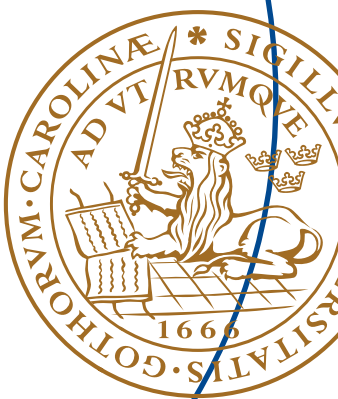
Master's Thesis

High Level Synthesis Tool for High Speed Packet Processing

Yian Wang
Venkata Soumya Pakki



Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, December 2013.

High Level Synthesis Tool for High Speed Packet Processing

by

Yian Wang

and

Venkata Soumya Pakki

Master's of Science in System on Chip

Department of Electrical and Information Technology

Lund University

Dec 2013

Abstract

The main objective of this Master Thesis is to design and implement a high level synthesis tool for high speed packet processing. For a given network packet, determining the destination and performing the required alterations to the packet are the key parts of Packet Processing. The idea is to provide customers a customized Ethernet switch which is reliable and flexible. As a requirement for this, a high level packet processing language (PPL) is designed instead of any hardware descriptive language because of the regularity of packet processing. The packet processing is described in a powerful way based on the PPL.

In this thesis, a design of Ethernet switch based on the PPL is proposed. Hardware implementation is done for the design and MyHDL is used as the hardware description language. Using Python, the compiled PPL program is translated into an hardware model. A tool has been developed which consists of a hardware generator and certain hardware infrastructures. Another part in the thesis is optimization of the initial design. For instance, optimization is done to run as much code as possible in parallel or for removal of unused hardware in the generated switch.

Verification is done and synthesis results have been listed comparing the two designs. Hence, we conclude that the initial design is more flexible and has more redundancy while the optimized design is more friendly to hardware cost and power consumption.

Acknowledgements

We initially thank Packet Architects AB for giving us an opportunity to work on this Thesis.

We are grateful to our supervisor Robert Wikander, CTO who gave us the opportunity to do this thesis and for encouraging us during the complete duration of the project. His ideas eased a lot to finish the project work.

Our sincere gratitude to Examiner at LTH, Viktor Öwall for his patience, constant support, motivation. Without his several feedbacks on our report, it would not have been done at this quality.

Finally, we would like to thank our family and friends for their encouragement.

Contents

List of Tables

List of Figures

Acronyms

ARP Address Resolution Protocol. 12

AXI4 Advanced Extensible Interface 4. 57

DEI Drop Eligible Indicator. 15

DFG Data Flow Graph. 66, 67, 69, 72

FCS Frame Check Sequence. 14

FIFO First In First Out. 22, 28, 29, 31, 52, 55, 58, 62, 63, 70, 77

FM Field Memory. 23, 24, 26, 27, 29, 31, 51, 52

FPGA Field Programmable Gate Array. 7, 8, 56, 57, 59

FSM Finite State Machine. 26, 40

IEEE Institute of Electrical and Electronics Engineers. 2, 5, 14–16, 71

IFG Inter Frame Gap. 16

IP Internet Protocol. 13

LAN Local Area Network. 2–4, 12, 14

LLC Logical Link Control. 2

MAC Media Access Control. 2, 4, 11, 12, 14–20, 56, 57, 62, 71

MM Modification Memory. 27–29, 31, 51, 52, 55

OSI Open Systems Interconnection. 1, 2, 7, 11

PCIe Peripheral Component Interconnect Express. 57, 77

PCP Priority Code Point. 15

PCS Physical Coding Sublayer. 56

PD Packet Decoder. 22, 23, 30, 40, 50, 61, 63, 64, 70, 71

PFP Packet Forwarding Pipeline. 22–31, 33, 34, 36–41, 44, 46, 47, 51, 52, 61, 62, 69–72

PHY Physical Layer. 1, 11, 56

PM Packet Modifier. 22, 27, 29, 31, 51, 55, 61–63

PMA Physical Medium Attachment. 56

PPL Packet Processing Language. 7, 8, 21, 22, 34, 37, 38, 67, 68

PS Parallel-to-Serial Converter. 17

RAM Random Access Memory. 31, 51, 52, 60

RR Round Robin. 52

RTL Register Transfer Level. 6–8, 21, 22, 34, 35, 38, 41–47, 49, 62, 63, 66–68, 72

SFP+ Enhanced Small Form-factor Pluggable. 56

SP Serial-to-Parallel Converter. 17, 18, 29, 71, 76

TPID Tag Protocol Identifier. 15

VF Variable File. 43–45

VID VLAN Identifier. 15, 19

VIO Virtual Input/Output. 57

VLAN Virtual LAN. 12–15, 17–20

WAN Wide Area Network. 13, 14

XGMII Ten Gigabit Media Independent Interface. 56

Chapter 1

Introduction

1.1 Background

It has been 40 years since the invention of Ethernet. This technology was born at Xerox PARC in 1973 which initially aimed at solving communication problems between personal computers and peripherals. In the past 40 years an explosive development has occurred and the speed of Ethernet has improved by one order of magnitude every 10 years, which gives us reason to believe that Ethernet will keep this growing trend in the future.

In December of 1982, the advent of IEEE 802.3 remarked the take-off of Ethernet standards. IEEE 802.3 provides the technical standards in order to use Ethernet in the Physical Layer (PHY) and the data link layer of Open Systems Interconnection (OSI) model. The OSI model is a layered framework describing data communication in a network system. It contains seven separate but related layers as shown in Table 1.1 [1].

Table 1.1: OSI model

Layer	Function
7. Application	Network process to application
6. Presentation	Data representation, encryption and decryption
5. Session	Interhost communication, managing sessions between applications
4. Transport	Reliable delivery of packets between points on a network
3. Network	Addressing, routing and (not necessarily reliable) delivery of datagrams between points on a network
2. Data link	A reliable direct point-to-point data connection
1. Physical	A (not necessarily reliable) direct point-to-point data connection

The seven layers can be divided into three subgroups. Layer 1 to 3 (the physical layer, the data link layer and the network layer) are the network support layers which are in charge of transferring data from one device to another. Layer 5 to 7 (the session layer, the presentation layer and the application layer) are the user support layers which allow interoperability among unrelated software systems. Layer 4 (the transport layer) in the middle links upper subgroups and lower subgroups. Normally, user support layers are implemented in software while lower network support layers are a combination of hardware and software. Figure 1.1 shows how a message is sent from device A to device B through the OSI model. During data transmission, it may pass through some intermediate nodes which usually involve network support layers [2].

The relationship between the OSI model and Institute of Electrical and Electronics Engineers (IEEE) 802.3 is shown in Figure 1.2. The IEEE has subdivided the data link layer into Logical Link Control (LLC) and Media Access Control (MAC). The LLC provides one single data link control protocol for all IEEE Local Area Network (LAN)s while the MAC protocol varies in different LAN standards. IEEE 802.3 is a LAN technology and also the most widespread LAN standard. Because of its forthright

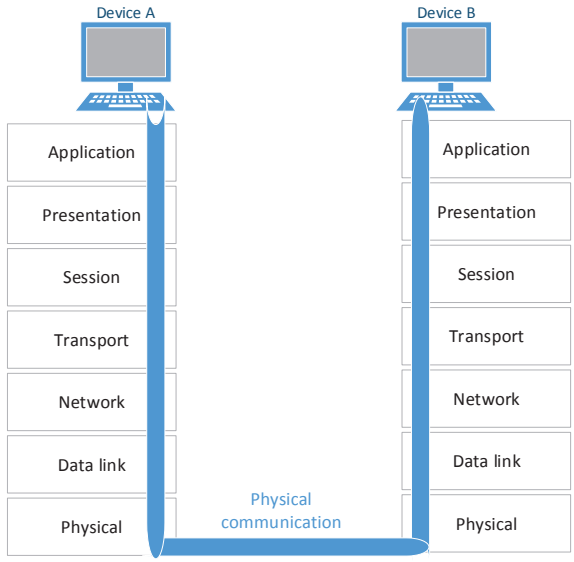


Figure 1.1: Example of data transmission through OSI model

principle and low cost implementation, it has replaced other LAN standards such as IBM token ring and Arcnet models to a great extent [3]. The remarkable property of Ethernet is its frame format. Based on the fixed frame format, Ethernet provides flexibility to be enclosed in different network equipments and medium so that the network operators will prefer to use Ethernet technology to organize low cost, high performance networks.

Information transmitted on an Ethernet network is called Ethernet packets. They are simply chunks of data enclosed with some identifiers so that they can be sent to the correct destination. If there are multiple devices in a network, it is a problem to make one to one communication possible. A switched network is the most common solution in current market which use switches as interlinked nodes to create temporary connections between multiple devices linked to the switch. An Ethernet

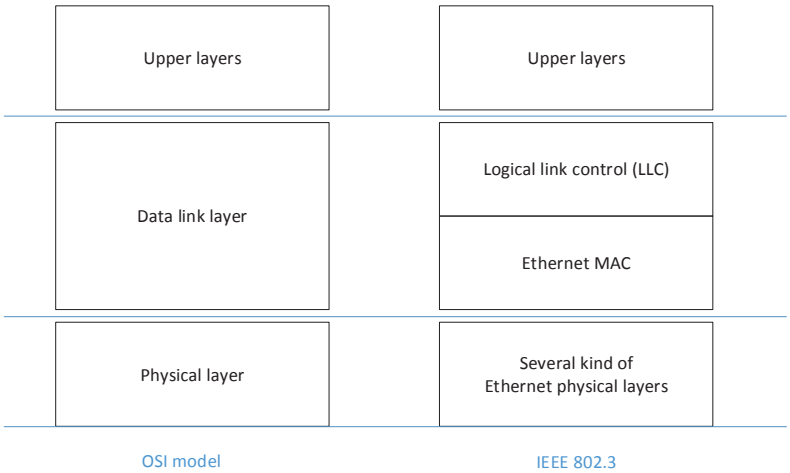


Figure 1.2: OSI Model VS IEEE 802.3

switch in a network system transmits packet data at Ethernet standard rates. It is functioning as the traffic control center for the LAN and managing the transmission of Ethernet packets between multiple devices. An Ethernet switch manages all connections through table lookups in order to find relevant destination addresses and connected ports. When a packet is received by an Ethernet switch, it extracts destination MAC address information from the packet and creates a temporary connection between source and destination MAC addresses. Hence, the data can be routed to the destination and the connection is closed. In addition, in a LAN configuration, each device has a specific bandwidth and connection to other devices on the network. The advantage of this structure is that all devices can work at full capacity without impacting other connected devices.

Nowadays, 10-Gigabit (10G) Ethernet switches are being manufactured in volume. Higher speed (e.g.40G, 100G) switches are also introduced on the market. By 2015, the higher speed Ethernet will have about a 25% share of network equipment ports,

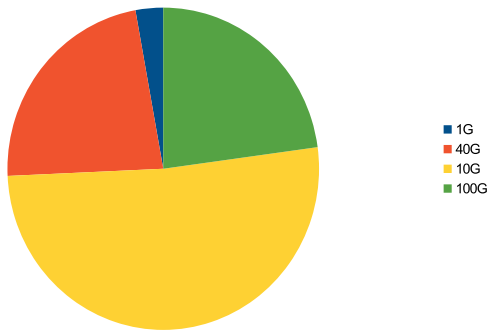


Figure 1.3: Projected Share of Network Ports - 2015

according to Infonetics Research in Figure 1.3. In April 2013, IEEE 802.3 announced the work group of 400G Ethernet standard and even a higher speed Ethernet standard was on the agenda. High speed Ethernet ensures the possibility of more complicated Ethernet applications and meanwhile it extends to more fields like automobile industry, super computers and data centres which currently have their own specialized networks. Automobile network system, as an example, is becoming more complicated and scaled as a result of explosion of electronic products and entertainment systems in the car. This provides a big challenge for specialized automobile networks offered by different vendors and requires high compatibility and fully open-ended network systems. Ethernet will obviously be a dream solution and several products are already on the market. [4]

Due to the coming tremendous number of applications of Ethernet, the next generation of the Ethernet equipment should be able to handle vast spans of features and performance requirements. To put this in a nut shell, it has to be flexible. On the

current Ethernet switch market, the customer can always find products that handle large scale processing capacity. However, an Ethernet switch for a certain specific task is seldom introduced. Therefore, the customer has to pay extra money to purchase a product with unnecessary functionalities.

