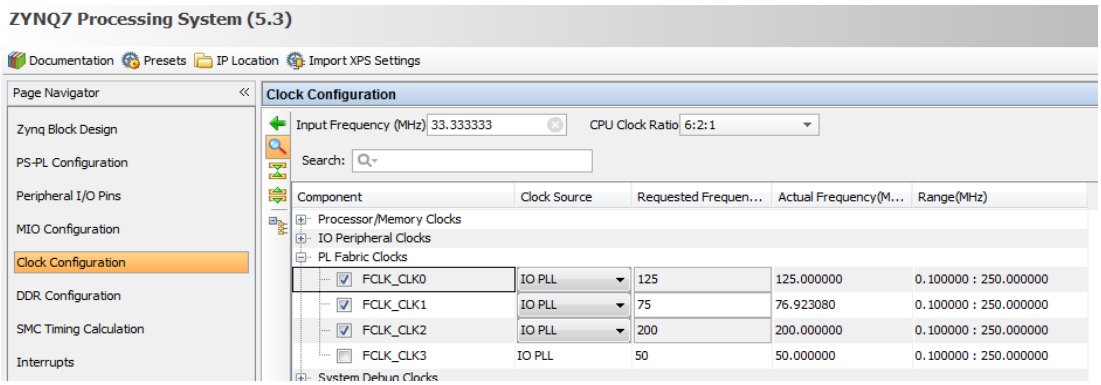

图 3.8 PS 端系统时钟配置

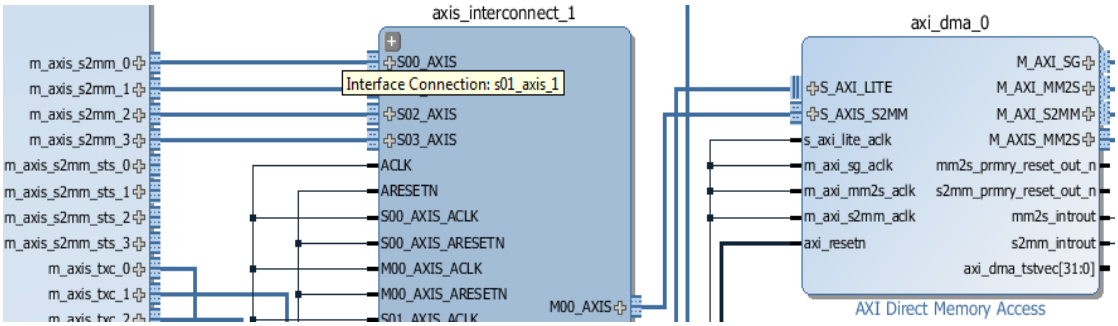

图 3.9 DMA 接口连接

PS 端向 PL 端提供了 16 根可配置中断线,通过 xlconcat 来对中断信号进行选择，这里主要使用其中两根用于 DMA 的读写中断控制，如图 3.10 所示。

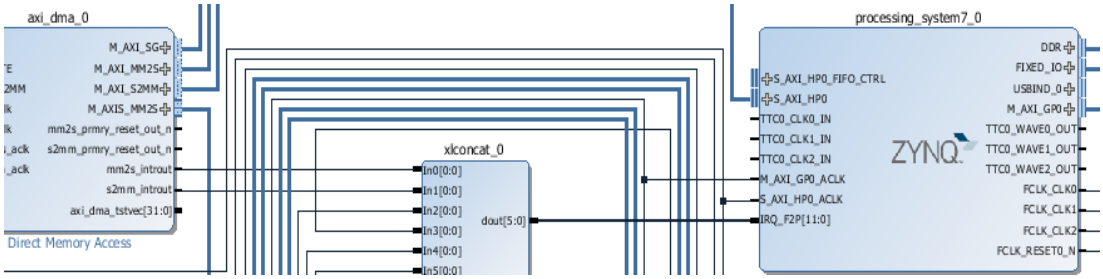

图 3.10 中断连接

考虑到 FPGA 的全局时钟资源的问题，4 个 AXI Ethernet 采用共享全局时钟资源的方式进行连接。只需要将 AXI Ethernet 0 配置为可共享即可。同时配置 AXI Ethernet 的接口方式为 RGMII,如图 3.11 所示 [19] 。

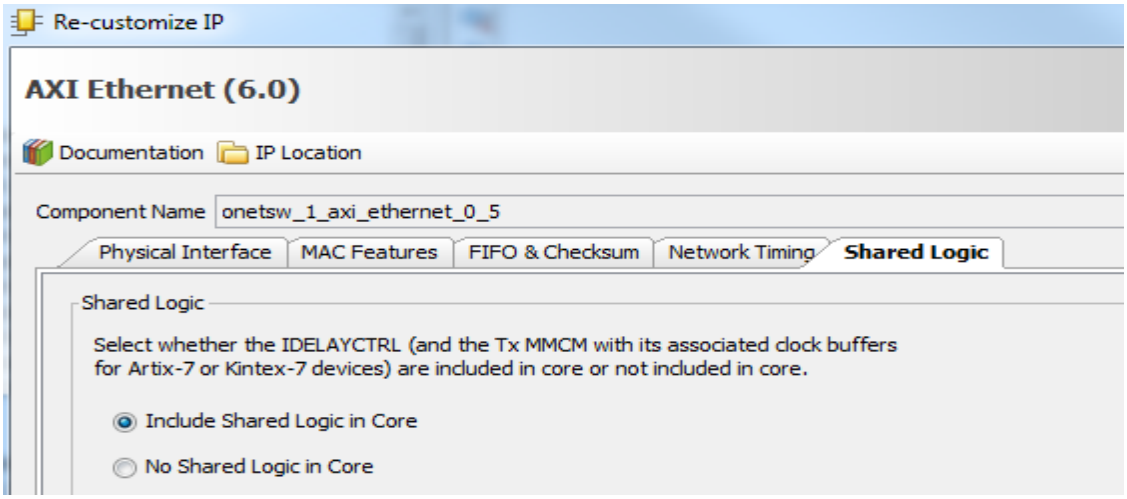

图 3.11 AXI Ethernet 配置

为了决解丢包的问题，应对 PHY 时序进行调整，主要是通过约束来设置 4 个 MAC 的 RGMII 的 rx 差分信号线的延时分别为 8、6、4、4 即可。

图 3.12 为系统的物理地址映射，在今后我们要对相应的设备进行控制，就是通过这些地址在加上一个偏移量来对设备进行读写。

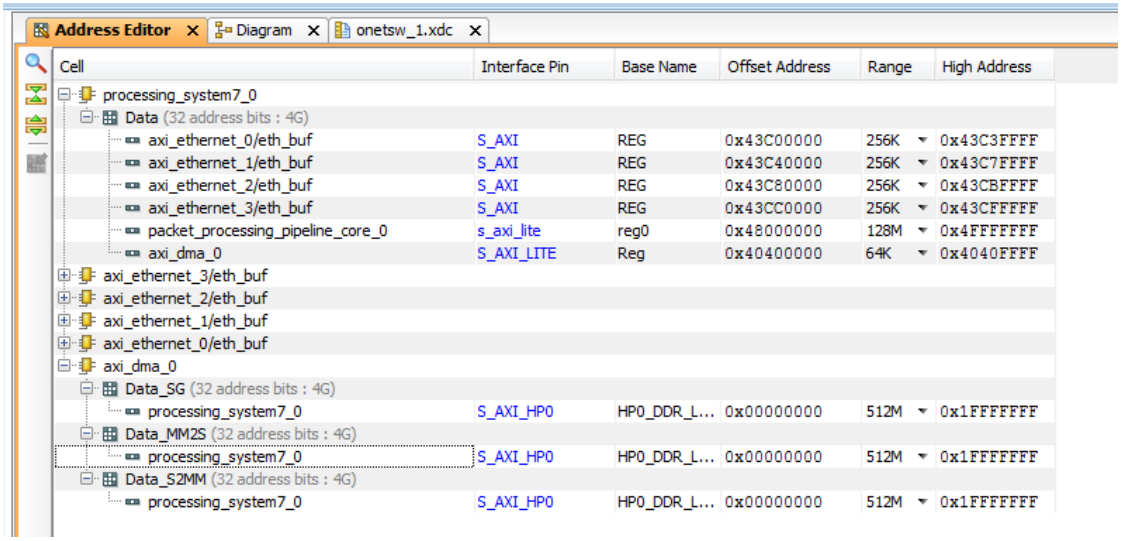

图 3.12 内存地址映射

3.4 本章小结

自此，我们的硬件设计就简单的描述到这里，本章介绍了 OpenFlow 交换机的硬件实现方案，总的来说， OPenFLow 模块需要处理的的数据接口有两种，一种是有软件层接口，种是物理端口的数据匹配接口，对于 host 接口来说，其想要实现的就是对流表状态的查询以及对流表进行配置；对于物理端口接口来说， 其最终目的是寻找转发端口。

OpenFlow 内部使用很多的小模块来进行构建，每个处理模块都会使用一个状态机来完成，每个连接模块之间都使用了请求应答的机制来保证数据的正确传输。整体使用单流水方式来对数据流进行处理，使得数据能够在一定的时间内处理完成，保证了系统的可靠性。

使用 Vivado 设计工具来对系统集成，大大缩短了开发时间，并且 Xilinx 在 Vivado 中也提供了很大的支持，很容易对系统进行构建。使用 ZYNQ 来作为开发平台，使系统真正实现软硬件可编程，同时也简化了很多复杂的总线设计，不像 NetFPGA 一样还用自己设计复杂的 PCI 总线，开发效率大大增加，同时开发难度也大大降低。

第4章 驱动设计与实现

4.1 驱动架构整体概述

设备驱动程序在 Linux 内核中扮演着重要的角色,在 linux 内核源码中很大一部分是 linux 驱动程序，它的作用是为特定硬件定义良好的编程接口，这些接口完全隐藏了设备的工作细节，用户的操作通过一组标准化的调用执行 [20] 。

如图 4.1 所示为系统驱动的架构。它的主要功能有：

1) PHY 的初始化
2) AXI Ethernet 的数据通路
3) OpenFlow 硬件模块的数据通路
4) DMA 中断控制以及数据传输

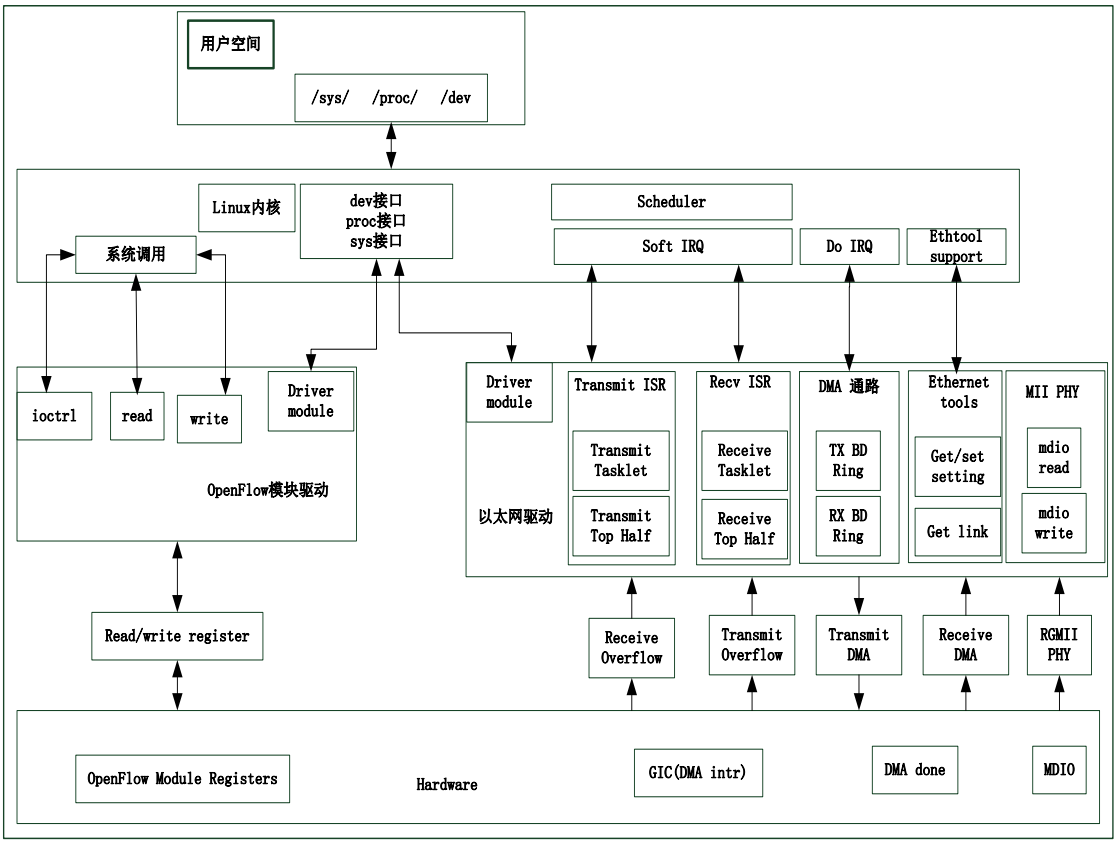

图 4.1 驱动整体架构

4.2 以太网驱动设计

在进行驱动设计之前，首先介绍一下 device tree。从 linux 内核 3.0 版本开始，内核开始使用 device tree 来管理系统硬件设备，在 linux 源码根目录下 arch/arm/boot/dts/目录下有很多可用的 device tree。device tree 是一种描述硬件的数据结构，在系统启动阶段，u-boot 将 devicetree 读入内存的某个区域，并将 devicetree 的地址通过参数传递到内核中，linux 内核通过读取该数据结构来对硬件进行相应的初始化以及操作。Device tree 由一系列的节点极其属性组成，一个节点可能会有子节点，比如 amba 总线是一个节点，所有挂载在 amba 总线上的设备都是他的子节点，如图 4.2 为 PS 端 Ethernet 节点表示。[6]

```
        reg = <0xf8003000 0x1000>;
    } ;
    ps7_ethernet_0: ps7-ethernet@e000b000 {
        #address-cells = <1>;
        #size-cells = <0>;
        clock-names = "ref_clk", "aper_clk";
        clocks = <&clkc 13>, <&clkc 30>;
        compatible = "xlnx,ps7-ethernet-1.00.a";
        interrupt-parent = <&ps7_scugic_0>;
        interrupts = <0 22 4>;
        local-mac-address = [00 0a 35 00 00 00];
        phy-handle = <&phy0>;
        phy-mode = "rgmii-id";
        reg = <0xe000b000 0x1000>;
        xlnx,eth-mode = <0x1>;
        xlnx,has-mdio = <0x1>;
        xlnx,ptp-enet-clock = <111111115>;
        mdio {
            #address-cells = <1>;
            #size-cells = <0>;
            phy0: phy@0 {
                compatible = "marvell,88e1510";
                device_type = "ethernet-phy";
                reg = <0>;
                marvell,reg-init = <0x3 0x10 0xff00 0x1e 0x3 0x11 0xfff0 0xa>;
            } ;
        } ;
    } ;
```

图 4.2 PS Ethernet 设备树节点

本设计采用设备驱动模型进行驱动设计，每一个设备对应一个驱动。驱动从 probe 函数开始执行，进行一些初始化。

1)初始化 Ethernet 结构体

2)通过设备树结构体，读取 DMA 以及 AXI Ethernet 的寄存器基地址，并将其映射为 linux 虚拟内存。

[6] http://www.devicetree.org/Main_Page

3)对设备进行是否支持校验和以及大数据帧传输进行检测

4)设置 MAC 地址

5)注册网络设备

当我们完成网络设备注册后，其会关联两个非常重要的结构体，一个结构体为 net_device_ops 结构体，另外一个是 ethtool_ops 结构体。

net_device_ops 结构体定义如下：

```
static const struct net_device_ops axienet_netdev_ops = {
    .ndo_open    =    axienet_open,
    .ndo_stop    =    axienet_stop,
    .ndo_start_xmit    =    axienet_start_xmit,
    .ndo_change_mtu    =    axienet_change_mtu,
    .ndo_set_mac_address    =    netdev_set_mac_address,
    .ndo_validate_addr    =    eth_validate_addr,
    .ndo_set_rx_mode    =    axienet_set_multicast_list,
    .ndo_do_ioctrl    =    axienet_ioctl,
    .ndo_poll_controller    =    axienet_poll_controller,
};
```

该结构体定义了对网络设备的操作接口。当我们使用 ifconfig ethx up 命令启动 ethx 的时候，驱动程序调用 axienet_open 函数来打开该网络设备，在这个函数中，首先将 PHY 禁止并对 axi ethernet 进行 reset，当 axi ethernet 复位后，在启用 PHY。之后启用 DMA 错误处理 tasklet，打开 DMA 中断。

当我们调用 ifconfig ethx down 施，驱动程序执行 axienet_stop 函数，做一些清除资源的操作。

axienet_start_xmit 函数是一个比较重要的接口。在 linux 内核中，网络数据包包头信息被存放在 sk_buff 结构体中，网络中的各层协议栈通过对该数据结构的操作来完成对数据包的解析。axienet_start_xmit 以 sk_buff 作为参数，首先对 DMA 通道进行选择，最后启动 DMA 传输。当 DMA 传输结束，DMA 发送 TX 中断，此时 axienet_start_xmit_done 函数被执行，该函数为 DMA TX 中断处理函数，主要功能是清除传输 Buffer，将总线控制权转给 CPU。

当 DMA 接受完一帧数据，也会向 CPU 发送接收完成中断，此时接收中断函数 axienet_rx_irq 中断处理函数被执行，其调用 axienet_recv 函数将数据封装到 sk_buff 中，并通知 TCP/IP 协议栈将数据取走。

其他的几个接口都是对 axi ethernet 的一些功能进行设置或者查询，比如对 MUT、MAC 地址等进行设置。

ethtool_ops 结构体定义如下：

```
static struct ethtool_ops    axienet_ethtool_ops = {
        .get_settings    =    axienet_ethtools_get_setting,
        .set_settings    =    axienet_ethtools_set_setting,
        .get_drvinfo    =    axienet_ethtools_get_drvinfo,
        .get_regs_len    =    axienet_ethtools_get_regs_len,
        .get_regs          =    axienet_ethtools_get_regs,
        .get_link          =    axienet_ethtools_get_link,
        .get_pauseparam = axienet_ethtools_get_pauseparam,
        .set_pauseparam = axienet_ethtools_set_pauseparam,
    };
```

该结构体主要是实现 ethtool 工具的接口。比如，当我们在 linux 用户空间中调用 ethtool ethx 命令时，驱动程序将会调用 axienet_ethtools_get_setting 函数来读取相关寄存器的值，并将结果返回到用户空间进行显示。使用 ethtool 主要是对驱进行调试，以及可以通过自己实现这些接口来读取一些相关的硬件信息。

4.3 OpenFlow 模块驱动

OpenFlow 模块驱动相对来说简单得多，它是一个简单的字符设备驱动。它所要完成的功能就是对 OpenFlow 模块相应寄存器进行读写。如表 4.1 所示为对 Table 0 的流表项操作的寄存器说明，其基地址为 0x48001000， 只要使用基地址加上其偏移量就可以对该寄存器进行访问，每个寄存器都是 32 位的，占 4 个字节。

表 4.1 table0 流表项寄存器定义

寄存器名字	偏移量	定义
T0_OPENFLOW_WILDCARD_LOOKUP_ACTION_0_REG	0	Action0
T0_OPENFLOW_WILDCARD_LOOKUP_ACTION_1_REG	4	Action1
……	.....	.....
T0_OPENFLOW_WILDCARD_LOOKUP_ACTION_9_REG	36	Action9
T0_OPENFLOW_WILDCARD_LOOKUP_MASK_0_REG	40	Mask1
T0_OPENFLOW_WILDCARD_LOOKUP_MASK_1_REG	44	Mask2
T0_OPENFLOW_WILDCARD_LOOKUP_CMP_0_REG	48	Cmp1

续表 4.1

寄存器名字	偏移量	定义
T0_OPENFLOW_WILDCARD_LOOKUP_CMP_1_REG	52	Cmp2
T0_OPENFLOW_WILDCARD_LOOKUP_READ_REG	56	通配读
T0_OPENFLOW_WILDCARD_LOOKUP_WRITE_REG	60	通配写

Table1 流表项的寄存器偏移和 Table0 的相同，只是基地址为 0x48002000。

流表维护了两种计数器：第一种为统计信息，包括 hit 的数据包数，miss 的数据包数。第二种为每一条 entry 的统计信息，包括 hit 的数据包数， hit 的字节数。每一个计数器是一个寄存器，少数计数器的位数较多，由两个寄存器组成。通过读取寄存器的值得到计数器的值。表 4.2 描述了 Table0 的统计信息，其基地址为 0x48009000， Table1 的统计信息和 Table0 相同，其基地址为 0x4800A000。

表 4.2 流表统计信息

寄存器名	含义
T0_OPENFLOW_LOOKUP_WILDCARD_MISSES_REG	匹配失败的包数
T0_OPENFLOW_LOOKUP_WILDCARD_HITS_REG	匹配成功的包数
T0_OPENFLOW_WILDCARD_LOOKUP_BYTES_HIT_0_RE	flow entry0 命中字节
T0_OPENFLOW_WILDCARD_LOOKUP_PKTS_HIT_0_REG	Flow entry0 命中包
……	……
T0_OPENFLOW_WILDCARD_LOOKUP_BYTES_HIT_7_RE	flow entry7 命中字节
T0_OPENFLOW_WILDCARD_LOOKUP_PKTS_HIT_7_REG	Flow entry7 命中包

每一条流表项都对应着一个时间戳寄存器，该寄存器记录了流表的插入时间，当流表项命中一个数据包的时候，将命中的时刻写入时间寄存器中，时间寄存器中的所有记录的时间是插入时间和活跃时间的混合值。表 4.3 描述了 Table0 的时间戳寄存器，其基地址为 0x48001000，第一个寄存器的偏移量从 0xC8 开始。Table1 的时间戳寄存器基地址为 0x48002000，偏移量也是从 0xC8 开始。

表 4.3 Table0 时间戳寄存器定义

寄存器名字	定义
T0_OPENFLOW_WILDCARD_LOOKUP_LAST_SEEN_TS_0_REG	Entry0 时间
T0_OPENFLOW_WILDCARD_LOOKUP_LAST_SEEN_TS_1_REG	Entry1 时间
……	……
T0_OPENFLOW_WILDCARD_LOOKUP_LAST_SEEN_TS_7_REG	Entry7 时间
T0_OPENFLOW_LOOKUP_TIMER_REG	Hw curTime

在表 4.3 的最后一条为硬件的当前时间寄存器，在软件层面可以通过读取该寄存器的值与 flow entry 的活跃时间的差值来判断一条流表项是否超时。

系统中共有 4 个 meter，主要用来对流表进行限速。通过在流表中指定数据包转发到某个 meter 实现对流的限速，meter_id=0 是系统保留使用，不限速。每个 meter 有三个寄存器。

1) CTRL_REG:这是一个控制寄存器,现在只是使用了第 0 位和第 1 位,bit[0]为使能位，如果为 1，则该 meter 将按照限速速率执行限速，如果为 0，则不限速，默认值为 0。bit[1]为速率计算控制位，如果为 1，则计算速率的时候把帧间隙、前导码、帧校验都要考虑进去。如果为 0，则不考虑这些东西。