For different network niches, the main structure of the Ethernet equipment could be the same but different packet forwarding mechanism might be needed. From the industry's point of view, trivial changes in packet forwarding algorithms or switch configurations might require a totally different custom hardware architecture design[1]. Most of current switch vendors prefer to provide switches with enough complexity to handle most packet forwarding types even though customer oriented market is almost a blank sheet. Hence, it is a challenging and rewarding task to find a way of delivering the needed performance and features for packet processing without resorting to a custom RTL design. As we know, the speed of Ethernet keeps growing, new Ethernet generation will require new network equipments. Traditional hardware design flow has a long development cycle but we will try to reduce it dramatically. Meanwhile since the development period is decreased, specific network equipment can be implemented that only meet the exact requirements and consequently reduce the developing cost and product price.

1.2 Scope of the Thesis Project

This master's thesis is part of the Flexswitch project at PacketArc AB, aiming at providing customer oriented and cost-effective high speed Ethernet switch solutions.

[1]The hardware architecture design in this report are referred to Register Transfer Level (RTL) design.

Imagine that as a customer there is no need for seeking appropriate Ethernet switches in the market, just describe particular requirements in a C-like descriptive language, and a dedicated switch for you is around the corner by transforming description via the Flexswitch platform.

The Flexswitch project attempts to finally create a platform containing a serial of tool chains. Based on this platform, customized high level Packet Processing Language (PPL) is offered where packet processing can be described in a simple, yet powerful way. The PPL can let customers define their own switch configurations and describe their demand packet forwarding strategies easily and quickly. With the aid of the PPL, customers have the flexibility to create customized Ethernet switches. This tool chain works in three steps to fill in the gap between the PPL and the RTL implementation:

1. Compile the PPL program to a intermediate model with lower level format which can present a certain hardware behaviour.

2. Analyse the optimized low level format and generate the corresponding RTL.

3. Instantiate the customized RTL with general hardware infrastructures to obtain the required Ethernet switch.

This master's thesis project mainly deals with the last two steps, striving for developing an RTL generation tool that can handle various packet processing requirements.

1.2.1 Target of the Thesis Project

The target of this thesis project is to create a prototype for a 10G switch working on Layer 2 of OSI model and run through a Field Programmable Gate Array (FPGA)

design flow, the design requirements as listed as follows:

- A Layer 2 switch supporting at maximum 6 ports.
- Target FPGA is Xilinx Virtex-7, part number XC7VX690T.
- Clock frequency in switch core domain is 105 MHz.[2]
- Scheduling used in the switch has fixed strict priority.
- Packet processing starts after getting a full Ethernet header.

The tool that needs to be developed mainly consists of one RTL generator and certain hardware infrastructures for packet processing. The basic idea of the RTL generator is to analyze the intermediate file format compiled from PPL in order to generate RTL for packet forwarding algorithms automatically. Parametrized hardware infrastructures are also on the design list to combine with the generated RTL and finally form a packet processing subsystem.[3]

1.2.2 Organization of the Thesis Report

This report is divided into chapters explained as below:

- Chapter 2 explains subsystems of a Layer 2 switch and forwarding algorithm.
- Chapter 3 elaborates on design details for a flexible packet processing subsystem.
- Chapter 4 Presents one Layer 2 switch implementation with different configurations.

[2]This is the calculated frequency to get a full cell rate for six 10G ports with 160-byte cell size. When a packet is queueing in a switch and waiting for sending to its destination, it is sliced into fixed size cells. The switch should promise to process one cell per clock cycle.

[3]Subsystems in a Layer 2 switch will be explained in the next chapter.

- Chapter 5 optimizes the implemented tool chain based on results from Chapter 4.

- Chapter 6 discusses the conclusions and the future work.

Chapter 2

Ethernet Switch

2.1 Overview

Traditional Ethernet switches operate at Layer 2 of the OSI model. Rather than Layer 1, which is the PHY describing electrical interfaces, it is in Layer 2 (Data link layer) where packets are sent to a specific switch port based on MAC addresses [5]. A Layer 2 switch can be treated as a multi port bridge which can learn new source MAC addresses automatically after receiving packets and building a table to maintain the decision of packet forwarding. The advantage is that multiple switching paths inside a switch can be active simultaneously. For example, consider Figure 2.1 which is a Layer 2 switch connecting 4 devices, A to D. If device A is sending data to device B while device C is communicating with device D, data transmission could happen at the same time in the switch. Each port has its own bandwidth so the speed is promised and the collision domain in this network is divided into 4 domains by the switch.

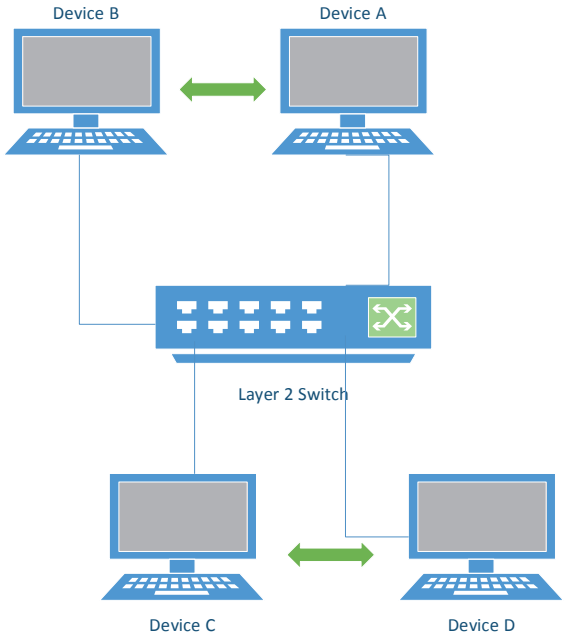


Figure 2.1: Example of a Layer 2 Switch

However, the scope of a LAN is seldom so simple. In a large LAN it is always necessary to divided it into several sub domains which is called Virtual LAN (VLAN). VLAN is raised in order to cut off broadcast domains. In an Ethernet network, communication is based on MAC addresses, hence one device will frequently use Address Resolution Protocol (ARP) request to get the MAC address of the destination port. This request will send a packet to all ports belonging to the same broadcast domain. In a large LAN, this costs huge amounts of unnecessary bandwidth. As a solution, each VLAN will have its own broadcast domain and devices in different VLANs cannot communicate through a Layer 2 switch. As in Figure 2.2, device A and B belongs to VLAN 1 while device C and D belongs to VLAN 2. The previous scenario which communication between device A and B or device C and D has no interaction,

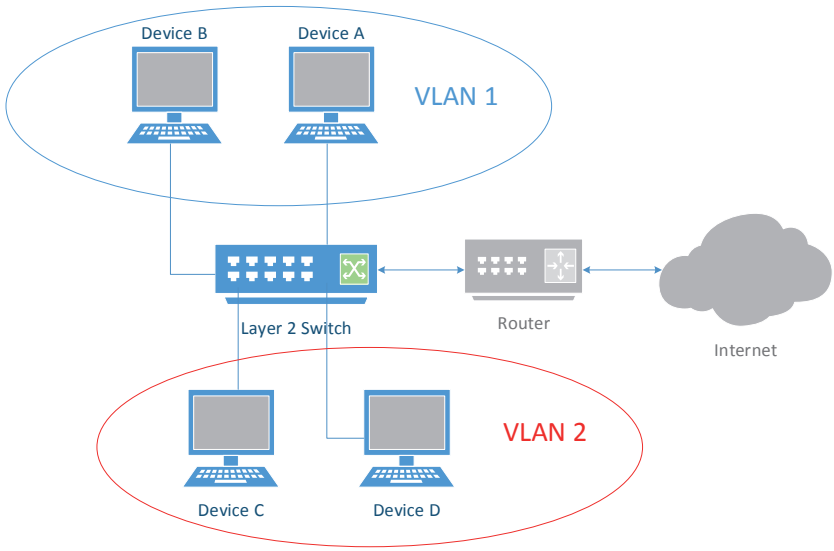


Figure 2.2: Example of a Layer 2 Switch with two VLANs

but the Layer 2 switch cannot establish switching path between device A and C because they belong to different VLANs. In this case, an additional router is needed. A router is normally working on Layer 3 (network layer), the most common protocol on this layer is Internet Protocol (IP) [6]. The router is typically connected to one of the switch ports and do IP routing between different VLANs. In reality, a router is also connected to Wide Area Network (WAN) (e.g. Internet) or other switches. However, the fact that routers are mostly based on software implementations lead to a performance bottleneck of the entire network. Therefore, Layer 3 switches are promoted on the market which are high performance devices for network routing. They are quite similar to routers and use the same IP routing table for fast Layer 3 forwarding. The key difference between Layer 3 switches and routers lies in the built-in hardware of the switch. Hardware inside a Layer 3 switch bridges Layer 2

Table 2.1: Typical Ethernet Frame structure

Preamble	Start of Frame Delimiter	Header	Payload	Frame Check Sequence (FCS)

switches and ordinary routers, accelerates a router by replacing software logics in it with hardware. Also, Layer 3 switches typically do not possess WAN ports as a router has, instead the routing functionality is to route between different subnets or VLANs of a large LAN, such as on a campus or within an enterprises [7].

In this thesis project, we focus on building Layer 2 switches to begin with. Layer 3 protocol is much more complicated than Layer 2 so Layer 3 switches require more sophisticated hardware infrastructures. During the development, features of the Layer 3 switch will not be implemented but the capability to extend the tool chain to support Layer 3 switches is taken into consideration.

2.2 Ethernet Frame Standard

Ethernet standard is specified by various IEEE 802.3 specifications and Table 2.1 gives a typical Ethernet frame.

Normally the header of an Ethernet frame contains all information required by a Layer 2 switch. Based on different Ethernet standards the frame header could have several encapsulations. During our design, the IEEE 802.3-2012 and the IEEE 802.1Q are taken into account. Table 2.2 is IEEE 802.3-2012 packet header containing a 6-octet destination MAC address, 6-octet source MAC address and 2-octet Ethertype or length.

If there is an IEEE 802.1Q tag in an Ethernet packet, it means VLAN is supported. In this case, 4-octet field will be added between the source MAC address and the

Table 2.2: 802.3 Ethernet Header

MAC destination	MAC source	Ethertype or length
6 octets	6 octets	2 octets

Table 2.3: Ethernet Header with VLAN Tag

MAC destination	MAC source	802.1Q t	
6 octets	6 octets	2 octets	
		Tag Protocol Identifier (TPID)=0x8100	Priority Code Point (PC

Ethertype/length fields as in Table 2.3.

In the 802.1Q tag, first 2 octets are TPID. It is set to 0x8100 in order to distinguish the frame from untagged frames. The last 12 bits indicate VID, specifying the VLAN to which the packet belongs. In between there are 3-bit PCP and 1-bit DEI.

2.3 Layer 2 switch subsystems

Figure 2.3 demonstrates a high altitude look at the data flow through a Layer 2 switch. It is exemplified using a switch with three 10G interfaces. The minimum port bus width is calculated from the MAC based on the bandwidth and clock frequency as shown in Equation 2.1. In our implementation, the 10G MAC is Xilinx 10G MAC Intellectual Property core working at 156.25 MHz so the minimum bus width is 64 bits. Another bottom line is the packet rate in the switch which will get the maximum rate when all packets are with minimum size. In the IEEE standard, the minimum Ethernet packet size is 512 bits. The clock cycle needed for the minimum packet is calculated by Equation 2.2 and the bit rate is in Equation 2.3.

$$buswidth = ceil(\frac{bandwidth}{frequency * 8}) * 8. \tag{2.1}$$

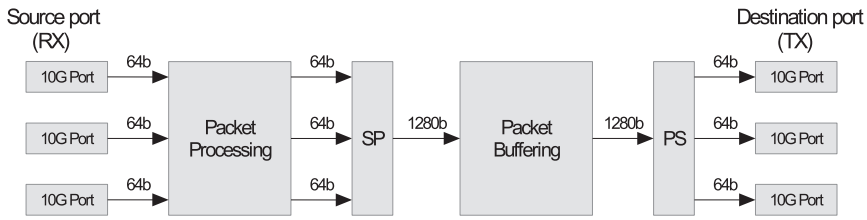


Figure 2.3: A Bird Eye view of the data flow

$$cycles = ceil(\frac{minPkt}{buswidth}). \tag{2.2}$$

$$brate = frequency * buswidth. \tag{2.3}$$

When it comes to the real packet transmission, there is an Inter Frame Gap (IFG) between two packets. In the IEEE standard it is 160 bits for a 10G MAC. Therefore, the transmission rate is shown in Equation 2.4

$$rate = bandwidth * \frac{cycles * buswidth}{cycles * buswidth + ifg}. \tag{2.4}$$

To ensure the speed of the switch, the transmission rate should be larger than the bit rate, otherwise the bus width needs to be increased. As a conclusion, the port bus width for a 10G MAC in our design is 64 bits and it is treated as the size of one packet chunk.