2) TOKEN_INTERVAL_REG 以及 TOKEN_INC_REG:这两个寄存器是这两个是速率参数寄存器,意义是每隔 TOKEN_INTERVAL_REG 个时钟周期发送 TOKEN_INC_REG 个字节。我们需要把设定好的速率转换成这两个参数，写到对应的寄存器中。目前板卡的时钟频率是 75MHz， 假设要把速率定在 r MBps，那么 r/75 得到一个分数，这个分数的分母和分子分别就是 TOKEN_INTERVAL_REG 和 TOKEN_INC_REG 的值。例如假如要把速率限定在 5Mbps，首先算出来 5Mbps=5/8 MBps，然后 5/8/75 等于 1/(8*15) = 1/120 ， 所 以 TOKEN_INTERVAL _REG=120 ，TOKEN_INC_REG=1。一个 meter 只能对一条流限速，多条需要限速的流须分配多个 meter。

4 个 Meter 的寄存器定义如下：

```
//rate_limiter0 基地址为 0x48000100
```

```
#define RATE_LIMIT_0_CTRL_REG                0
#define RATE_LIMIT_0_TOKEN_INTERVAL_REG      4
#define RATE_LIMIT_0_TOKEN_INC_REG           8
//rate_limiter1 基地址为 0x48000200
#define RATE_LIMIT_1_CTRL_REG                0
#define RATE_LIMIT_1_TOKEN_INTERVAL_REG      4
#define RATE_LIMIT_1_TOKEN_INC_REG           8
//rate_limiter2 基地址为 0x48000300
#define RATE_LIMIT_2_CTRL_REG                0
#define RATE_LIMIT_2_TOKEN_INTERVAL_REG      4
#define RATE_LIMIT_2_TOKEN_INC_REG           8
//rate_limiter3 基地址为 0x48000400
#define RATE_LIMIT_3_CTRL_REG                0
#define RATE_LIMIT_3_TOKEN_INTERVAL_REG      4
#define RATE_LIMIT_3_TOKEN_INC_REG           8
```

以上对 OPenFlow 模块的寄存器进行了说明，驱动程序通过对这些寄存器读写来来实现对硬件的控制。在上一章的硬件实现模块中，我们也多次提到这一点，当驱动程序通过 I/O 函数对寄存器进行读写操作后，host_info 模块的状态机会自动的与流表控制模块进行交互， 从而实现对流表的配置或者状态查询功能。

本设计在内核空间中主要是实现了 3 个用户接口，read、write 以及 ioctl，其定义如下。

```
static struct file_operattions ofnet_fops = {
    .owner = THIS_MODULE,
    .read    = ofnet_read,
    .write    = ofnet_write,
    .unlocked_ioctl = ofnet_ioctl,
    .open = ofnet_open,
    .release = ofnet_release,
};
```

其中 ofnet_read 主要是对 OpenFlow 模块寄存器进行读操作，ofnet_write 主要都是对寄存器进行写操作，ioctl 主要实现 OpenFlow 模块的一些状态查询操作。不过它们最终都是对模块的寄存器进行读写，定义如下：

```
#define    ofnet_readreg(offset)    readl(offset)
#define    ofnet_writereg(offset, val)    writel(val, offset)
```

上面的实现为标准的字符型设备驱动实现方式，但是，对于系统需要不断的修改的情况时，使用这个方式就会变得很麻烦了。故而，我们使用了直接在用户空间来对硬件地址直接操作的方式来驱动 OpenFlow 模块。/dev/mem 为操作系统向用户空开放的整个硬件物理内存的全映象，并在其内核驱动中实现了 mmap 接口，用户可以通过对它在用户空间中 mmap 操作来实现对物理内存的访问。通过如下函数在用户空间将所需的物理地址映射为虚拟地址。在实现内存映射后，对 map_base 的操作即为对硬件地址的操作。

```
int getMem(unsigned long phyaddr,    unsigned char **map_base;)
{
    int fd;
    int pagesize = getpagesize();
    fd = open("/mem/dev/", O_RDWR | O_SYNC);
   *map_base = mmap(NULL, pagesize, PROT_READ|PROT_WRITE,
                        MAP_SHARED, fd, phyaddr & 0xfffff000);
    if(fd == -1 || !(*map_base))
        return -1;
    return 0;
}
```

在上述设计中，最重要的是要考虑内存对齐的问题，所以映射的内存空间必须是操作系统的 pagesize 的倍数。使用该方法必须是 root 用户权限，而且不能够使用中断，最后使用 munmap(map_base, pagesize)释放映射内存。

4.4 本章小结

本章对设备驱动进行了描述。一个是基本的以太网驱动，主要完成了网络设备的注册，DMA 的相关控制，底层 Ethernet 以及 PHY 的数据通路设置等工作。以太网驱动主要是将底层网络接口抽象为操作系统用户空间中的软件接口，在软件层面的 OpenFlow 软表就是通过对这些软件接口的嗅探、监听等来获得原始数据包。对于 DMA 中断的处理是完成以太网驱动的重点。

另一部分是对 OpenFlow 模块驱动的构建。首先对 OpenFLow 模块的相关寄存器的功能以及物理地址进行介绍，我们在对 OpenFlow 模块进行操作的过程中就是对这些寄存器进行读写来实现对模块的控制。我们所描述的寄存器只对其偏移量进行定义，对其进行读写的过程中，在 I/O 函数中只要加上其基地址即可。

最后我们还介绍了如何在用户空间中进行用户驱动程序的编写，通过对/dev/mem 的 mmap 操作来获得所需物理地址的虚拟地址空间。

在编写驱动程序的时候，我们尽量的只是提供机制，而没有实现其策略，策略的实现应该交给应用程序来实现。比如，在 OpenFlow 模块驱动中，我们只提供了读写寄存器的接口，至于如何对寄存器进行读写，如何来来读取交换机的状态等功能并没有实现。这些策略我们将在下一张软件应用程序的构建中间进行描述。

第5章 OpenFlow 软件流表

5.1 应用程序设计概述

前面我们已经对 OpenFlow 硬件流表的实现、基本的网络通路、驱动程序接口等进行了描述，但是要让整个系统正常的运行起来，用户应用程序的实现必不可少。在用户应用程序中，我们继续构建 OpenFlow 软件流表，作为硬件流表的一个补充，软件流表完全按照 OpenFlow1.3 规范来进行构建。从整体上来说，我们可以用如图 5.1 所示的流程来描述数据包在整个系统中流动过程。

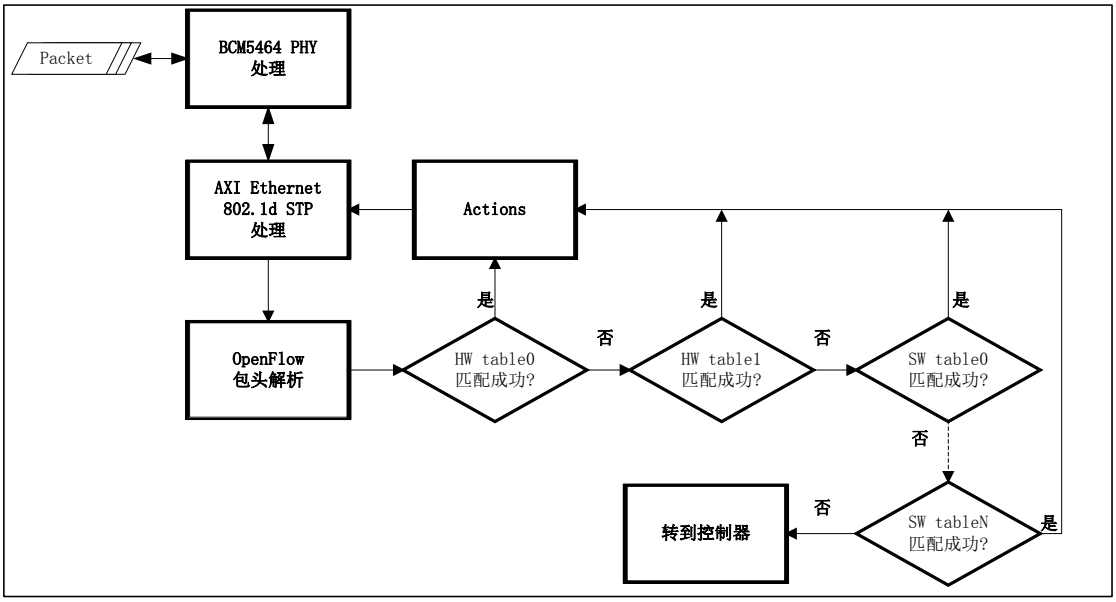

图 5.1 数据包整体流表匹配流程

在用户程序端，我们所要实现的功能就是为用户提供统一的资源视图以及一些控制策略。这里的应用程序仍然是在提供接口，用户通过这些接口可以轻松的来对硬件层或者软件层的抽象进行管理，从而实现自己的 SDN 算法。简单来说就是用户不需要知道底层的实现细节，就能够对交换机进行操作，这些操作包括对交换机进行配置，读取交换机的状态等。同时，用户看到流表就是一些串联起来的流表，而并不用关心对他们的读写是如何实现的。

在软件层，本课题使用了由爱立信巴西实验室以及 CPqD 共同维护的开源项

目，这个项目主要由 OpenFlow 软件交换机、NOX 控制器、wireshark 插件、以及一些测试用例组成。该项目最初版本基于 OpenFlow1.1 规范进行开发，现在已经支持到了 OpenFlow1.3。下面我们主要对 OPenFlow 交换机以及 NOX 控制器进行介绍 [21] 。

5.2 OpenFlow 软件交换机

目前，OpenFlow 软件交换机主要有两种版本。

1) 用户空间版本：本项目的 OpenFlow 软件交换机就是在用户空间中进行构建的。它的特点是稳定、可靠、操作简单，但是缺点是速度慢，主要用于算法研究和软件研究。
2) Linux 用户空间-内核空间版本：这种版本速度相对较快，但是修改和操作比较麻烦，需要有内核编程和底层驱动的知识。比如 Open vSwitch 就是这种模式。

本项目的 OpenFlow 软件交换机在 NetBee 的基础上进行构建。NetBee 是一个用于对种类型数据包处理的库，包括对数据包进行嗅探、过滤、解码、以及流分类等。它通过一个 NetVM 虚拟机来获取网络接口的数据包，并在上层提供一些 API 对数据包进行处理。所以，OpenFlow 软件交换机获得的数据包并没有经过 TCP/IP 协议栈，而是直接有 NetBee 传过来的原始数据包。

本项目使用的 OpenFlow 软件交换机在用户空间中构建，它主要有四个部分组成。

1) ofdatapath: 交换机核心实现部分，主要包括流表的数据通路、流表匹配、组表管理、action 处理等
2) ofprotocol: 实现同外部控制器的连接
3) oflib: 提供一些标准化 API，主要实现对流表的操作接口
4) dpctl: 这里主要是提供从控制台来对交换机进行配置的命令工具。

由于要加入对硬件流表的支持，所以我们需要在 oflib 中修改相应的代码，主要是对软件 Table0 以及 Table1 的相应接口进行修改，使得当上层对 Table0 以及 Table1 进行操作的时候能能够自动的转到对硬件中的 Table0 以及 Table1 进行操作。根据第三章中的描述，我们得到下面两个数据结构。

Table0 的流表项的匹配域和相应的 mask 定义为:

```
struct    nf2_of_entry_t0    {
```

```
    uint16_t    src_port;
    uint8_t    eth_dist[6];
};
```

该流表主要对 L2 进行匹配，只要源端口和目的 MAC 地址相同，则执行相应的 Action。

Table1 的匹配域和相应的 mask 定义为：

```
struct    nf2_of_entry_t1    {
    uint32_t    ip_dst;
    uint16_t    transp_src;
    uint16_t    transp_dst;
};
```