From the general view in Figure 2.3 the switch could be typically divided into two subsystems: Packet Processing and Packet Buffering. In the packet processing subsystem, the processing unit is one packet chunk. The packet header is analysed, based

on header properties forwarding decision such as unicast, multicast, broadcast or drop packet is made. In the packet buffering subsystem packets are received by a fixed cell size (in our case it is 1280 bits) and the cells are stored in memories with linked addresses until sending to the destination ports. Between the two subsystems there is a Serial-to-Parallel Converter (SP) which divides packets into a number (depending on packet size) of packet cells. The packet cells from the packet buffering subsystem will be concatenated into a stream of bytes by a Parallel-to-Serial Converter (PS) and finally received by destination ports.

The packet processing subsystem includes 4 parts which are shown in Figure 2.4:

- Packet decoding
- Packet forwarding
- Lookup tables
- Packet modification

`Packet decoding` parses and analyses the packets to detect Ethernet encapsulation and determines if the packet was VLAN tagged or not. Also, all information in the header that is used for processing packets such as source and destination MAC addresses will be extracted.

`Packet forwarding` processes packet information provided by the packet decoder and makes forwarding decision based on table lookups. There are several `lookup tables` in a switch, some containing default information of source ports while others may be dynamic during packet processing because of learning and ageing. Learning occurs when a packet is sent from a new source MAC address that has not been stored

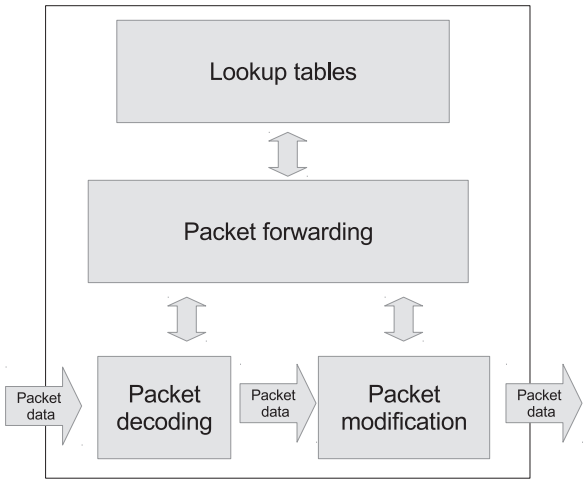


Figure 2.4: Block Diagram of Packet Processing Subsystem

in the switch. The switch will then store this address with its port number to a lookup table and therefore the next time if this address is the destination address the switch will know its corresponding port. However, if a switch learned some address a long time ago that not has been needed afterwards, it might waste time searching through stale addresses. To solve this, when the mapping of MAC addresses and port numbers are added to the lookup table they will be attached with timestamps. If an entry has no activity for a certain amount of time it will be freed, which is called ageing.

`Packet modification` is necessary if the source packet should be modified based on forwarding result, such as remove VLAN tag, replace the MAC header, etc. Finally, the received packets are divided by the SP into a number of cells with their forwarding results.

The packet buffering subsystem takes packet cells as input. The cells are stored in

buffer memories until a queue engine and packet scheduling figure out the sequences of transmitting these cells to their destination addresses. Since the packet buffering subsystem is not involved in this thesis project, details will not be discussed.

2.4 Packet Forwarding Algorithm

The basic packet forwarding algorithm in a Layer 2 switch is to forward packets from source MAC address to destination MAC address. Based on this, various functionalities can be added according to specific requirements. One example is Layer 2 address learning or ageing that has been shown in the previous section. Different packet forwarding algorithms may require different packet processing subsystems. That is the reason of developing a flexible switch generation platform.

The packet forwarding algorithm that has been proposed initially to test the platform is shown in Figure 2.5. The process starts with checking the VLAN tag in the packet. If the packet is VLAN tagged, the correct VLAN information is extracted from the VLAN field in the packet, otherwise, a lookup in specific table is performed to find default VLAN information of the source port. Once VID and VLAN priority is settled, two table lookups will be done in a VLAN membership table and VLAN priority table separately to get more content. The first lookup tells the global identifier of the packet as well as membership port mask. The second lookup gets the mapping from VLAN priority to its relevant queue in the packet buffering subsystem. Next check is the broadcast bit (bit 40) in the destination MAC address. If bit 40 is 1, this packet will be sent to the every port listed in a membership port mask. If bit 40 is 0, firstly VLAN membership of the source port will be checked. If the source port

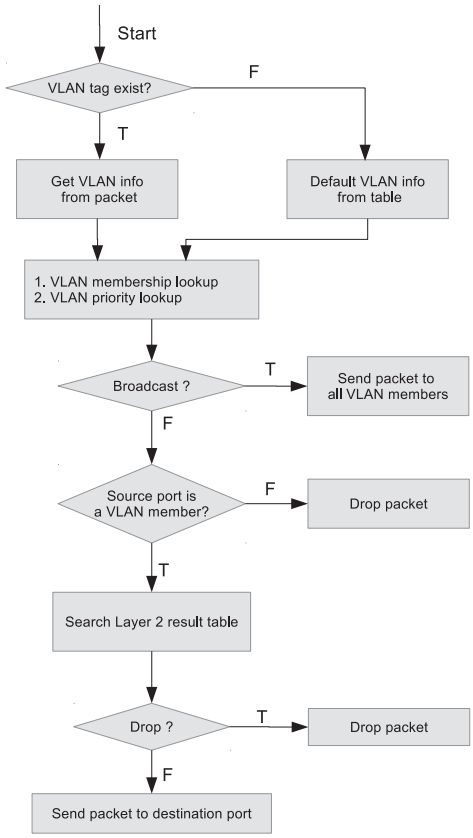


Figure 2.5: Flowchart of Proposed Packet Forwarding Algorithm

is not a VLAN member, this packet should be dropped. Otherwise, there is a Layer 2 result table for searching destination MAC addresses. Based on the lookup result, the packet could be either dropped or sent to the destination port.

Chapter 3

Packet Processing Implementation

According to the project target, to implement different hardware-based packet forwarding algorithms in switches, we firstly use PPL to describe needed algorithms, together with other initializations or configurations belong to the packet processing subsystem. Secondly, the PPL program needs to be compiled to a low level format which is an intermediate stage between PPL and RTL. To figure out potential options for this intermediate stage, the first reasonable idea suggested is the assembly program. In a processor based architecture, assembly program can be operated on hardware directly. If we develop specific instruction sets for packet forwarding algorithms, each instruction could be directly mapped to a detailed hardware structure.

As the framework in Figure 3.1 shows, the PPL that has been developed for this project is similar to C and includes two parts, a header file and a main function. On one side, the main function is referred to packet forwarding algorithm. The packet forwarding algorithm is compiled to assembly codes which will be used to generate the corresponding hardware packet forwarding block by an RTL generator. On the

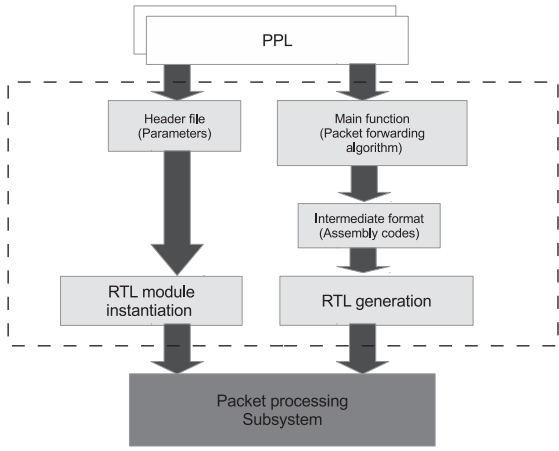


Figure 3.1: Framework of Flexswitch

other side, the header file contains all parameters and global table/register definitions. Since the rest of the packet processing subsystem has a relatively fixed structure for various instantiations, only different parameters are needed and the header file will be in charge of all parameter declarations. Based on the two flows in Figure 3.1, the proposed packet processing subsystem could be roughly divided into two parts as in Figure 3.2:

- Packet Forwarding Pipeline (PFP): This block will be generated by the RTL generator entirely.
- Hardware infrastructure around PFP : This part mainly contains the rest of the blocks belonging to packet processing subsystem like Packet Decoder (PD), Packet Modifier (PM), Packet First In First Out (FIFO) and lookup tables. All hardware infrastructures are configurable by header file in the PPL program.

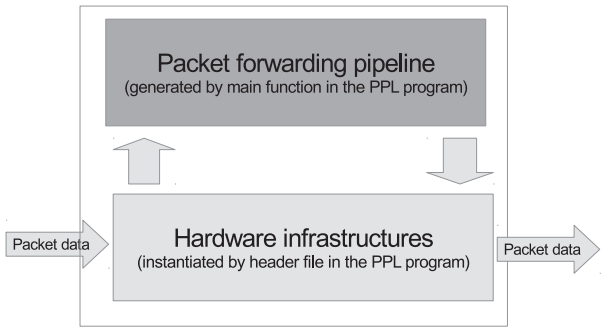


Figure 3.2: Design division of Packet Processing Subsystem

In the following sections, this proposal will be presented and discussed in more detail to determine if this is a reasonable solution.

3.1 Hardware Infrastructure

3.1.1 Packet Decoding

The basic requirement of packet decoding is to read certain bits from the packet frame. The decoding part demonstrated in Figure 3.3 contains PD, Field Memory (FM) and packet read stage in PFP. As we know, one packet will be fragmented into chunks according to port bandwidth. When the chunk size varies, the data needed for packet forwarding might appear either inside one chunk or continuously in several packet chunks. The flowchart of reading one single needed packet field is demonstrated in Figure 3.4. The required packet field is stored in a register, once all the bits are stored, the decoding block will send this field to the FM.

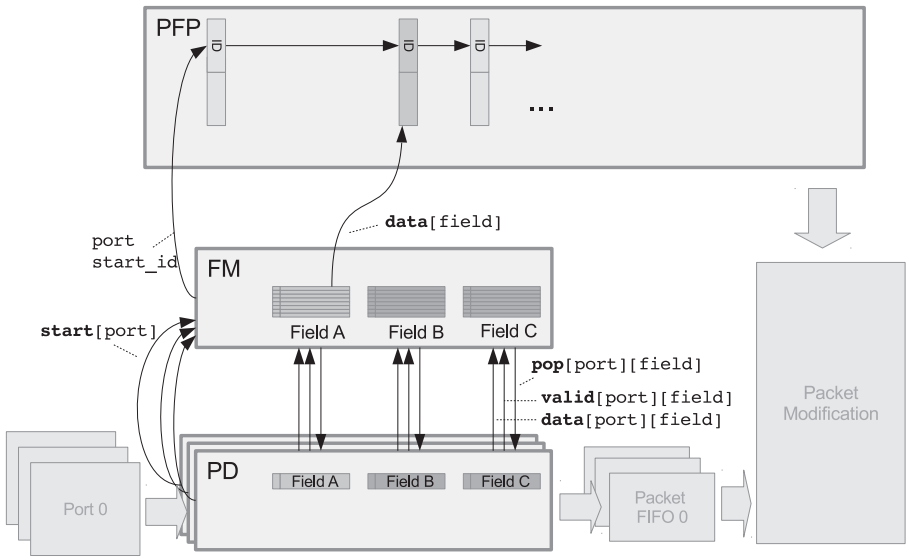


Figure 3.3: Block diagram of Packet Decoding

Figure 3.4 is an example for a single port. In reality, each source port has one decoder and between those packet decoders and packet forwarding units, a controller is introduced which in our case is the FM. The FM is in charge of allocating needed packet data fields to relevant PFP stages for each source port. In addition, since packet fields from different ports are collected in parallel and the PFP can only process one port per clock cycle, the FM is also responsible for scheduling the visit frequency to all ports. The FM is supposed to be intelligent enough to allocate different priorities if port speed varies. In the initial design, since only 10G ports exist, the scheduling here is the round robin selection [8]. Correspondingly, in the forwarding pipeline, a new variable will be created for storing the data for every

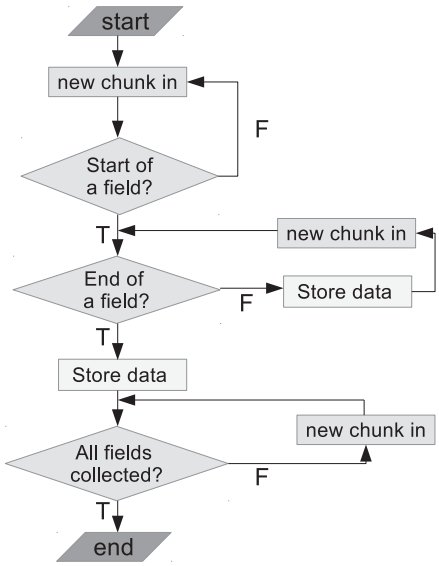


Figure 3.4: Flowchart of reading certain bits from one packet frame

relevant stage.

The relevant stage in the PFP is called packet read stage which refers to a particular instruction and will be introduced in Section 3.2. However, its hardware structure differs depending on the start strategy we choose for PFP. For a particular packet read stage, it is driven by the previous pipeline stage as well as extracted packet data. Only when both of these two inputs are ready, the packet read stage could be triggered. Suppose variables from previous pipeline stage and data from packet get ready for packet read stage randomly, for each port there is a buffer inside this stage. As a consequence, the packet read stage may have four status:

1. Ready: Data from the packet has been stored in the buffer and meanwhile, the previous stage has provided a valid variable.

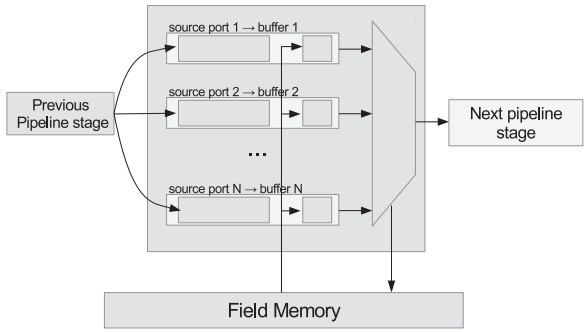


Figure 3.5: Block diagram of packet read stage

2. Packet data ready: Data from the packet has been stored in the buffer, but variables in the previous PFP for this source port is not ready.

3. Variable ready: Previous pipeline stage has provided variables for this source port, but the packet data is under preparation.

4. Empty: There is no useful data in the buffer, either it has been transmitted to the next stage or both variables and packet data are not stored in the buffer.

If we look inside this stage as in Figure 3.5, for a certain number of source ports, same number of buffers are provided. Only when the buffer is in status `Ready` it could be used for the next stage. Otherwise, the packet read stage should switch to another port buffer with status `Ready` for further processing. In fact, each source port in this stage is controlled by a Finite State Machine (FSM) shown in Figure 3.6.

However, this is under the assumption that trigger time cannot be estimated. Actually, instead of starting the PFP as soon as new packet comes in, if the PFP starts only after the FM has collected all required fields, buffers inside a packet

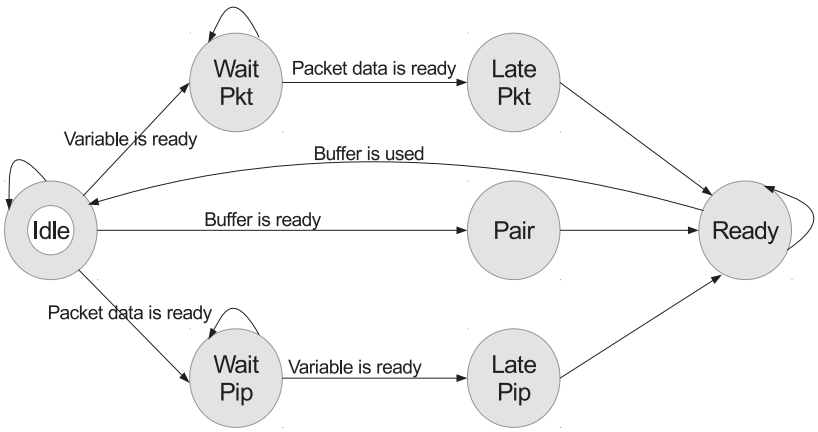


Figure 3.6: FSM of packet read stage with single port

read stage is no longer needed. Therefore, the status of a packet read stage is only dependent on valid variables from previous stages in the pipeline. As specified in the project target, packet processing is started after getting a full Ethernet header, so the scheduler inside the FM is enough to handle the status of packet read stage. But still, different kind of hardware could be easily configured based on different requirements, that is what the project is aiming at.

3.1.2 Packet Modification

When the forwarding decision is made in the PFP, packets will be filtered in packet modification block as in Figure 3.7. Packet modification contains Modification Memory (MM) and PM which is similar with the components in packet decoding. Ideally, the PM has three functionalities: packet overwrite, packet insert and packet delete. Normally in a Layer 2 switch packets will not be modified, but we still keep this

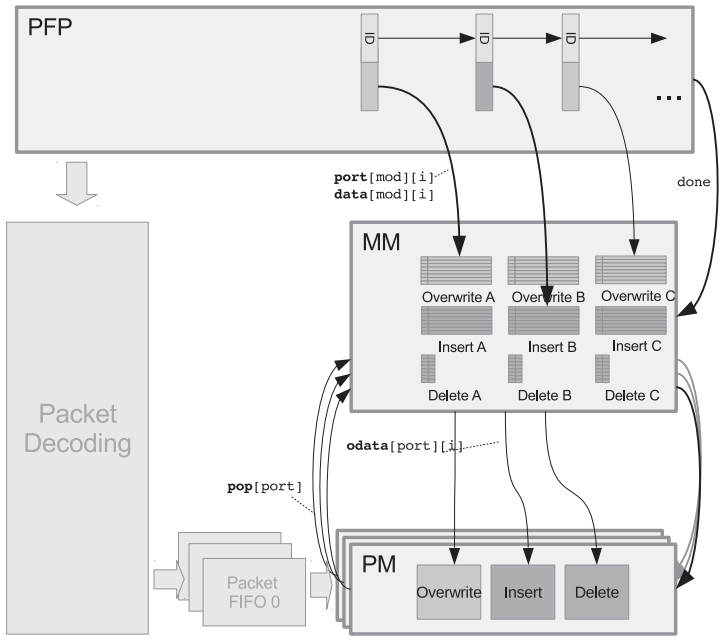


Figure 3.7: Block diagram of Packet Modification

structure in case of further extension. Hence for a Layer 2 implementation, packet modification mainly plays a role of synchronizing packets from packet FIFO and forwarding decisions from the PFP. Depending on various forwarding possibilities, the decision out from the PFP might be out of order, so in MM, the decision order will be sorted again and in keeping with packet FIFOs.

3.1.3 Packet Processing Interface

The interface between blocks of hardware infrastructures and the PFP can be sorted into two groups: packet transmission interface and PFP interface. Packet trans-

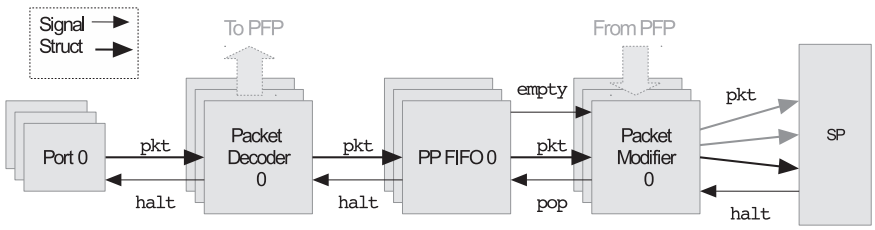


Figure 3.8: Packet Transmission Interface

mission interface (`pkt` interface) is used to transmit packet data and PFP interface manages connections between the PFP, the packet decoding block and the packet modification block.

As demonstrated in Figure 3.8, all the packet data transportation is achieved using the `pkt` interface from source ports to the SP. Meanwhile, in the opposite direction normally there is a `halt` signal between two modules except the interface between the packet FIFO and the PM. The packet FIFO stores packet data when the PFP is making the forwarding decision, so it will drive the next valid data as soon as it is available, but it will not pop the next data to the PM until `pop` is set high which means the forwarding decision for this packet has been made. Except the packet FIFO, the rest of the modules in the `pkt` interface will transmit packet data to the next module as fast as possible unless the next module cannot consume the incoming data and send a halt flag to the previous module.

Figure 3.9 shows interfaces around the PFP. Inside the FM, there is a free block waiting for start flags from different source ports. If `start` is high it means a new packet comes in, immediately the free block will allocate an ID for this source port

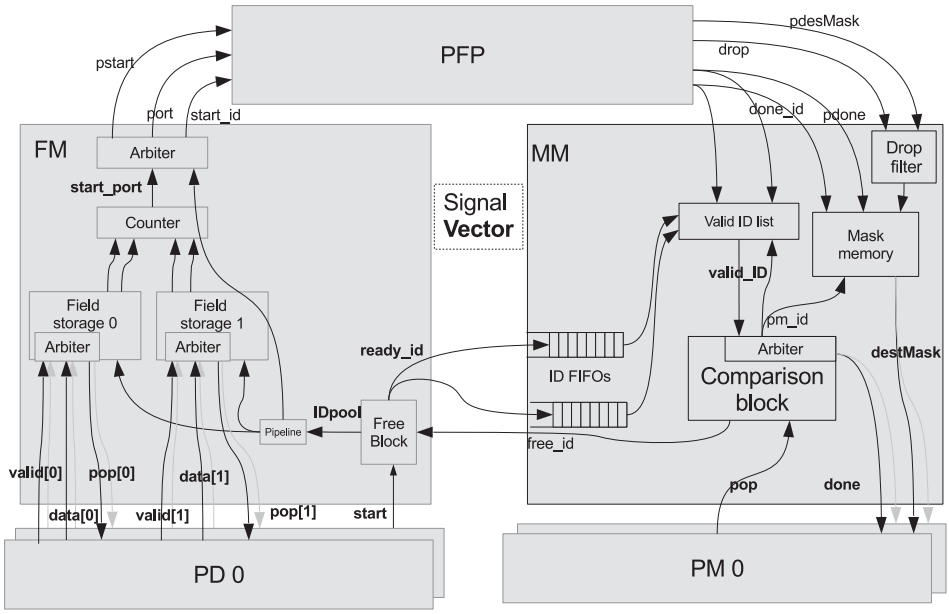


Figure 3.9: Interface of Field Memory and Modifier Memory

packet, and also send this ID to the relevant FIFO in MM for synchronization with `pkt` interface.

In a single field storage block, the arbiter chooses a field from different PDs every clock cycle if they have valid signals and stores the corresponding field data in the memory based on the allocated ID. The counter block counts each port's field status separately, if there is a port with all fields stored, an inner start bit for this port will be sent and the final arbitration will be done between inner start bit for all ports before activating the PFP.

Considering the interface on the packet modification side, when the PFP sends done signal (`pdone`), relevant ID in the valid ID list will be pulled high. Meanwhile,

the destination port mask (`pdesMask`) will be stored in the mask memory. If the decision turns to be dropping the packet, 0 will be stored in the corresponding entry. Secondly, the first ID to be popped in each ID FIFO will check its valid status from the valid ID list. If the valid bit is high and the PM is available for processing, this ID will be marked as an valid ID. In the comparison block, all valid IDs are arbitrated and one of them will be selected as `pm_id`, its valid bit in the valid ID list will be reset to low. Then there comes the interface between the PM and the MM, corresponding port numbers will be found for `pm_id`, meanwhile forwarding decision is sent to the PM via `done`,`destMask`. As soon as the PM send `pop` to the MM, `done` for this port will be reset, and `pm_id` is deallocated to the free block in the FM.

3.1.4 Configurable Memory Interface

Figure 3.10 is the interface of tables, or Random Access Memory (RAM)s, which communicate with the PFP and the CPU. A lookup stage in the packet pipeline contains two data paths for hit and miss respectively. While reading, a RAM always return the data of the relevant entry hence no miss branch in a RAM read stage. The tables/RAMs here are called configurable memories since they consist of a normal RAM and a arbitration wrapper.