Table1 主要对 L3 以上进行匹配，主要匹配 TCP 目的端口以及源端口以及目的 IP 地址。

两个表的 Action 定义相同，一共 32 字节，如结构体 nf2_of_action 所示：

```
#pragma pack(1) //一字节对齐
struct    nf2_of_action {
    uint16_t    forward_bitmask;    //转发 mask
    uint16_t    nf2_action_flag;    //Action 标记
    uint16_t    vlan_id;
    uint8_t    vlan_pcp;
    uint8_t    eth_src[6];    //源 MAC 地址
    uint8_t    eth_dst[6];    //目的 MAC 地址
    uint32_t    ip_src;    //源 IP 地址
    uint32_t    ip_dst;    //目的 IP 地址
    uint8_t    ip_tos;
    uint16_t    transp_src;
    uint16_t    transp_dst;
    uint8_t    meter_id;
    uint8_t    next_table_id;    //转发到下一个表的表 id
    uint8_t    reserved[6]; //保留，48 字节
};
```

forward_bitmask 的作用是通过掩码的形式来指定数据包的转发端口，一共 16 位，只用了其中低 8 位。系统中一共有 8 个端口可以提供转发，其中包括 4 个物

理端口，和 4 个虚拟 CPU 端口，物理端口主要用在硬件流表中，CPU 端口和硬件端口对应，主要用在软件流表中，其定义以及其对应 mask 取值如表 5.1 所示。

表 5.1 port 定义及取值

端口	端口值	mask value
Phy port 0	0	0x01
Phy port 1	2	0x04
Phy port 2	4	0x10
Phy port 3	6	0x40
CUP port 0	1	0x02
CUP port 1	3	0x08
CUP port 2	5	0x20
CUP port 3	7	0x80

action_flag 共 16 位，每一位代表一种 action，如果需要执行某种 action，只需要将该 action 所对应的掩码置 1，如表 5.2 所示。

表 5.2 action_flag 标志位

bit 位	action	mask value
0	OUTPUT	0x0001
1	SET_VLAN_VID	0x0002
3	SET_VLAN_PCP	0x0004
4	STRIP_VLAN	0x0008
5	SET_DL_SRC	0x0010
6	SET_DL_DST	0x0020
7	SET_NW_SRC	0x0040
8	SET_NW_DST	0x0080
9	SET_NW_TOS	0x0100
10	SET_TP_SRC	0x0400
11	GOTO_TABLE	0x0800
12	METER	0x1000

meter_id 可取值为 0、1、2、3。0 表示 meter 是系统保留，不限速，其他表示启用相应的 meter 来进行限速处理。

如果需要精确匹配某个域， 可以将其掩码全设为 0，如果只需要精确匹配部分比特，那么把这部分比特对应的掩码设置为 0 即可，否则设置对应的比特为 1。比如 table1 可以匹配 ip_dst、transp_src、transp_dst，现在构造一条 entry，对 ip_dst 为 1.2.3.*, transp_dst=80 的数据包，则可这样设定。

```
nf2_of_entry_t1    flow_entry_t1,    flow_mask_t1;
flow_entry_t1.entry.ip_dst = 0x01020300;
flow_mask_t1.entry.ip_dst = 0x000000FF;
flow_entry_t1.entry.transp_src = 0;
flow_mask_t1.entry.transp_src = 0xFFFF;
flow_entry_t1.entry.transp_dst = 80;
flow_mask_t1.entry.transp_dst = 0;
```

要对硬件流表进行配置，首先得初始化这三个结构体，然后对这个三个结构体进行相应项进行装填，最后将三个结构体的内容写入硬件流表寄存器中，如下所示为配置 Table0 的示例。

```
nf2_of_entry_wrap_t0    flow_entry_t0;
nf2_of_mask_wrap_t0    flow_mask_t0;
nf2_of_action_wrap    flow_action_t0;
//初始化结构体
nf2_flow_entry_init_t0 (&flow_entry_t0, &flow_mask_t0, &flow_action_t0);
flow_entry_t0.entry.src_port = 0;
//Exact match src port: set mask to all zeros
flow_mask_t0.entry.src_port= 0x0000;
//设置修改 IP action
flow_action_t0.action.nf2_action_flag |= 0x0040; //set modify ip src action mask
flow_action_t0.action.ip_src = 0x01010101; // new ip src
flow_action_t0.action.nf2_action_flag |= 0x1000; //set goto actions mask
flow_action_t0.action.next_table_id = 0x1; //goto table1
//写入流表寄存器
nf2_install_flow_entry_t0 (&nf2,31, &flow_entry_t0,&flow_mask_t0,
                           &flow_action_t0);
```

nf2_install_flow_entry_t0 函数为向 Table0 写入相关寄存器的函数，第一个参

数为设备名称，第二个参数为写入的流表项 ID，第三个参数为流表匹配域结构体，第四个参数为流表匹配域掩码结构体，最后一个参数为 action 结构体。

这里我们只对如何在软件交换机中加入硬件流表的支持做了描述。至于如何启动交换机、如何使用 dpctl 命令来对交换机进行配置、以及如何与控制器进行交互等都将在下一章中进行描述。

5.3 NOX 控制器

控制器运行在 PC 或服务器中，它通过一个安全通道与 OpenFlow 交换机进行通信。在本设计中，OpenFlow 交换机使用 PS 端的 Ethernet 端口来同运行在 PC 机的控制器进行连接，两者之间使用 TCP/IP 协议进行消息的传递。

NOX 是第一个 OpenFlow 控制器，于 2008 年开始研究，它最初只支持拓扑发现、交换学习、全网交换等功能，本项目的 nox13oflib 也是基于 NOX 的一个开源项目，其基本组件如图 5.2 所示 [22] 。

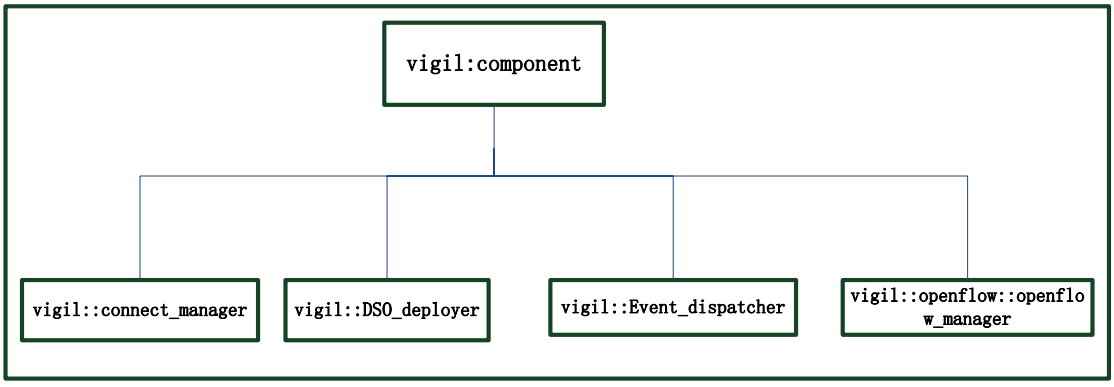

图 5.2 NOX 组件

connect_manager 类主要负责连接管理，它会一直监听，直到有 OpenFlow 交换机向其请求连接；DSO_deployer 类的主要作用是对 NOX 数据通路的部署进行管理；Event_dispatcher 类的主要作用是对各项事件进行统一的调度管理；OpenFlow_manager 类主要处理同 OpenFlow 交换机之间的通信。

目前，nox13oflib 项目仍然在开发完善中，只支持 C++语言，还没有完全支持 Of1.3 规范，暂且还不支持多通道连接。在 OPenFlow1.3 规范中，默认的 table-miss 选项的 action 是直接将数据包丢弃，如果需要将默认 action 设置为转发到控制器，我们还需要定义个优先级相对较高的流表项来完成这一功能。

控制器不仅向底层 OpenFlow 交换机提供通信机制，它还向上层应用提供编

程接口，用户可以通过这些接口来实现一些图形化的管理界面，从而能够简单方便的对 OpenFlow 交换机进行操作。

5.4 本章小结

本章描述了如何在 linux 用户空间构建 OpenFlow 软件交换机。软件交换机弥补了硬件交换机的不足，并建立了同控制器之间的连接。对 OpenFlow 软件交换机项目移植过程中遇到了一些麻烦，尤其是对在对 OpenFlow 软件交换机工程进行交叉编译的时候花费了大量时间，最后我们是通过 QEMU 来构建了一个 ARM 虚拟机，然后直接将项目移植到虚拟机上进行编译。

本项目使用软硬件协同设计来构建软硬件流表，在软件端通过修改相应的接口来实现软硬件流表逻辑上的连续，使得用户所看到视图是一个完整的流表，用户可以像平时操作软件流表一样来对硬件流表进行控制。

由于硬件成本和实现上的一些限制，我们不可能将所有的流表都移植到硬件上，并且在硬件端实现一些用户接口和算法是不现实的，所以软件流表是必不可少的，通过软件层面的定义，我们可以很容易的实现同控制器端的连接以及通信。

在本章的最后我们简单介绍了一下 NOX 控制器，在 OpenFlow 网络甚至是 SDN 中控制器都是必须的，其扮演着一个网络控制层面接口的角色，它不仅同底层 OpenFlow 交换机进行连接，同时还向上层应用提供编程接口，从而使得上层应用的实现变得更加的简单。

第6章 系统测试

6.1 系统概述

本系统采用的开发板为 ZedBoard，ZYNQ 型号为 XC7Z020-1CLG484。通过 ZedBoard 上的 FPGA Mezzanine Card (FMC) 接口扩展了 4 个千兆网口，如图 6-1 所示为交换机的实物图。

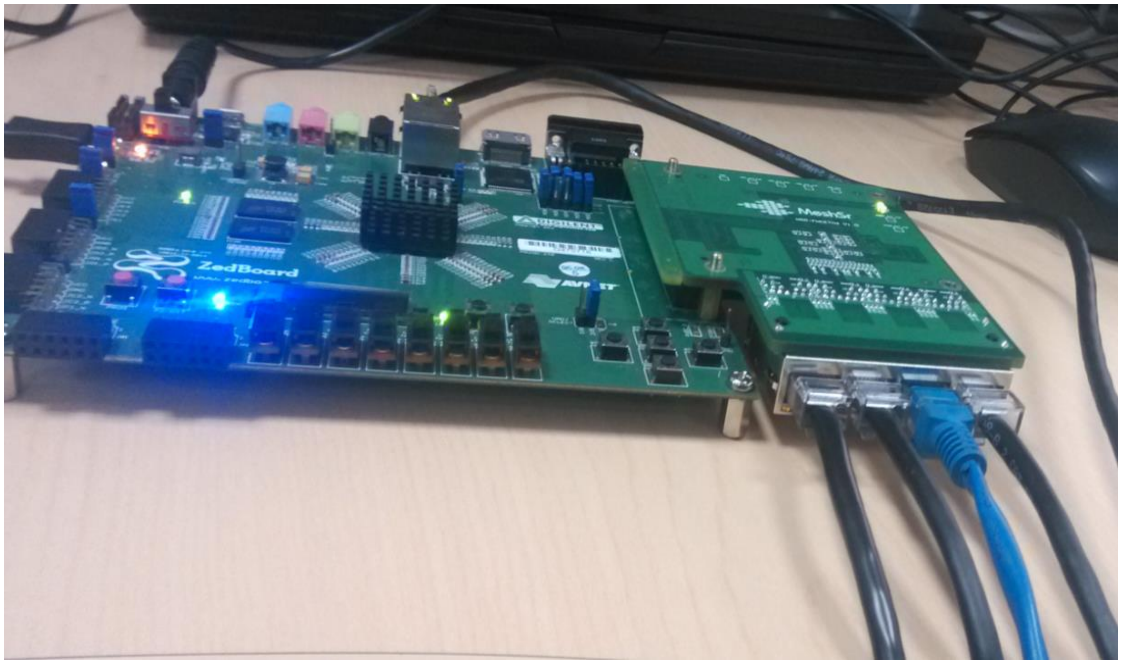

图 6.1 交换机实际效果图

这里，FMC 通过 Low-Pin-Count (LPC) 的方式对 ZYNQ 和 BCM5464 进行桥接，实现 RGMII 协议和 MDIO 管理接口。

在进行系统测试的之前，首先得将整个系统启动起来。如图 6.2 所示为系统启动所需的几个文件。其中 boot.bin 文件为系统启动镜像，它主要由 ZYNQ 第一级启动程序 fsbl.elf、系统硬件比特流、以及 linux 系统启动程序 u-boot 三个文件在 SDK 中使用 bootgen 命令生成。uImage 文件为 linux 内核镜像文件，我们所需的驱动程序都已经集成在其中；devicetree.dtb 为系统的设备树文件。