All lookup stages, RAM read stages and RAM write stages should go through an arbitrator before interacting with tables/RAMs. Access to the same table/RAM from different pipeline branches at the same clock cycle could be an issue due to the property of forward processing. The packet pipeline cannot be stalled to wait for the preparation of a certain stage. If multiple stages need to occupy the same resources

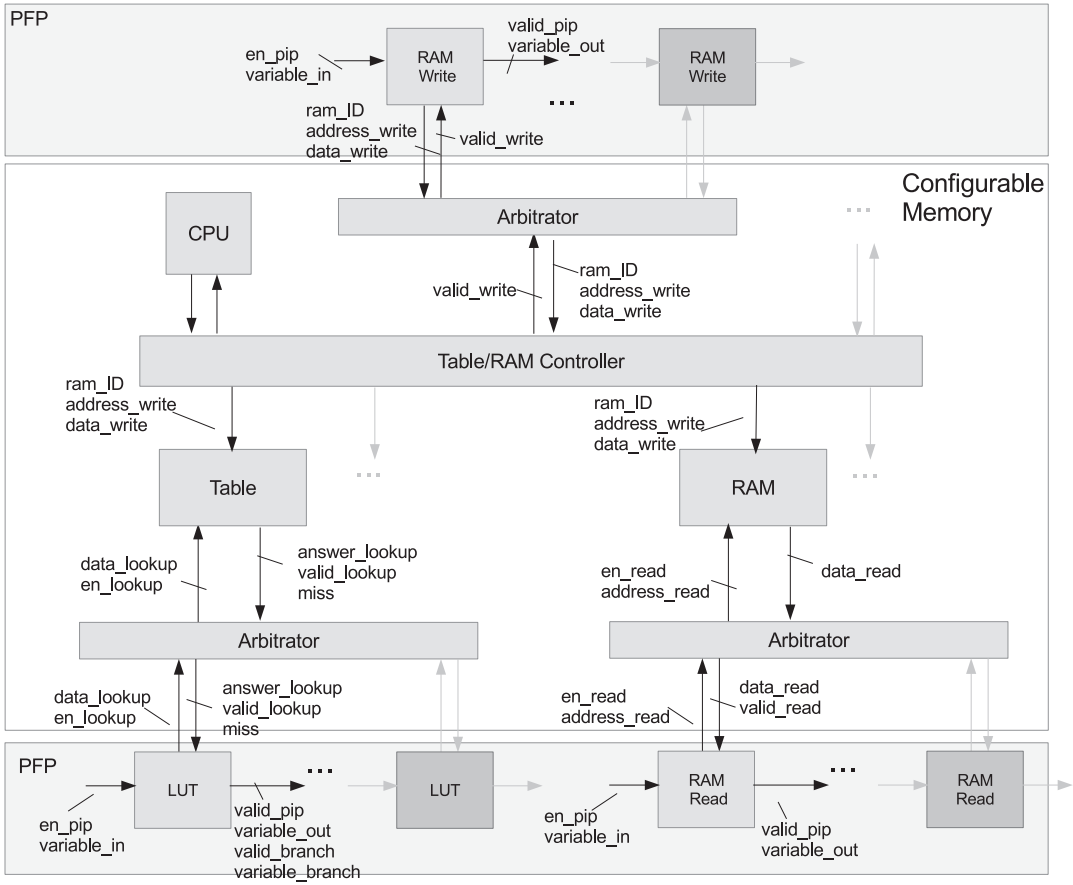


Figure 3.10: Interface Between Tables and PFP

(e.g. in our structure multiple stages send access requests to tables/RAMs, or several branches merge to the same destination), only one stage could be executed while others stop pushing data to next stages but keep receiving data from previous stages.

In fact, for each configurable memory there are two levels of arbitration. Level one is between CPU access and PFP access and level two is for PFP accesses which arbitrate requests from different pipeline stages. In this proposal, we have decided to set that CPU access has the highest priority which means before start packet processing, there will be a table initialization phase. For accesses from different stages, we use round robin selection.

Finally, Figure 3.11 shows an overall view of the hardware infrastructures (configurable memory is not included). Since the interfaces between PFP and hardware infrastructures are fixed and all these blocks are parametrized, there is no need to modify their structures to adapt different PFP implementations. In the next section, it will be explained how to generate the PFP RTL automatically from assembly program.

3.2 Packet Forwarding Pipeline

As we know, the auto-generated packet forwarding hardware structure is based on assembly codes which are compiled from PPL programs and the instructions in the assembly program will be executed in a sequential order. To reflect this property in hardware, a fully pipelined structure could be applied.

Figure 3.12 is the basic structure of the packet pipeline. It looks like a pipelined multiprocessor structure where each processor executes a dedicated function on the

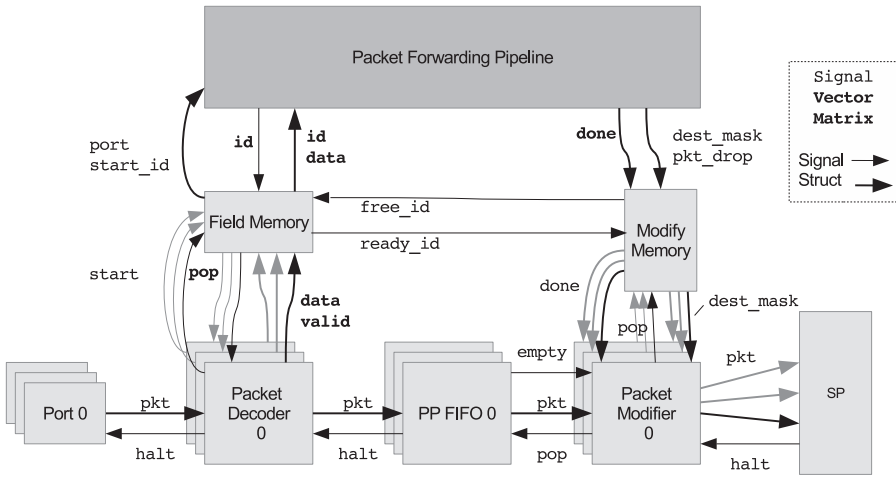


Figure 3.11: Overall View of Hardware Infrastructures

incoming data and forwards the result to the next core in the pipeline. In fact, the function of each core is dedicated for every implementation. That is to say, in a pipelined multiprocessor structure new instruction could be reloaded to a instruction core. However, in this structure each core positioned in the pipeline has a fixed function which is determined by the automatically generated assembly program. Based on this tool, the structure in Figure 3.12 is actually implemented as in Figure 3.13. This is an example structure generated by an assembly program containing 20 instruction sets which have an execution order: instruction number 4,3,3...20,1,7. Correspondingly, all the required instructions have their own RTL blocks in the RTL library.

The advantage of the high regularity packet forwarding is that we can develop the instruction sets particularly to cover all the executions we need, and give each instruction its own hardware structure.

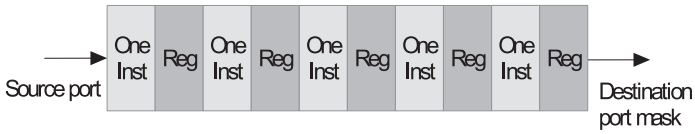


Figure 3.12: Structure of packet forwarding pipeline

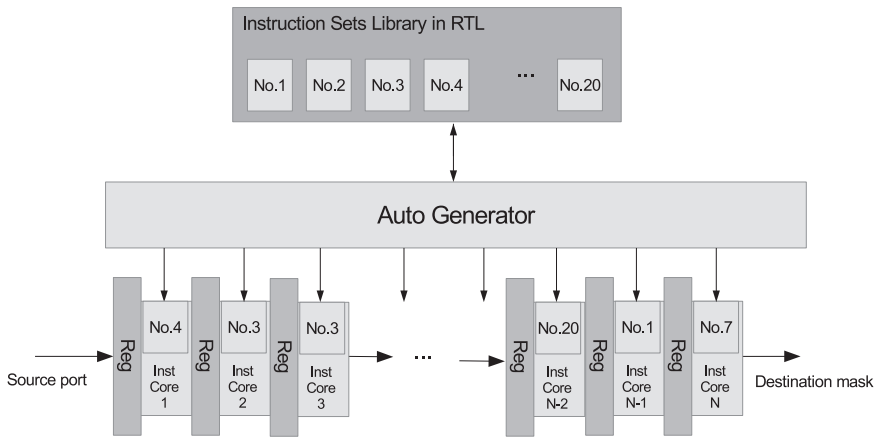


Figure 3.13: Example of one packet forwarding pipeline implementation

3.2.1 Instruction Set

The instructions referred to as packet forwarding modules can be classified into several categories as follows:

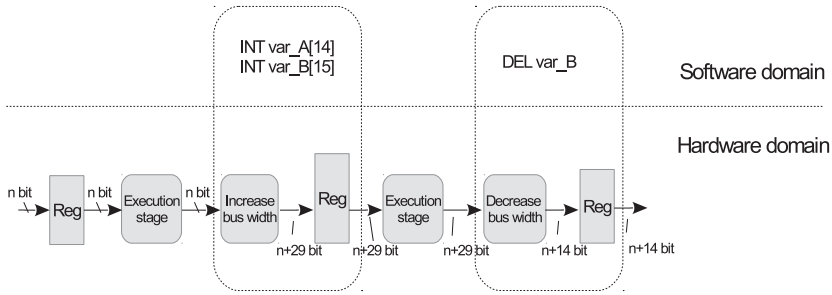


Figure 3.14: Example of creation and deletion of variables

Variable Instruction

The packet forwarding pipeline contains two parts, register stages for variable management, and execution stage for functionalities.

Normally the initial input to the PFP is only the source port, but during processing, some execution stages require data from packets or tables. Thus, the register stage is not only inserting flip flops but also handling various required data for the next execution stage.

Three kind of instructions are created for variable management:

- INT <*variable name*> [*variable width*]: create a new variable.
- DEL <*variable name*>: delete an unused variable.
- MOV <*destination operand*>, <*source operand*>: the destination operand is a variable, but the source operand could be a variable or a number, if the two operands have different width, the overflow of higher bits will be unused.

Figure 3.14 shows how to manage variables in the pipeline. All instructions are

mapped to execution stages in the PFP except variable instructions. `INT` instructions and `DEL` instructions are in the register stages increasing or decreasing the bus width respectively.

Arithmetic Logical Unit Instruction

- ADD/SUB/MUL/DIV <*destination operand*>, <*source operand 1*>, <*source operand 2*>: add/subtract/multiply/divide two variables together, the result is stored in the destination operand.

- XOR/NOT/AND/OR <*destination operand*>, <*source operand 1*>, <*source operand 2*>: boolean operations of two source operands.

Similar with MOV instruction, the destination operand of Arithmetic Logical Unit (ALU) instructions should be a variable while the source operands could be variables or numbers.

Lookup Table Instruction

This category contains instruction for table lookups.

- LKUP <*table name*> <*source operand*> <*destination operand*> : the source operand is the lookup address, a lookup is based on the table name and the result will be stored in the destination operand.

According to the PPL in this project, all tables that will be accessed in the main function have to be defined in the header file in advance. As a special data structure in the PPL, a table definition contains width and depth of the table, as well as

the default values. Since the main function in the PPL program passes through an RTL generation flow and the header file is used for the hardware infrastructure instantiation, table lookup instructions will occupy additional interfaces to tables outside the PFP.

Different from previous instructions, a table lookup instruction is a time-consuming module in hardware. After sending the lookup address, this request will be firstly accepted by an arbitrator, then the arbitrator will communicate with referred tables. Hence, between lookup stages and tables is a handshake interface which has been demonstrated in Figure 3.10.

Normally after a lookup stage, the data-path in the PFP will be divided in two directions based on the lookup result like in Figure 3.15, so flow control instructions will be needed.

Branch Instruction

The basic structure in Figure 3.12 is a single data-path pipeline. However, in a real situation there might be lots of branches since the decisions could be different in certain stages. Thus multiple data-paths should exist in the packet forwarding including data-path control functionalities which makes RTL generation a bit more complicated. However, the branch instruction of packet forwarding has several properties which reduce the complexity of the hardware structure.

Most importantly, all branches in the PFP are forward branches, that means a branch cannot jump to a previous stage. This is quite obvious since the instructions will be implemented directly with a hardware module and there is no memory to store

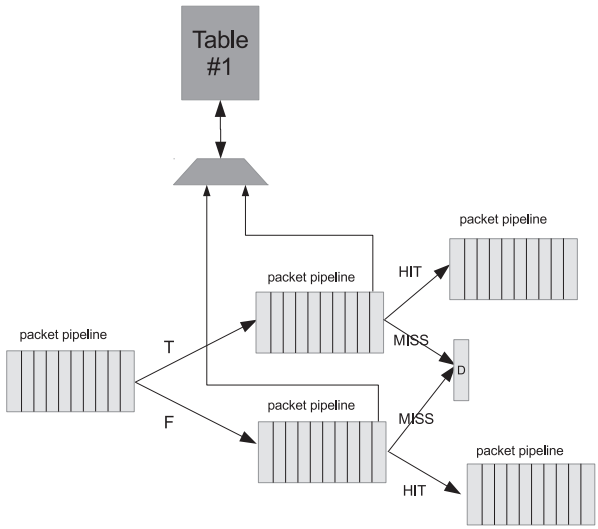


Figure 3.15: Example of Branch in Packet Forwarding

program counters to call for a previous stage.

Secondly, although branches in the packet forwarding program make the PFP spread to different paths, finally the entire program is a spindle structure and ended either in drop the packet or send the packet to a number of ports. If we draw out the longest processing path as the main data-path, all other branches will jump to this master branch after certain processing except the special `DROP` instruction which will drop a packet and stop further processing.

Currently five kind of branch instructions are supported.

- BEQ/BNE <*source operand 1*> <*source operand 2*> <*destination label*>: branch to the destination label if source operand 1 and source operand 2 are equal/not equal.

- BGT/BST <*source operand 1*> <*source operand 2*> <*destination label*>: branch to the destination label if source operand 1 is greater/smaller than source operand 2.

- JMP <*destination label*>: unconditional branch, jump to the destination label.

Here, the source operand could be both variables and numbers. Inside the branch instruction there is an ALU unit to make the decision.

Packet Instruction

Besides the above instructions, there are some instructions that operate on the packet data.

- PRD <*source operand 1*> <*source operand 2*> <*destination operand*>: packet read instruction, source operand 1 is the start bit and source operand 2 is the end bit, the read data will be stored in destination operand.

- DROP: this is a instruction without operand, it means a stage that a packet shall be dropped and not processed any further.

- SEND <*destination operand*>: this is the last instruction if there is no packet drop, the destination operand is the destination port mask for the processing packet.

The packet read instruction is responsible for creating interface between the PFP and input packets. For each packet read instruction, there is a Finite State Machine (FSM) inside PD which belongs to packet processing infrastructures. Instruction

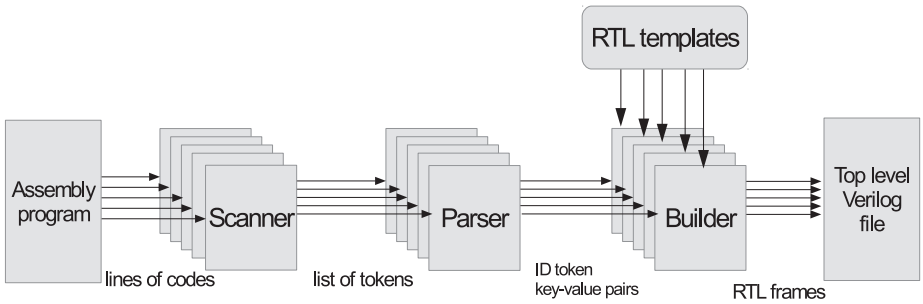


Figure 3.16: Steps of Auto RTL Generation

`SEND` is the ending instruction in a PFP, it contains the destination port mask which will be sent to the packet buffering subsystem

3.2.2 RTL Generator

The RTL generator is brought up as a Python script and basically consists of three parts: a scanner, a parser and a builder. Figure 3.16 explains generation steps of the RTL generator, it takes assembly program as an input and outputs a Verilog module which has fixed interface in order to be instantiated in an RTL project. As a primary step, the assembly program is scanned and each instruction gets its own token list. Secondly, every token list will be parsed by a parser which finds the keywords (referred to module parameters in the hardware view) with its corresponding values. Thirdly, these key-value pairs are transferred to the builder with an opcode which are used to identify different RTL structures. The builder invokes RTL templates according to ID tokens and do a multiple substitution based on key-value pairs. Finally, the output file assembles all RTL together to form a packet forwarding module which has

instantiation of hardware modules and wire connections.

Scanner

The scanner in the design is based on SPARK which is a toolkit for implementing domain-specific languages in Python[9]. It does a simple lexical analysis based on regular expressions and generates a token list for each instruction[1]. A token has two attributes, type and value. New kind of tokens could be defined with the increment of complexity. Since the processing in this generator is based on tokens, compatibility of the tool is easily promised.

If a line starts with a white space or a hash key, then it only generates an empty token list. However, for each non-empty token list, there always exists one token that can describe the behaviour of that line, we call it the ID token. Normally, all the opcodes are ID tokens. In addition, if an in-line code describes table definition, or is marked as a branch label, the ID token will be the name of the table or label. ID tokens are important since they will be used later for invoking RTL modules.

Parser

The parser is used for transferring information from software domain to hardware domain. The scanner provides plenty of tokens representing information in a assembly program. However, the tokens are not enough to reveal hardware behaviours. For instance, the sequential scanning provides all the operations and variables, but how to interact variables in different token lists? In a CPU instruction perspective, variables

[1]List is a built-in Python data structure used as a container that holds a serial of objects in a given order.

Table 3.1: Variable File Management Unit

List Name	List Item			
	index 0	index 1	index 2	...
variable name	V_a	V_b	...	...
start bit	0	6	9	...
variable width	6	3	...	...

are stored in cache memories with unique addresses while in a RTL design they are stored in registers and delivered to other logic gates by wire connections. In order to keep identical variables in the assembly problem and the RTL program, a variable management unit called Variable File (VF) is introduced to take charge of the variables above the sequential line scan.

The maintenance of the VF occurs when the opcode is `INT` or `DEL` and referred to three lists shown in Table 3.1.

As listed in the table (Table 3.1), `variable name`, `start bit` and `width` are assigned with the definition of every new variable. When a new variable is defined in the assembly program, these three attributes are stored simultaneously. The idea of the VF is mapping variables from the software domain to the hardware domain. In the hardware domain, the VF is represented as registers with the same sum-up bit width of all variables. Each software variable is referred to the bit extraction of a hardware signal while start bit and data width assist to locate the field of a software variable in the hardware wire.

After adding one particular variable (refers to `INT` instruction in the assembly program) and if it is needed somewhere else in the assembly program, the RTL generator will match the variable name in the `variable name` list and translate it to information with start bit and width. If the opcode is `DEL`, then the variable name,

the corresponding start bit and the width are deleted in the VF together.

In a PFP RTL module, the outputs of the previous stage are used as the inputs to the next stage. In this case, the wire width varies between input and output which reflects the functionality of the VF in the RTL generator. For each instruction, there is a corresponding RTL frame that includes one RTL block instantiation and one or more wire declarations. The RTL frame has a fixed hardware interface for the same opcode but parameter varies which is the interface for software control. The parameters in the hardware domain are extracted as keywords from tokens in the software domain (e.g. the start bit and the variable width). The RTL block template for each opcode contains all the possible keywords wrapped with `$`. Listed below is a RTL sample frame for opcode `INT`, which is used for adding a new variable:

```
wire [$owidth$-1:0] pip_$stage$;
BUF #(.iwidth($iwidth$),
      .owidth($owidth$))
BUF_$stage$ (.iv(pip_$stage-1$),
             .ov(pip_$stage$),
             .clk(clk),
             .rstn(rstn));
```

Here `$iwidth$` is the total width of the old VF while `$owidth$` is the total width of the VF after adding the new variable. `$stage$` counts for the identification of different registers between stages and it is controlled by a counter and is updated for every instantiation. In the RTL generator, all keywords are put in a Python

dictionary data structure[2] as keys and the value of these keys will be determined by the parser. The key-value pairs are sent to the RTL builder accompanied by ID tokens (opcodes) after been parsed.

In terms of a hardware perspective, it is a wire connection issue when there is a branch instruction. The additional branch wire of the output should be connected to the destination label which is implemented by an arbitrator that receives all the incoming branches and connects to further pipeline stages. From the software perspective, this is a copy issue about the VF and the wire numbers. When the scanner finds start position of the branch destination, all the branches that jump to this label will be gathered and their wires to the input of the arbitrator in this stage will be connected. Meanwhile, the VF for this destination stage need to be substituted with the copy from the branch stage.

While dealing with branches in the parser of the RTL generator, two dictionaries are designed to manage the wire connections and the VF copies respectively. The branch labels in the assembly program will be used as keys of the dictionaries. If there is a join stage with a label, the previous wire and the VF required here could be searched in the dictionaries based on the label name.

As discussed in the hardware part, several branches might jump to the same place. As a result, in the label dictionary one key can have more than one value. In simple terms, the arbitrator will manage the selection of multiple copies of the VFs that are merged from different branches. However, branch management in hardware is not flexible enough and large buffers are required for synchronizing the latencies since the processing time of each branch cannot be estimated in advance. This drawback

[2]Dictionary is a built-in Python data structure that maps hashable values to arbitrary objects

brings hardware waste and will be discussed in the next chapter.

Management of tables is similar to branches as they all belong to multiple access issues. Since the PFP has a forwarding structure, as long as a destination label exists, a relevant hardware join stage should be added. According to the forwarding principle, all branch instructions should be executed before their join stages and the follow-up branch instructions cannot jump back to a previous join stage. However, in a table access case the number of table access requests cannot be determined until the last instruction. Hence the parser will not map table access tokens to its corresponding table definitions tokens until token `End`, which means when the end of the program is detected. As an initial step, one table dictionary is used to store the information of table connections and when the PFP RTL module ends there will be a separate table connection unit per table for different table access.

Builder

The final part of the RTL generator is used to assemble the PFP RTL modules. According to the parser, for each valid line of the assembly program there will be an ID token with dictionary. If the switch complexity grows and a new instruction set is needed, a new ID token and relevant RTL sample frame will be defined for it. As explained in Parser subsection, a RTL sample frame consists of an ordinary RTL block as well as keywords wrapped with `$`. Multiple substitution function is applied to search all the keywords in the sample frame, then substitute with the values in the dictionary. Table 3.2 shows the components of the RTL builder's output. According to different ID tokens, it is divided into four parts and assembled together to generate

Table 3.2: Components Created by RTL Builder

ID Token	Element	
"Start"	Header	Include files
		Parameter declaration
		IO declaration
		Default wire declaration
		Default packet buffer instantiation
"Table"	Table instantiation	
"Opcode" & "Branch"	Pipeline stages	RTL block instantiation
		Wire connection
"End"	Tail	Table connection
		End module

a PFP RTL module.

Once the PFP RTL module is generated, it will be embedded with other hardware infrastructures as stated in Section3.1 to finally make up a complete packet processing subsystem.

Chapter 4

Implementation Results

During the design of the hardware, the developing tool we use is MyHDL which is a package turning Python into a hardware description and verification language. One of the most significant advantages of using MyHDL is the incredible interface management ability [10]. By the aid of Python data structures, list of signals or even list of list of signals can be defined as a single data structure directly and MyHDL will split them into a cluster of signals automatically while generating RTL. This property provides the possibility to simplify the interfaces between hardware modules during the design process. Furthermore, MyHDL offers a software testing method to test hardware. The test-bench created by MyHDL can interact with Python directly and hence various self tests and random tests can be created. However, since MyHDL is not a mature language, it lacks some features also. What bothers us most is that the generated RTL will remove all hierarchies and only one flat top file is left. It makes hardware debugging more difficult than hierarchical RTL designs and pushes heavy workload to place and route during synthesis.

Table 4.1: Packet decoder ports

Name	Direction	Description
idata[63:0]	In	Packet data from MAC
ivalid_bytes[3:0]	In	Packet valid bytes from MAC
ifirst	In	Packet first flag from MAC
ilast	In	Packet last flag from MAC
odata[63:0]	Out	Packet data to packet FIFO
ovalid_bytes[3:0]	Out	Packet valid bytes to packet FIFO
ofirst	Out	Packet first flag to packet FIFO
olast	Out	Packet last flag to packet FIFO
ppp_start	Out	Port valid flag to FM
fdone[field] (one signal per filed)	Out	Field done flag to FM
fvalid[field] (one signal per field)	Out	Field valid flags to FM
fdata[field][x:0] (one signal per field)	Out	Field data to FM
fpop[field] (one signal per field)	In	Acknowledge signals from FM to indicate the field data is accepted
ihalt	In	Transmission halt signal from packet FIFO
ohalt	Out	Transmission halt signal to MAC
clk	In	System clock
rstn	In	Global reset
pp_params	Parameter	A Python dictionary containing all needed parameters

In this chapter, hardware schematics are introduced and synthesis results are presented. Finally, pros and cons of this design proposal are discussed.

4.1 Hardware Infrastructure Schematic

Figure 4.1 is the schematic of one PD and the ports are shown in Table 4.1. The structure of each PD mainly differs based on two parameters: the number of fields (`nf` in the figure) and the length of each field (`x` in the figure). The `Field FSM` is instantiated for each field that need to be extracted. The input `frange` is not a signal but a parameter indicating the start bit and end bit of this field in the packet.

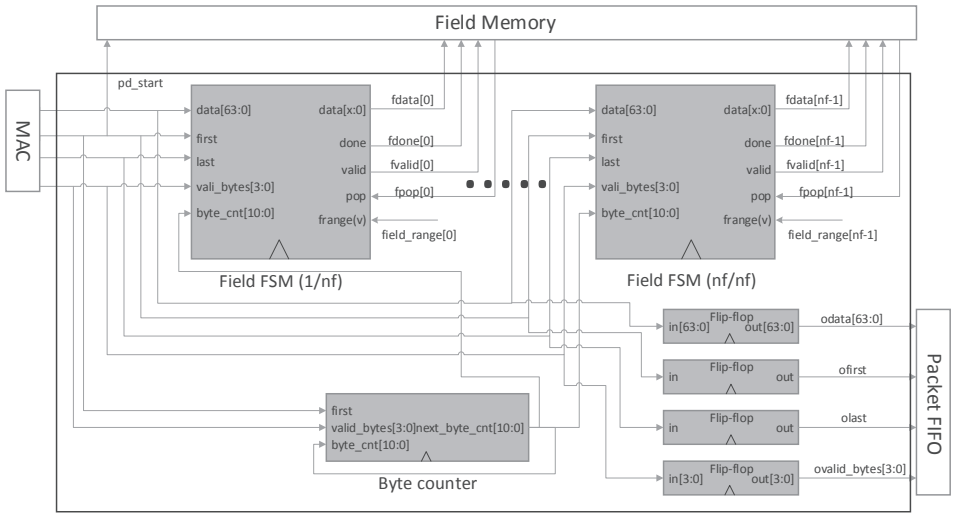


Figure 4.1: Schematic of Packet Decoder

MyHDL allows diverse parameters as inputs to a hardware module which makes architecture configuration extremely flexible. Figure 4.2 and Figure 4.3 are schematics of the FM and the MM respectively. These two modules bridge the PFP and packet transmission modules and their ports are listed in Table 4.2 and Table 4.3.