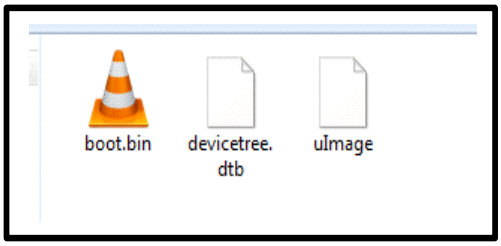

图 6.2 系统启动文件

Linux 文件系统使用 EXT4 的形式存储在 SD 卡的 EXT 分区，将图 6.2 所示的启动文件放入 SD 卡的 FAT 分区，并将 SD 卡插入 ZedBoard，打开电源即可完成系统的启动。当系统启动以后，使用如下命令启动软件交换机。

```
$ifconfig eth1 up
$ifconfig eth2 up
$ifconfig eth3 up
$ifconfig eth4 up
$ofdatapath --datapath-id=000a35000001  --interfaces=eth1,eth2，eth3,eth4  \
 ptcp:6632  --no-slicing -D
$ofprotocol  tcp:127.0.0.1:6632 tcp:192.168.4.160:6633  --no-slicing  -D
```

在启动软件交换机的时候，首先使用 ifconfig 命令将需要监听的网口启动起来，然后才能使用 ofdatapath 命令来启动交换机。ofdatapath 的第一个参数为 datapath 的 id, 用来表示这一个 datapath,第二个参数为需要监听的网络端口号，这里我们只启用了两个端口进行测试，第三个参数为 passive TCP, 它会开启一个 TCP 端口来对相应的网口进行监听 –D 表示让该程序在后台运行。ofprotocol 命令用于启动同控制器之间的连接，第一个 tcp 为本机的监听网口以及端口，第二个 TCP 为控制器的监听端口和网口，两者之间使用 TCP 来作为消息的载体。[7]

在软件交换机中提供了一个 dpctl 工具来对交换机进行操作，我们可以通过这个命令来对交换机进行一些相应的配置。对于硬件流表的配置使用 forward_pkt 工具来进行，forward_pkt 是我们自己实现的一个配置硬件流表的小工具， 主要对相应流表寄存器进行写。下面对 dpctl 进行介绍。

1) 对给定的流表进行状态检测。

```
 $ dpctl  tcp:127.0.0.1:6632  stats-flow  table=0
```

2) 添加一条流表项到给定的流表中,具体参数参考--help。

[7] https://github.com/CPqD/ofsoftswitch13

```
$dpctl tcp:127.0.0.1:6632 flow-mod cmd=add，table=0, prio=1 in_port=1 \
  apply:output=2
```

上面这条命令的意思是添加一条流表项到 table0 中，优先级为 1，输入端口为 1，输出端口为 2。那么对这条流表项进行匹配时，只要是从端口 1 进入的数据包，都将从端口 2 进行转发。

3) 添加一个限速表。

```
$dpctl tcp:127.0.0.1:6632 meter-mod cmd=add,meter=1 drop:rate=50
```

4) 将所要限速的流添加到 meter 中

```
$dpctl tcp:127.0.0.1:6632 flow-mod table=0,cmd=add in_port=1 meter:1
```

这样，从端口 1 进来的数据流将会被限制速率为 50。

本系统使用 wireshark 来进行抓包，如果想要抓取 Of 协议包头，在这之前，我们必须得进行相关插件的安装。具体安装过程如下所示， 在安装完成后，在 wireshark 的 filter 一栏输入 of 就能对 OpenFlow 协议进行监听。[8]

```
$sudo apt-get install wireshark scons
$git clone https://github.com/CPqD/ofdissector
$cd src && export WIRSHARK=/usr/include/wireshark
$scons install
```

本文主要对交换机的功能和性能进行测试。功能测试主要针对交换机能实现某些功能进行测试，包括系统启动、控制器连接、端口直接转发、L2 转发、L3 及以上转发、速率限制、以及流量控制等。性能测试主要包括，硬件流表性能、端口自适应能力、全网端口限速能力、二层汇聚转发性能、超长帧检测、异常帧检测等，这里我们主要挑几个比较重要的进行讲解。

6.2 端口强制/自适应能力测试

以对 DUT (Device Under Test) 的 1 号以太网端口 eth1 进行测试为例，其他端口测试均一致并省略。首先确保 DUT 以及 TG 都为 1000Mbps，可以使用 ethtool 工具查看，如图 6.3 所示。

使用 ethtool -s 命令手动降低 TG 的 eth0 网口速度为 100Mbps,同时观察 DUT 的对应网口 eth1,发现其检测到了 TG 速率的变化并将链路工作速率自适应到 100Mbps,如图 6.4 所示。

[8] https://github.com/CPqD/ofsoftswitch13/wiki/OpenFlow-1.3-Tutorial

OPenFLow Switch(DUT)

```
root@zynq:~# ifconfig eth1
eth1      Link encap:Ethernet  HWaddr 00:0A:35:00:00:01
          UP BROADCAST RUNNING  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@zynq:~# ethtool eth1
Settings for eth1:
        Supported ports: [ TP AUI BNC MII FIBRE ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Supported pause frame use: No
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Advertised pause frame use: No
        Advertised auto-negotiation: Yes
        Speed: 1000Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 3
        Transceiver: external
        Auto-negotiation: on
        Link detected: yes
```

PC(TG)

```
[root@sigma sigma]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr D4:3D:7E:B7:56:19
          inet addr:192.168.5.1  Bcast:192.168.5.255  Mask:255.255.255.0
          inet6 addr: fe80::d63d:7eff:feb7:5619/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:9870397 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2453899 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2628462534 (2.4 GiB)  TX bytes:732863264 (698.9 MiB)

[root@sigma sigma]# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Supported pause frame use: No
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Advertised pause frame use: Symmetric Receive-only
        Advertised auto-negotiation: Yes
        Link partner advertised link modes:  10baseT/Half 10baseT/Full
                                             100baseT/Half 100baseT/Full
                                             1000baseT/Half 1000baseT/Full
        Link partner advertised pause frame use: No
        Link partner advertised auto-negotiation: Yes
        Speed: 1000Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 0
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: g
        Current message level: 0x00000033 (51)
                               drv probe ifdown ifup
        Link detected: yes
```

图 6.3 产看端口链路状态

TG

```
[root@sigma sigma]# ethtool -s eth0 speed 100 autoneg on duplex full
[root@sigma sigma]# ethtool eth0
Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Supported pause frame use: No
        Supports auto-negotiation: Yes
        Advertised link modes:  100baseT/Full
        Advertised pause frame use: Symmetric Receive-only
        Advertised auto-negotiation: Yes
        Link partner advertised link modes:  10baseT/Half 10baseT/Full
                                             100baseT/Half 100baseT/Full
        Link partner advertised pause frame use: No
        Link partner advertised auto-negotiation: Yes
        Speed: 100Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 0
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: g
        Current message level: 0x00000033 (51)
                               drv probe ifdown ifup
        Link detected: yes
```

DUT

```
root@zynq:~# libphy: 43c00000:03 - Link is Down
libphy: 43c00000:03 - Link is Up - 100/Full

root@zynq:~# ethtool eth1
Settings for eth1:
        Supported ports: [ TP AUI BNC MII FIBRE ]
        Supported link modes:   10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Supported pause frame use: No
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
                                1000baseT/Half 1000baseT/Full
        Advertised pause frame use: No
        Advertised auto-negotiation: Yes
        Speed: 100Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 3
        Transceiver: external
        Auto-negotiation: on
        Link detected: yes
```

图 6.4 端口自适应

6.3 端口转发以及速率限制测试

将两台测试主机 TG1 和 TG2 分别连接到 DUT 的 eth1,eth2 网口上，先通过串口连接到 DUT 并启动 OpenFLow 控制监听端口，如图 6.5 所示。其中，datapath-id 一般指定为 eth0 的 mac 地址，这里只启用了两个网络接口 eth1 以及 eth2，使用

6632 端口来做为 TCP 被动监听端口，--no-slicing 指定了无切片，-D 表示运行在后台运行，当配置成功，交换机将对两个端口进行监听，当端口有数据包进入时，系会自动进行流表匹配，并根据输出端口直接转发。

这里的端口转发和数据包的具体协议无关，系统也不会对包进行解析，只是做了一个纯粹的端口转发。最后使用 iperf 工具来对系统进行测试。

```
root@zynq:~# ofdatapath --datapath-id=000a35000001 --interfaces=eth1,eth2 \
> ptcp:6632 --no-slicing -D
device eth1 entered promiscuous mode
net eth1: Promiscuous mode enabled.
net eth1: Promiscuous mode enabled.
net eth1: Promiscuous mode enabled.
device eth2 entered promiscuous mode
net eth2: Promiscuous mode enabled.
net eth2: Promiscuous mode enabled.
net eth2: Promiscuous mode enabled.
device tap0 entered promiscuous mode
```

图 6.5 启动 OpenFlow 控制监听端口

通过 dpctl 来配置一个 0 号限速表，限制速率为 1Mbps,如图 6.6 所示。

```
root@zynq:~# dpctl tcp:127.0.0.1:6632 meter-mod cmd=add,flags=1,meter=0 \
> drop:rate=1000

SENDING:
meter_mod{cmd="add", flags="0x1"", meter_id="0"", bands=[{type = drop, rate="1000", burst_size="0"}]}


OK.

root@zynq:~# dpctl tcp:127.0.0.1:6632 meter-config

SENDING:
stat_req{type="mconf", flags="0x0"{meter_id= ffffffff"}


RECEIVED:
stat_repl{type="mconf", flags="0x0", stats=[{meter= 0"", flags="1", bands=[{type = drop, rate="1000", burst_size="0"}]}]}
```

图 6.6 配置限速表

配置两条端口转发流表项，将来自 1 号网口的数据包转发到 2 号网口，反之从 2 号口的数据包转发到 1 号网口，如图 6.7 所示。

```
root@zynq:~# dpctl tcp:127.0.0.1:6632 flow-mod cmd=add,table=0,prio=1 \
> in_port=1 apply:output=2 meter:0

SENDING:
flow_mod{table="0", cmd="add", cookie="0x0", mask="0x0", idle="0", hard=


OK.

root@zynq:~# dpctl tcp:127.0.0.1:6632 flow-mod cmd=add,table=0,prio=2 \
> in_port=2 apply:output=1 meter:0

SENDING:
flow_mod{table="0", cmd="add", cookie="0x0", mask="0x0", idle="0", hard=


OK.
```

图 6.7 配置转发表

设置两台主机的地址。TG1 为 192.168.5.2/24,TG2 为 192.168.5.1/24，基于上面的配置，eth1 与 eth2 能够互相 ping，如图 6.8 所示。

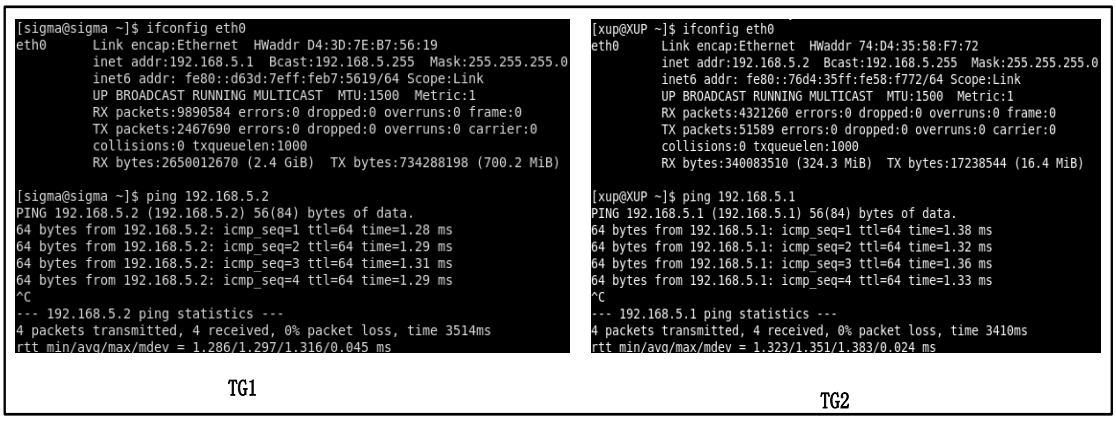

图 6.8 TG 互 ping 成功

使用 iperf 工具来对数据包进行测试。先使用 4KB 的短包来对交换机进行测试。如图 6.9 所示。由于前面将速率限制为 1Mbps，在 4KB 短包测试下实际的 TCP 带宽为 600Kbps。

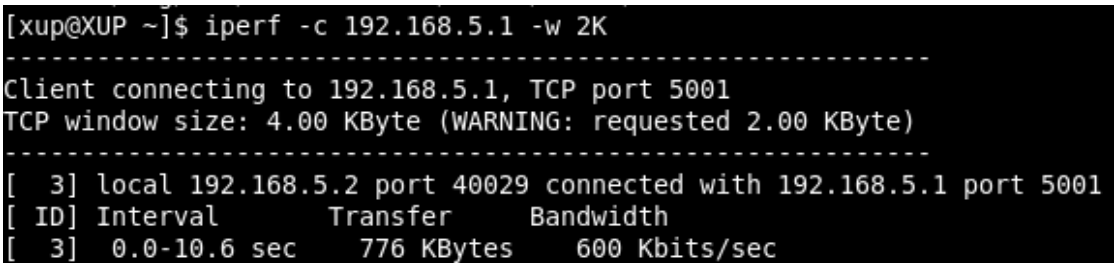

图 6.9 4KB 短包限速转发

在使用 244KB 的长包进行测试，如图 6.10 所示。可见测试结果为 983Kbps，考虑到 TCP 带宽少了 L2 以及 L3 包头，因此限速速率会小于理论值。但验证了 DUT 的端口可以按配置进行限速。

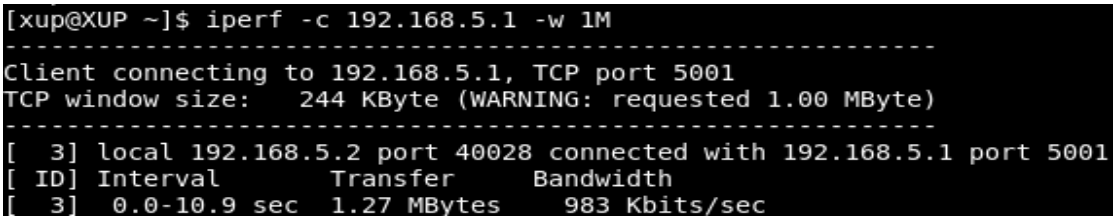

图 6.10 长包限速转发测试

6.4 硬件流表 L2 转发及性能测试

使用如下命令对硬件 Table0 进行配置，根据第四章的描述，硬件 Table0 只有两个选项，第一个选项为网络包 inport 端口，第二个选项为目的 MAC 地址。使用 SmartBits 对交换机进行测试，流表项内容为 in_port=0，mac_dst = 06:05:04:03:02:01，发送流量为 64 字节 (加上 CRC) ，链路利用率为 100%，转换之后为 1488095pps，pps 为每秒发送数据包数，如图 6.11 所示。

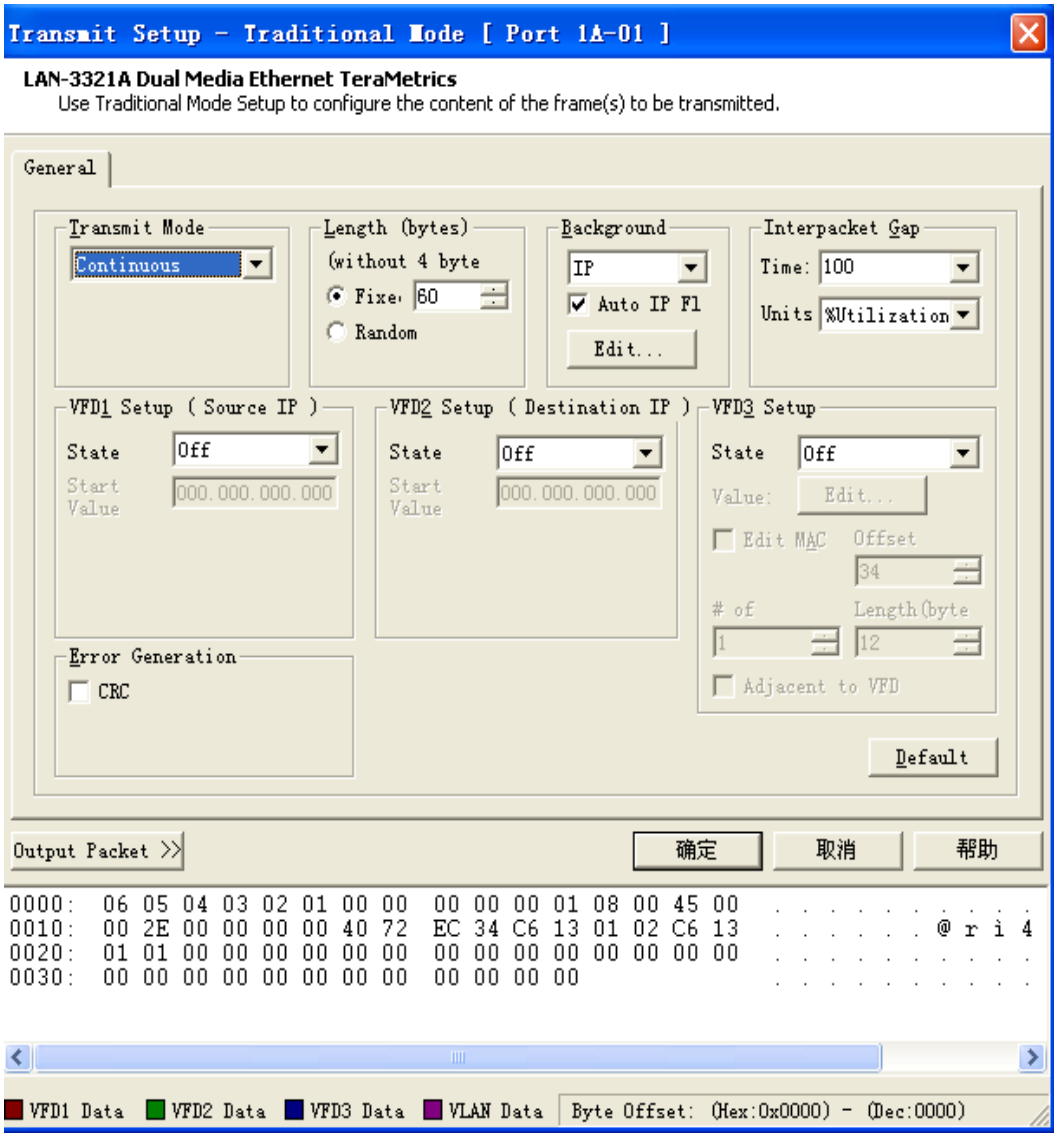

图 6.11 SmartBits 发送数据包

如图 6.12 为测试结果，上侧为发送流量统计，发送速率为 1488094pps。下侧为接受流量统计，接受速率为 1488094pps，字节速率稍有不同。

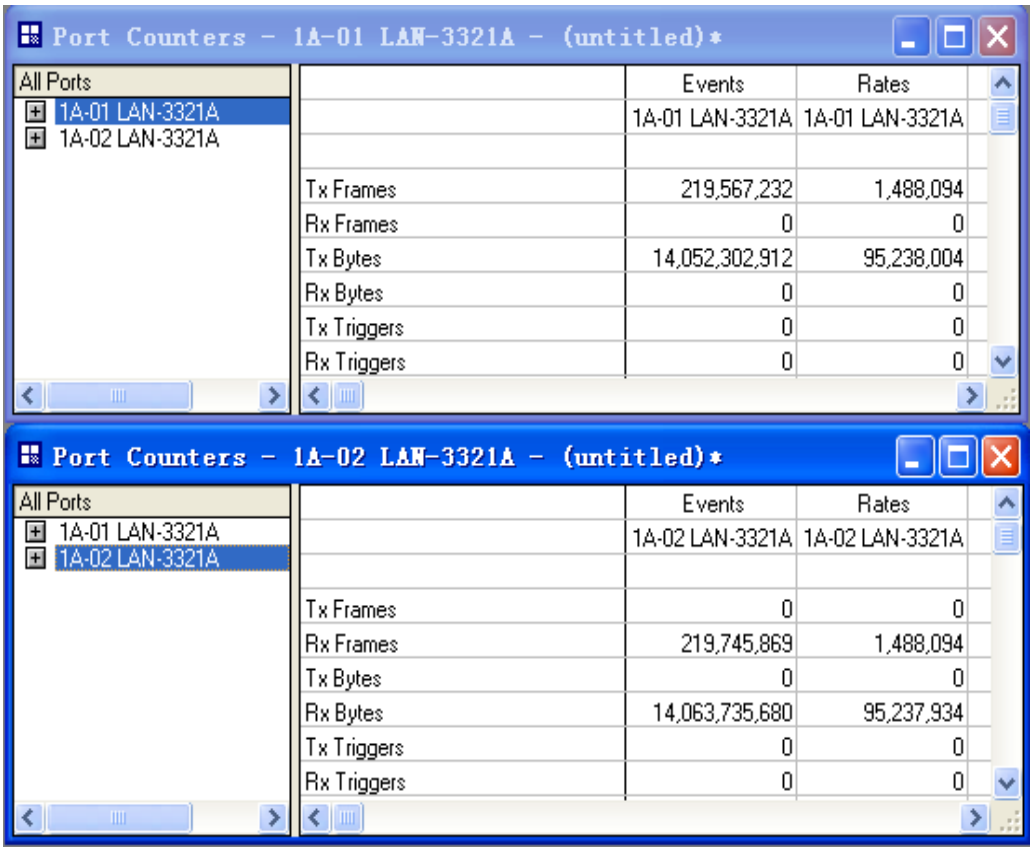

图 6.12 L2 转发测试结果

6.5 硬件流表 L3 转发及性能测试

硬件流表 Table1 的 flow_entry 配置如下：

```
//192.168.1.1/24
flow_entry_t1.entry.ip_dst = (((((192 << 8) + 168) << 8) + 1) << 8) + 1;
//精确匹配
flow_mask_t1.entry.ip_dst= 0x000000ff;
flow_action_t1.action.nf2_action_flag |= 0x0001;
flow_action_t1.action.forward_bitmask = 0x04;
```

forward_bitmask=0x04 表明只要 ip_dst=192.168.1.*，则从 port1 转发。使用 Smartbits 测试发送流量，如图 6.13 所示。