Compulsory parameters in the FM and the MM are the number of ports (`np`), the number of fields (`nf`) and the number of ids (`nid`). Starting with the FM, each port with a new packet coming in will be allocated an ID from the `free` block and it will not be deallocated until the decision of this packet reaches the PM. In this case, multiple packets from the same source port can be processed simultaneously in the PFP without collision. In the FM, the `Field Storage` is instantiated per field, a RAM is encapsulated in each `Field storage` with all IDs as the entries. For each source port, there is a `Port counter` counting the fields that have been written for the current port ID and the PFP will not start processing this ID until all fields are

Table 4.2: Field memory ports

Name	Direction	Description
pd_start[port][63:0] (one signal per port)	In	New packet ready flag from PD
fvalid[port][field] (one signal per field per port)	In	Field valid flag from PD
fdata[port][field][x:0] (one signal per field per port)	In	Field data from PD
fpop[port][field] (one signal per field per port)	Out	Acknowledge signals to PD indicating the field data is accepted
srcport[np_w-1:0]	Out	Source port to PFP
start_id[3:0]	Out	ID of the source port to PFP
pdata[field][x:0] (one signal per field)	Out	Field data to PFP
pstart	Out	Start flag to PFP
pid[field] (one signal per field)	In	Field storage read address from PFP
start_push[port] (one signal per port)	Out	Push signal to the FIFO in MM
port_id[port][3:0] (one signal per port)	Out	Data to the FIFO in MM
free_push	In	Push signal from MM to the free block in FM
free_id	In	Data from MM to the free block in FM
clk	In	System clock
rstn	In	Global reset
pp_params	Parameter	A Python dictionary containing all needed parameters

written. In both ingress and egress of the FM, two (`Round Robin (RR) selector`) exist for arbitrating between available ports since in one clock cycle only one ID can write to a certain `Field storage` RAM and the PFP can only process sequentially.

In the MM, there are ID FIFOs for each source port to receive the IDs that have been allocated for it. Although the order of packets for one port can be reversed in the PFP due to various execution time cost, the sequence of the packets will be sorted again in the MM. Those packet IDs that get decisions before the earlier sending IDs,

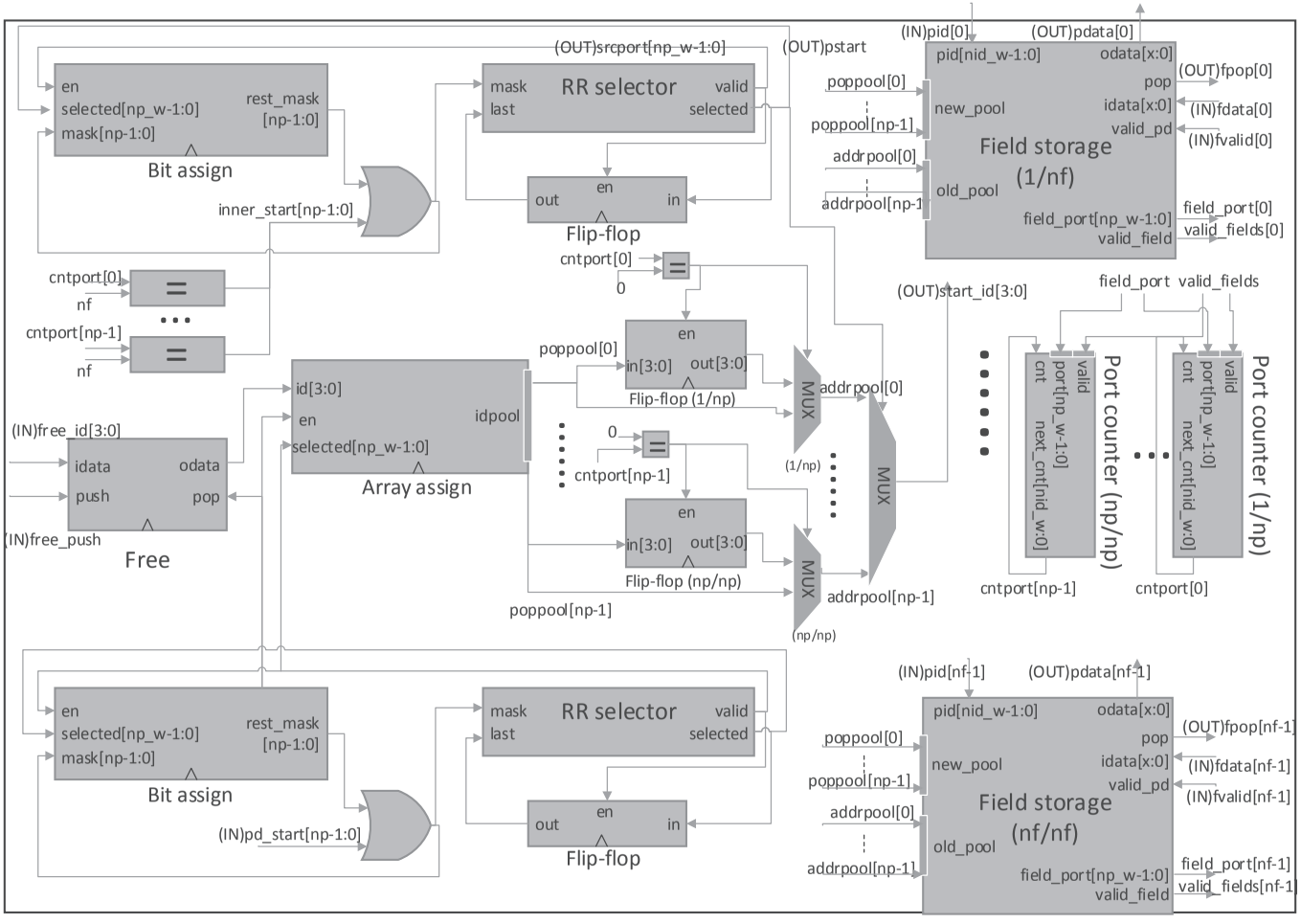


Figure 4.2: Schematic of Field Memory

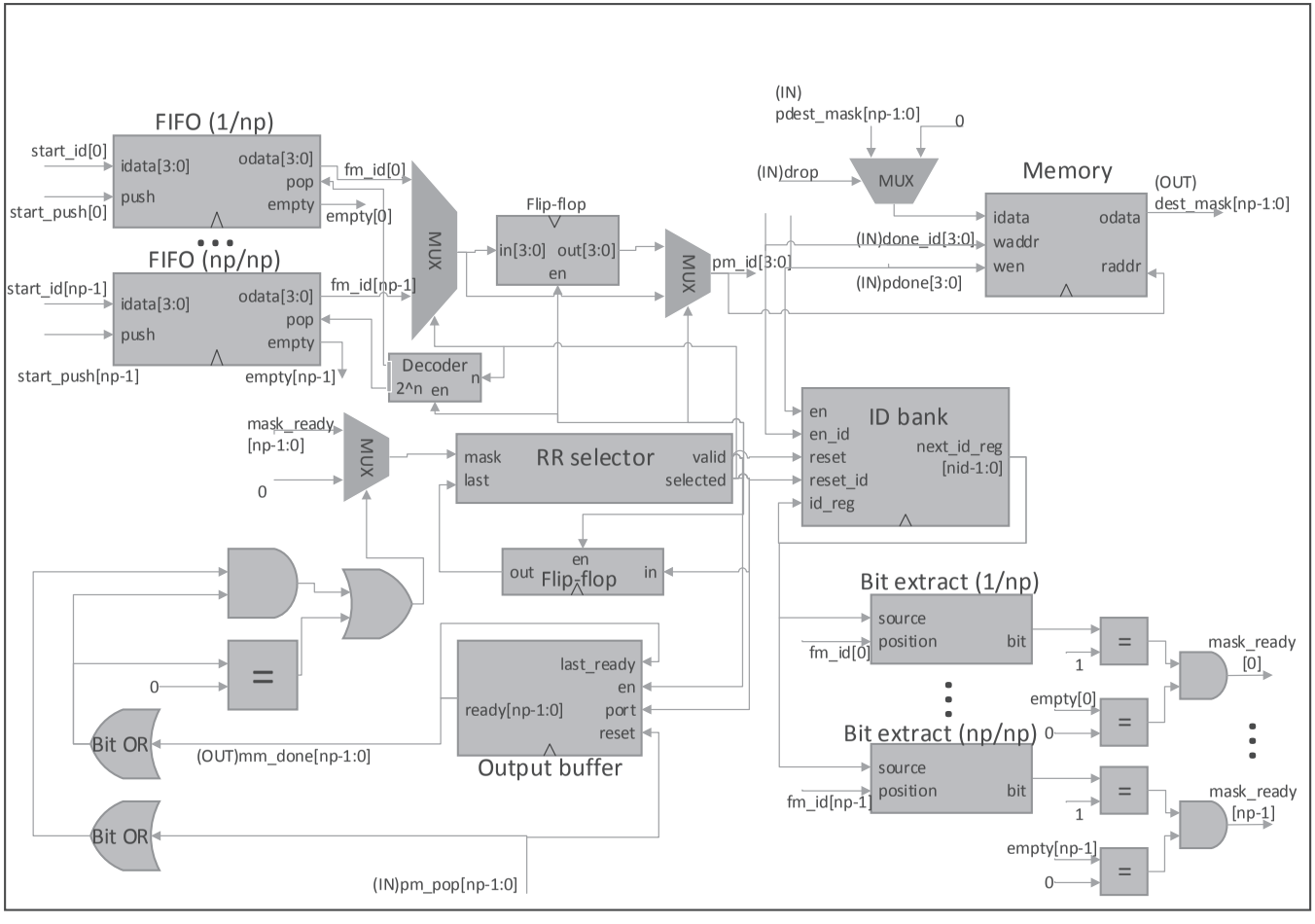


Figure 4.3: Schematic of Modification Memory

Table 4.3: Modification memory ports

Name	Direction	Description
pdone	In	Valid flag from PFP
done_id	In	ID of source port from PFP
pdest_mask[np-1:0]	In	Destination port mask from PFP
drop	In	Packet drop flag from PFP
start_push[port] (one signal per port)	In	Push signal from FM to the FIFO in MM
start_id[port][3:0] (one signal per port)	In	Data from FM to the FIFO in MM
free_push	Out	Push signal to the free block in FM
free_id	Out	data to the free block in FM
mm_done[port] (one signal per port)	Out	Ready signal to FM
dest_mask[port][np-1:0] (one signal per port)	Out	destination port mask to FM
pm_pop[port] (one signal per port)	In	Acknowledge signal from PM indicating the request is accepted
clk	In	System clock
rstn	In	Global reset
pp_params	Parameter	A Python dictionary containing all needed parameters

are stored in the `ID bank` and waiting for the ID FIFO to pop it. In the egress of the MM, there is a buffer stage (`Output buffer`) which is enabled by the pop signals (`pm_pop`) from the PM. The PM has a schematic in Figure 4.4 with ports defined in Table 4.2, once a source port's PM knows the decision is available from the MM and its packet FIFO is not empty, the destination port mask will be popped to the PM while the first chunk of the packet is popped from the packet FIFO. The MM will not be accessed again until the current packet has been sent through the PM entirely.

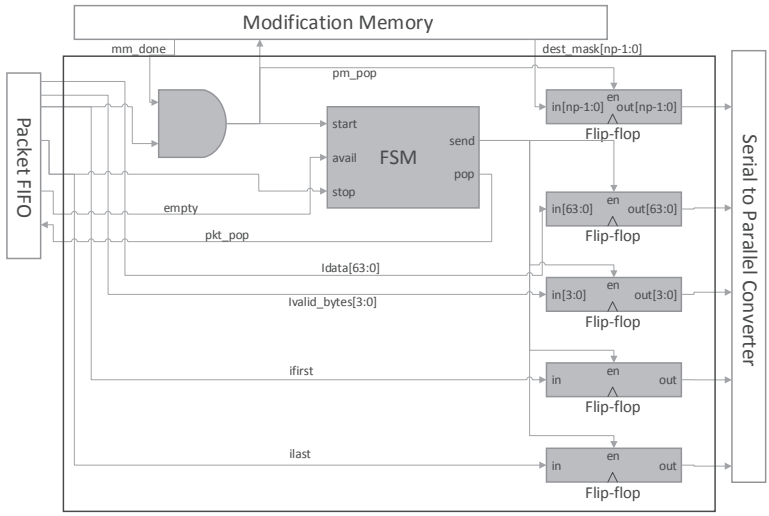


Figure 4.4: Schematic of Packet Modifier

4.2 Testing Environment

In this section, a real testing environment is briefly introduced. With the aim of testing the generated switch, it should be put in a wrapper to map the pins on board and several external blocks are required to establish the testing enviroment. The testing connection and all on board blocks are shown in Figure 4.5.