发送字节：64（加上 CRC）

发送速率：1488095pps

发送流量的 IP_dst 为 192.168.1.2

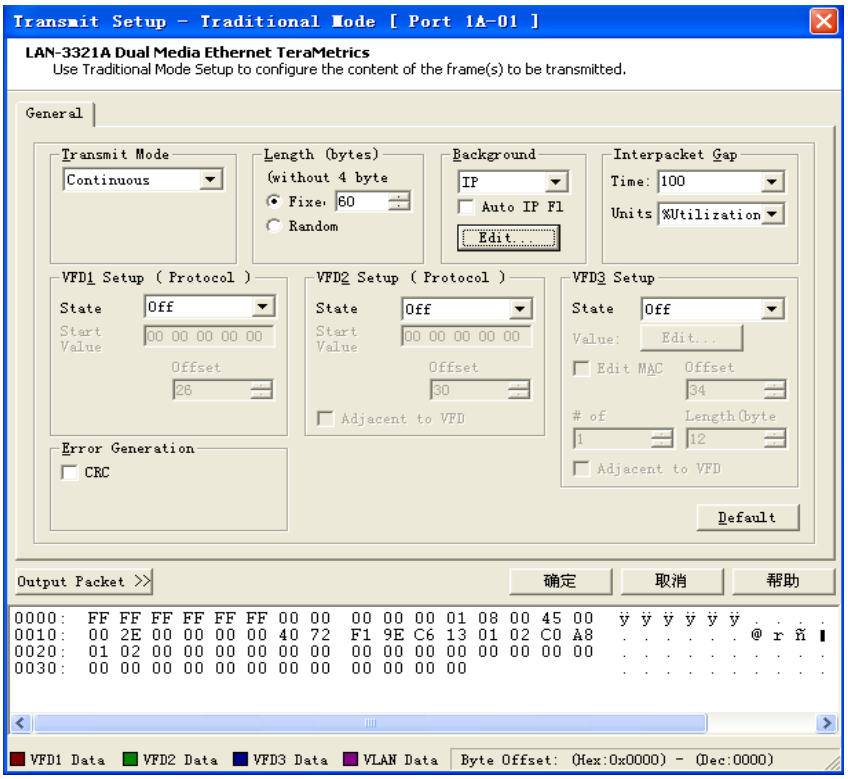

图 6.13 Smartbit L3 测试

测试结果如图 6.14 所示。上侧发送流量统计，发送速率为 1488093pps，下侧接受流量统计，接受速率为 1488093pps,字节接受和发送速率稍有不同。

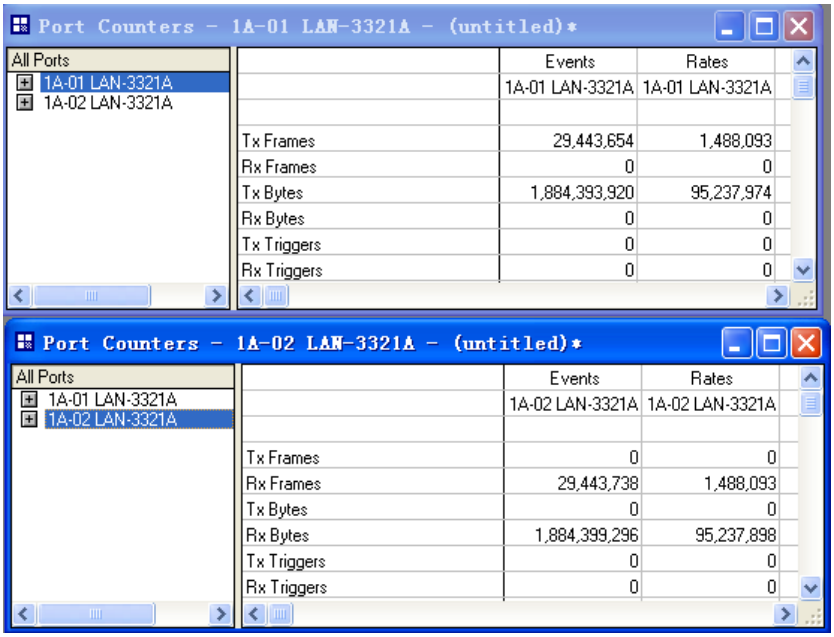

图 6.14 L3 测试结果

6.6 硬件流表 L4 转发及性能测试

L4 转发即直接对 TCP 端口号进行匹配，如果匹配成功，则直接进行转发。向硬件流表 Table0 下发 entry:

```
ip_dst=*;
transp_src=*;
transp_dts=0x80;
meter_id=1;//利用 meter1 对 TCP 流进行限速
```

发送流量，如图 6.15 所示。

1) 发送字节：64 （加上 CRC）
2) 发送速率：1488095pps
3) 发送 TCP 端口：0x80

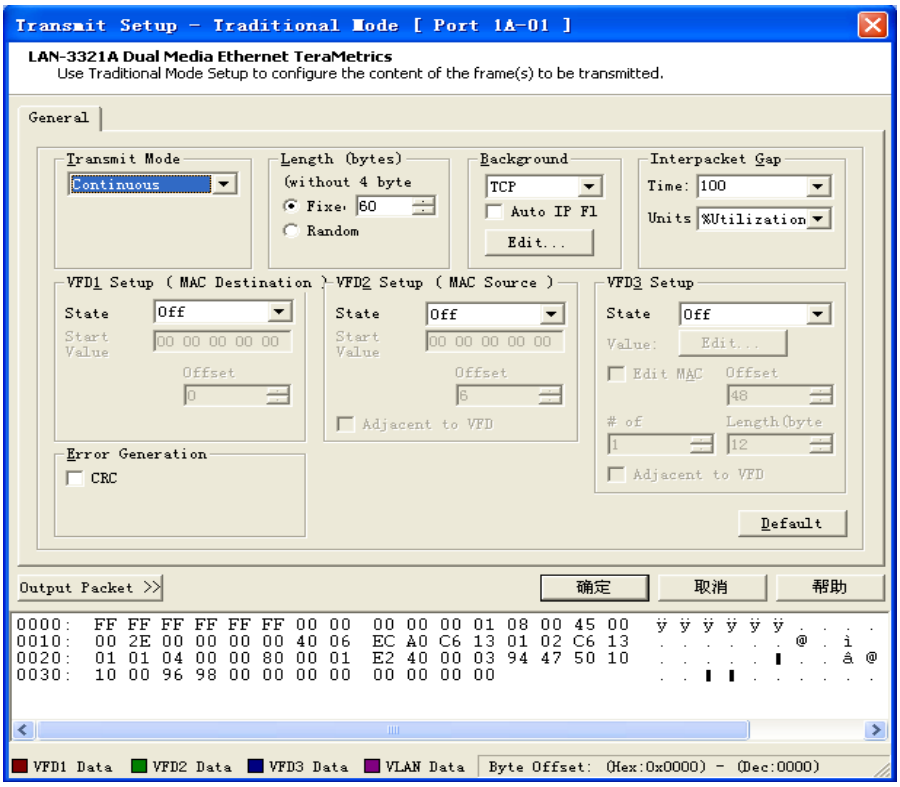

图 6.15 发送 TCP 流量

测试结果如图 6.16 所示。从测试结果可以看出，上行速率和下行速率相同，转发达到了线速。

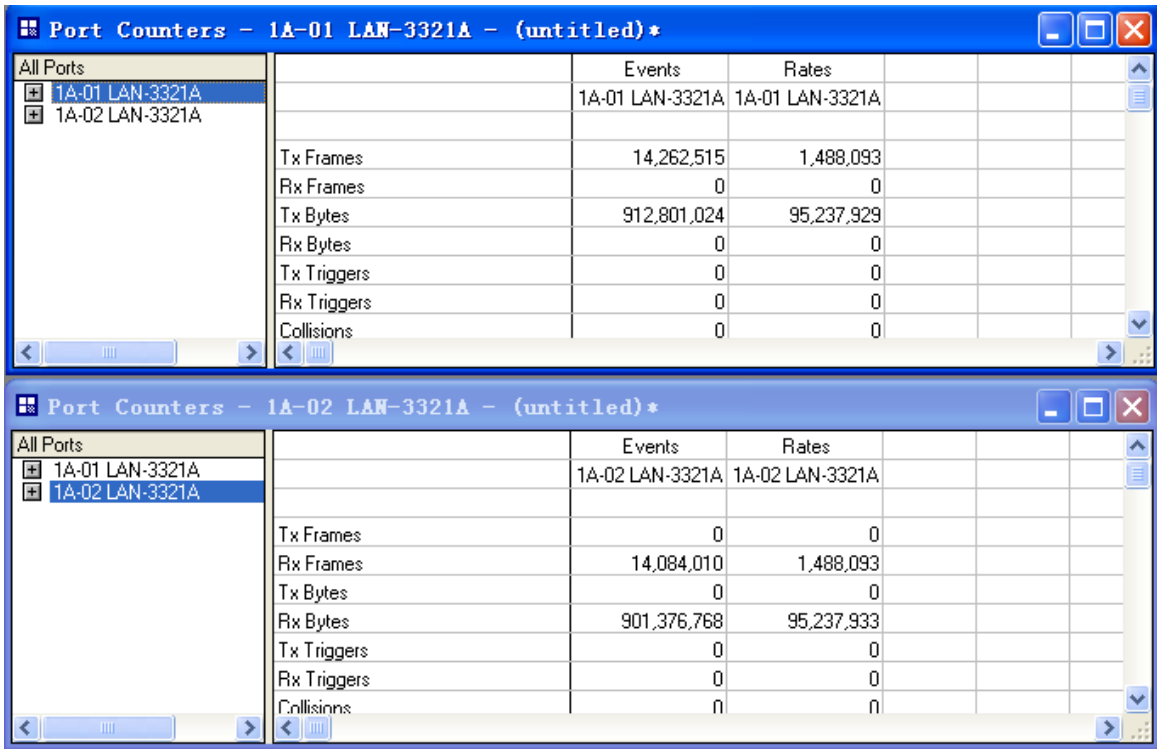

图 6.16 TCP 转发测试

使用 meter1 来对流进行限速。为使链路达到千兆极限， 这里使用 1500 字节的大包对交换机进行测试，如图 6.17 所示。

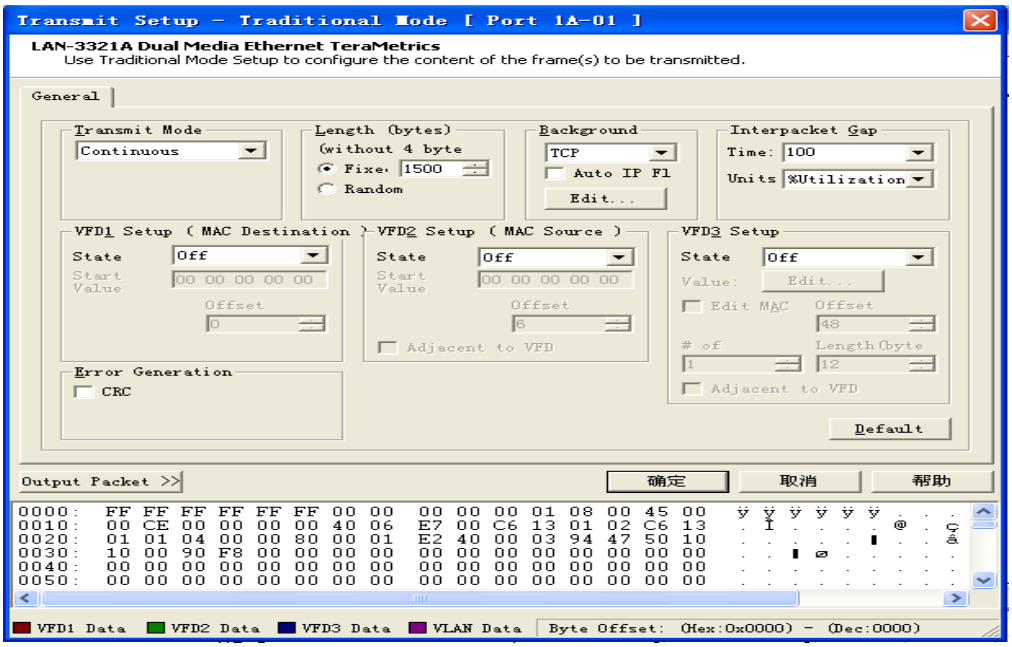

图 6.17 1500 字节大包传输

表 6.1，给出了测试结果，这里我们发送了从 64K 短包到 1500K 长包对交换机进行测试，并给出了 meter1 限速 1M 到 500M 的测试结果。

表 6.1 TCP 限速测试

包长	限速 1M	限速 5M	限速 10M	限速 50M	限速 100M	限速 500M
64	128177	640950	1281715	6401881	12787256	62317660
128	128244	640952	1281904	6406216	12803905	62714560
256	128081	641209	1281910	6408392	12812237	63898172
512	128510	640952	1281923	6409044	12816400	63636195
1024	127977	640953	1281905	6410493	12818075	63988995
1500	128444	640652	1281305	6409660	12819216	64025101

图中的转发速率单位为 Bps，换算成 bps 只需乘 8 即可。

6.7 交换机与控制器通信测试

测试过程：

1) 在主机上运行控制器程序并监听 6633 号端口

2) 在交换机上启动 ofprotocol 协议进程，与远端的 6633 端口进行连接

3) 使用 wireshark 对连接线路进行抓包，观察握手过程

预期结果：观察交换机与控制器成功建立通道，且协议类型为 OPenFlow1.3

如图 6.18，在 OpenFlow 交换机中启动连接控制器。

```
root@zynq:~# ofprotocol tcp:127.0.0.1:6632 tcp:172.16.75.158:6633
Jan 01 09:03:45|00001|secchan|INFO|OpenFlow reference implementation version 1.3.0
Jan 01 09:03:45|00002|secchan|INFO|OpenFlow protocol version 0x04
Jan 01 09:03:45|00003|secchan|WARN|new management connection will receive asynchronous messages
Jan 01 09:03:45|00004|rconn|INFO|tcp:127.0.0.1:6632: connecting...
Jan 01 09:03:45|00005|rconn|INFO|tcp:172.16.75.158:6633: connecting...
Jan 01 09:03:45|00006|rconn|INFO|tcp:127.0.0.1:6632: connected
Jan 01 09:03:45|00007|rconn|INFO|tcp:172.16.75.158:6633: connected
Jan 01 09:03:45|00008|port watcher|INFO|Datapath id is 000a35000001
```

图 6.18 OpenFLow 建立同 Controller 连接

使用 wireshark 进行抓包，设置 filter 为 of13.of_header，如图 6.19 所示。

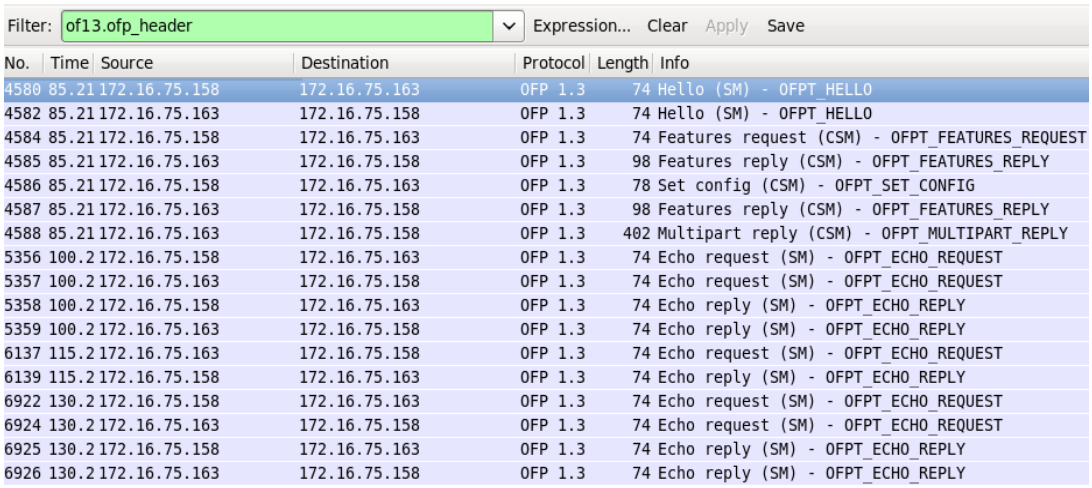

图 6.19 Wireshark 抓包

可以看到 wireshark 捕获了 OF1.3 的 hello 和 replay 包，也就两者之间建立连接的过程。打开一个包，查看其结构，如图 6.20 所示。OpenFlow1.3 协议目前是在应用层实现的，因此实际的练级仍是建立在 TCP 握手之上，使用 TCP 来在两者之间传输协议。

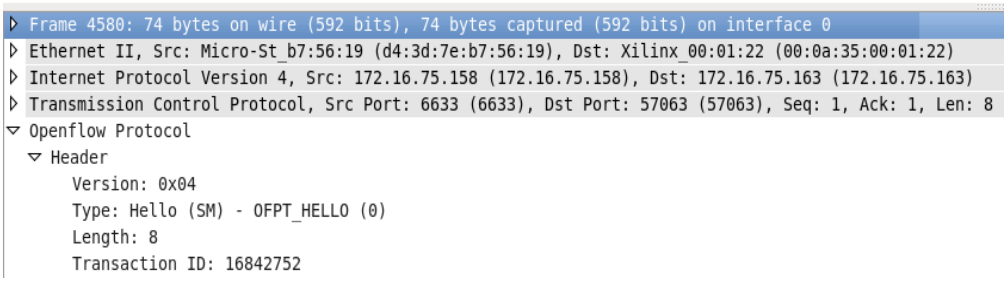

图 6.20 OF1.3 结构

6.8 本章小结

本章主要对交换机进行测试，在进行测试之前，首先介绍了使用该交换机的启动流程、一些注意事项、以及一些命令工具的使用，之后对交换机从物理层到 L4，再到控制器进行了简单的测试，尤其是对硬件流表的测试可以看出，使用硬件基本上可以实现数据包的限速转发。

通过这个测试，我们还可以看到 OpenFlow 的巨大优势，我们只需要对流表进行简单的操作，就能实现不同的层协议的转发，而并不需要更改网络硬件的任何东西。这里我们只是做了一些简单的测试，其实这些只是 OpenFlow 的冰山一角，如果上升到 SDN 的层面，我们还可以做的东西很多，包括高性能数据中心

的搭建，高层次网络应用，云计算等。

到这里，我们对如何在 ZYNQ 上搭建 OpenFlow 交换机进行了简单的介绍。我认为，OpenFlow 的出现并不是对传统网络的否决，而是对它进行了拓展，使得传统网络资源能够更加的合理的被利用，网络数据流量能够根据人们意愿进行合理的管理。

参考文献

[1]NetFPGA .Programmable Networking Hardware[EB/OL].http//netfpga.org, 2014.02.24/ 2014.04.08.

[2]Nick Mckeown, Tom Anderson, Hari Balakrishnan.OpenFlow:Enabling Innovation In Campus Netwoirks[J].ACM SIGCOM,2008,38(2):1-6.

[3]ONF.openflow enabled SDN and Networks Functions Virtulization[DB/OL].http://www.opennetworking.org, 2014.2.17/2014.4.8.

[4]Cisco.Cataly 6500 series overview website[EB/OL].http://www.cisco.com/en/US/products/hw/switches/ps708, 2013.2.20/2014.04.08.

[5]Openstack.Open source software for building private and public clouds[DB/OL].https://www.openstack.org, 2014.02/2014.04.08.

[6]Sushant Jain,Alok Kumar,Joon Ong,Arjun Singh.B4: experience with a globally deployed software define wlan[J].ACM SIGCOMM, 2013,43(4):3-14.

[7]ONF.OpenFlow Switch Specification1.3[DB/OL].http://www.opennetworking.org, 2012.06.25/2014.04.10.

[8]Xilinx Company.Zynq-700 All Programmal Soc Technical Reference Manual [DB/OL].http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-700-TRM.pdf, 2014.4.9.

[9]ARM Company.AMBA AXI and ACE Protocol Speicification[DB/OL].www.arm.com,2011.

[10]Xilinx.PS and PL Ethernet Perfomance and Jumbo Fram Support[DB/OL].http://www.xilinx.com/support/documentation/application_notes/xapp1082-zynq-eth.pdf, 2013.08.5/2014.04.09.

[11]Jad Naous, David Erickson,G.Adam,Guido Appenz, Nick Mckeown.Implementaing an OpenFLow Switch on the NetFPGA platform[J].ACM/IEEE,2008,4(1):1-9.

[12]ONF.OpenFlow Switch Spicification1.1.0[DB/OL].https://www.opennetworking.org/sdn-resources/onf-specifications/openflow, 2011.2.28/2014.4.9.

[13]NetFPGA.NetFPGA 10G OpenFlow Switch[DB/OL]https://github.com/NetFPGA /NetFPGA-public/wiki/NetFPGA-10G-OpenFlow-Switch,2013.04.10/2014.04.09.

[14]NetFPGA.NetFPGA 1G OpenFlow Switch[DB/OL]. https://github.com/NetFPGA /netfpga, 2013.02.24/2014.04.09.

[15]Kostas Pagiamtzis, Ali Sheikholeslami.Content-Adreessable Memory Circuits and Architectures[J].IEEE,2006,41(3):1-3.

[16]陆佳华，杨卫，周剑等.零存整取 NetFPGA 开发指南[M].北京:北京航空航天大学出版社，2010:147-182.

[17]Xilinx.Vivado Design Tools User Guide[DB/OL]. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_4/ug973-vivado-release-notes-install-license.pdf, 2014.01.10/2014.04.10.

[18]Xilinx.LogiCore IP AXI EthernetV6.0 Product Guide[DB/OL]. http://www.xilinx.com/support/documentation/ip_documentation/axi_ethernet/v6_0/pg138-axi-ethernet.pdf,2013.02.

[19]Xilinx.LogiCore IP AXI Direct Memory Acess V5.0 Product Guide[DB/OL]. http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v5_0/pg149_axi_dma.pdf, 2012.10.

[20]Jonathan Corbet, Alessandro Rubin, Greg Kroah.Linux Device Drivers[M].北京:中国电力出版社，2009:46-74.

[21]Arsalan Tavakoli, Martin Casado Scott Shenker.Applying Nox to Datacenter[J]. HotNets-VIII,2009.

[22]NOX.NOX architecture[EB/OL].www.noxrep.org, 2013.05/2014.04.

作者简历

教育经历：

2008 年~2012 年 贵州大学，计算机科学与技术专业，本科

2012 年~2014 年 浙江大学，软件学院，软件工程，硕士

工作经历：

2013 年 5 月到 2014 年 6 月， 于 Xilinx 上海实习，担任嵌入式软件工程师职务

致谢

感谢我的导师施青松教授在百忙之中对我论文的细心指导，感谢 Xilinx 陆佳华工程师在毕业设计中提供的大力支持和帮助。感谢你们的悉心栽培，感谢你们给我提供了展示自己的舞台和机会。

感谢项目中的小伙伴们，很庆幸能跟你们在一起工作，感谢你们在整个设计过程中的真诚帮助。感谢南京叠锶网络信息科技有限公司、西安交大网络中心实验室在本设计中提供的大力支持。

感谢斯坦福大学的 SDN 团队、OpenFlow 开源社区、NetFPGA 开源社区的研究人员和开发人员们，没有你们无私的奉献，就不会有这么先进的网络架构的产生，也就不会有本次毕业设计的完成。

最后感谢我的家人和朋友们，感谢你们给我的鼓励和关心。

署名　潘祖龙

于浙江大学软件学院

2014 年 4 月 17 日