In the FPGA, a 10Gbps serial single channel PHY gives a direct connection to Enhanced Small Form-factor Pluggable (SFP+) optical module which is used to connect different 10G devices. On top of it, several kind of bus interfaces are involved from PHY to the generated switch core. The outermost data transmission interface is a Ten Gigabit Media Independent Interface (XGMII) between Xilinx 10-Gigabit MAC and a 10Gbps-capable PHY provided by Xilinx 10-Gigabit Ethernet Physical Coding Sublayer (PCS)/Physical Medium Attachment (PMA) core. On the switch

Table 4.4: Packet modifier ports

Name	Direction	Description
idata[63:0]	In	Packet data from packet FIFO
ivalid_bytes[3:0]	In	Packet valid bytes from packet FIFO
ifirst	In	New packet flag from packet FIFO
ilast	In	Packet end flag from packet FIFO
odata[63:0]	Out	Packet data to SP
ovalid_bytes[3:0]	Out	Packet valid bytes to SP
ofirst	Out	New packet flag to SP
olast	Out	Packet end flag to SP
empty	In	Empty flag from packet FIFO
pkt_pop	Out	Pop signal to packet FIFO
mm_done	In	Ready signal from MM
pm_pop	Out	Acknowledge signal from PM indicating the request is accepted
dest_mask[np-1:0]	In	Destination port mask from MM
ihalt	In	Transmission halt signal from SP
clk	In	System clock
rstn	In	Global reset
pp_params	Parameter	A Python dictionary containing all needed parameters

core side, Advanced Extensible Interface 4 (AXI4) protocol gives transmit and receive interfaces between MACs and the switch core. Finally, AXI4-Stream is translated to our `pkt` interface with a AXI4 bridge on both transmit and receive side. In addition, a asynchronous region is attached to transfer data to the 105 MHz core clock domain since Xilinx MAC is working under 156 MHz.

Ideally, another Peripheral Component Interconnect Express (PCIe) interface is required to support communication to the switch. However, it turns out to be a long term task due to its complexity. As a temporary solution, Vivado logic analyzer offers an option to use Virtual Input/Output (VIO) instead. This is a customizable core that can both monitor and drive internal FPGA signals in real time. Once the bit stream is loaded to the board, all tables could be initialized using VIO as well as

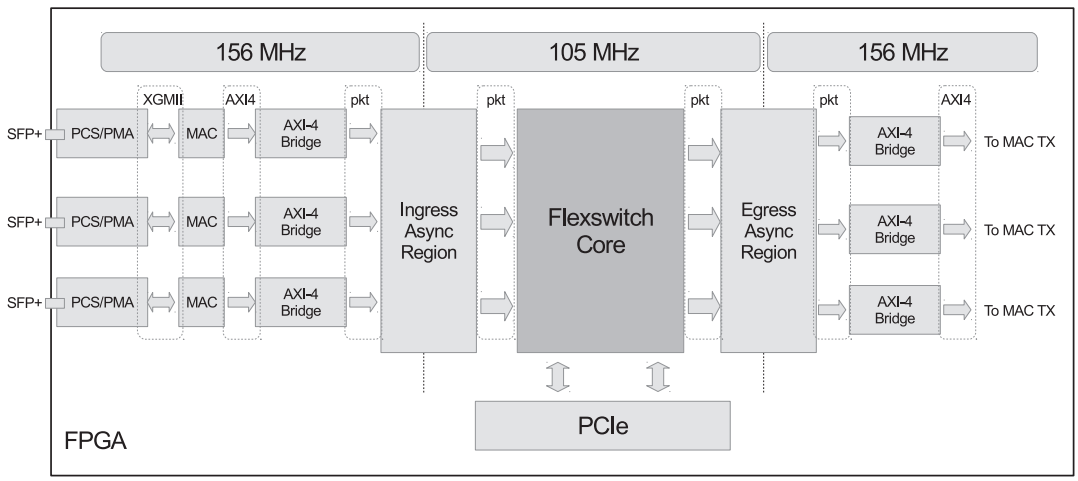


Figure 4.5: Testing Connections on Board

monitoring drop counters, FIFO status, etc.

4.3 Synthesis Result

As the primary target of this project, the tool chain we have developed can generate completely different hardware designs. The functionality of the generated hardware is including but not limited to a Layer 2 switch with forwarding algorithm as in Figure 2.5. After completing the development of tool chain, a 2-port and a 6-port Layer 2 switch are generated and synthesised with Xilinx Vivado Design Suit. Below the results of these two configurations are presented. The timing requirement is a big challenge especially when the number of ports increases and lots of work has been done in infrastructures of the packet processing subsystem to finally push critical

Table 4.5: Maximum frequency for two configurations

	2 port	6 port
Maximum frequency	129.655MHz	106.205MHz

paths to the packet buffering subsystem where it is supposed to be. As in Table 4.5 both the 2-port switch and 6-port meet the 105MHz timing requirement.

The power summary of the 2-port design and the 6-port design is in Table 4.6. The key factors affecting power consumption can be categorized as statical or dynamic. The static power consumption is due to leakage current when the FPGA is powered on while dynamic power is consumed during operations on FPGA transistors. Static power consumption is mainly determined by semiconductor process parameters and the voltage of the power supply, hence we can see from Table 4.6 that device static power is similar between 2-port design and 6-port design. Dynamic power consumption can be modelled as Equation 4.1 which models dynamic power with the productive switching activity of the transistors (α), clock frequency, total capacitance and the square of the voltage supply.

$$Dynamicpower = \alpha \cdot f_{clk} \cdot C_L \cdot V_{DD}^2. \tag{4.1}$$

Typically, the last three parameters are fixed so the switching activity is the key point between different implementations.

From Table 4.7 and Table 4.8 detailed on-chip components are listed with power consumption as well as device utilization. Generally, there is a linear relation between power consumption and utilization for each component. It should be reflected by resource occupations between the two designs that most of hardware infrastructures are instantiated per port. Obviously from the tables, the number of transceivers

Table 4.6: Power summary for two configurations

	2 port	6 port
Total On-Chip Power (W)	1.955	4.030
Dynamic (W)	1.578	3.622
Device Static (W)	0.377	0.408

(GTH) is equal to the number of ports therefore the power of this component in the 6-port design is just about three times than the 2-port design. Meanwhile, utilization of slice logics in the 6-port switch is slightly less than three times of 2-port switch, this means between these two implementations, some shared slice logics exist so that not all slice logic resources need to be duplicated. Furthermore, we can notice that the occupation of block RAMs only increases 30% in the 6-port design which is on account of the packet buffering subsystem owning most of the block RAM resources. Most of memories used in packet processing subsystem are synthesized as distributed RAMs to save area, only those large memories will be synthesized to block RAMs to get better performance and routing results. Since packet buffering subsystem is a single block shared by all ports, it will not be affected by port number configurations. The MMCM in the table is Xilinx mixed-mode clock manager used to generate different clocks from system clock. There are two MMCMs in the design, one for generating clock for Xilinx MACs and one for generating 105MHz clock for the switch core.

All in all, the total on-chip power is doubled while the number of ports is tripled which demonstrate that the resources can be shared partly between various configurations.

Table 4.7: On-chip components power report for a 2-port Layer 2 switch

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.414	18	—	—
Slice Logic	0.026	152421	—	—
LUT as Logic	0.025	83623	433200	19.30
Register	<0.001	42226	866400	4.87
CARRY4	<0.001	1697	108300	1.56
LUT as Distributed RAM	<0.001	7982	174200	4.58
F7/F8 Muxes	<0.001	1361	433200	0.31
LUT as Shift Register	<0.001	192	174200	0.11
Others	0.000	465	—	—
Signals	0.073	120034	—	—
Block RAM	0.234	100	1470	6.80
MMCM	0.211	2	20	10.00
I/O	0.004	28	600	4.66
GTH	0.616	2	—	—
Static Power	0.377			
Total	1.955			

4.4 Implementation Analysis

We conclude that the packet processing subsystem achieves the initial requirements which is extremely flexible and fully configurable. This tool chain sets all the configurations under software control unlike any hardware reconfiguration which may have redundancy logics. The final generated hardware of this tool chain includes minimum logic units for the required applications. The switch generation process is entirely parametrised and the interface oriented to customer consists of only several parameters like number of ports, the port speed, the forwarding algorithm etc. In spite of some unsupported features in the initial version, it is quite convenient for extension. For instance, the tool chain in this thesis project is designed only for a Layer 2 switch which does not require packet modifications in Layer 3. It is known that the two independent hardware modules PD and PM around PFP can be updated to have

Table 4.8: On-chip components power report for a 6-port Layer 2 switch

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.899	26	—	—
Slice Logic	0.064	391084	—	—
LUT as Logic	0.061	217009	433200	50.09
Register	0.002	100669	866400	11.61
CARRY4	<0.001	5050	108300	4.66
F7/F8 Muxes	<0.001	2641	433200	0.60
LUT as Shift Register	<0.001	564	174200	0.32
LUT as Distributed RAM	<0.001	22124	174200	12.70
Others	0.000	1359	—	—
Signals	0.192	302839	—	—
Block RAM	0.422	132	1470	8.97
MMCM	0.211	2	20	10.00
I/O	0.004	44	600	7.33
GTH	1.830	6	—	—
Static Power	0.408			
Total	4.030			

more compatibility to support more complicated protocols. In a Layer 2 switch the PM is only used to synchronize the PFP and the the packet FIFO (i.e., ensure the packet forwarding decision from the PFP and the packet chunk from the packet FIFO belongs to the same source packet). If a packet modification is required, each part of the tool chain can be updated simultaneously as below:

- In the instruction set, packet overwrite/insert/delete instructions will be added.
- In the RTL generator, new tokens and the corresponding RTL frame will be created for each new instruction.
- In the hardware infrastructure PM, new control logic should be appended to modify packet data from the packet FIFO.

Thanks to the high scalability and flexibility of the Flexswitch platform, it is friendly in maintenance and limited in hardware consumption. However, unfortu-

nately and inevitably, during the development, part of the design is regarded as inappropriate and some are even critical.

First of all, a fully pipelined PFP is not efficient. As we know, normally instruction sets are developed for processor applications. Clock frequency in a processor could be in GHz, which is much faster than the system clock in a hardware switch. For example, in Xilinx 10G MAC [11], the reference clock frequency is 156.25 MHz. The fully pipelined PFP not only leads to an intolerable latency with an increase in complexity of forwarding algorithm, but also duplicates a lot of hardware units that could have been reused. Although it is possible to do some optimization like executing instructions in parallel based on their dependency, finally it will turn to a wire connection issue in hardware. Due to branch control and table access, the RTL generator is already under pressure to cope with the wire mappings. Therefore, it is unlikely that the instruction dependency could be handled easily if more complex connection logics are required.

Increase in the complexity of forwarding algorithm leads to other potential problems. Because of no control flow, all branches relying on hardware arbitrators, relatively simple arbitration mechanism will lead to unnecessary hardware buffers used for synchronizing different branches in a same destination stage. However it is a arduous task to develop a smart arbitration or scheduling mechanism suited for our design.

Other drawbacks lies in hardware infrastructures. As we know, all the existing hardware modules are parametrized. On the one hand, it gains capability for handling various scenarios, while on the other hand most of the hardware infrastructures (PD, packet FIFO and PM)are instantiated per source port. As the port number

grows, these hardware modules are duplicated and the hardware area is multiplied. Meanwhile, the duplicated hardware infrastructure complicates the interfaces between modules and lead to a tough routing task. On all accounts, hardware complexity is required in order to promise flexibility. In our case, we cut off the flexibility based on the physical number of ports and accumulate the complexity as the port amount increases. Instead of this solution, we can try to extract shared logic between different ports as much as possible. Thus, reducing the hardware cost when the number of port increases. For instance, the PD introduced in the design is allocated per port, it provides the possibility to execute sophisticated decoding requirements with respect to each port. However, in most cases one decoding strategy can fit for all ports. Therefore, one PD can be shared between all source ports to reduce hardware area and routing complexity.

Chapter 5

Packet Processing Optimization

Based on the conclusions that we get according to the previous design, it is necessary to find solutions to the problems resulted after the initial design. In principle we hope to do optimizations in two aspects: find a certain way to glue instruction set and hardware architecture closer, as well as limit the increment of hardware infrastructures to a reasonable scope when design configuration raises complexity.

5.1 Intermediate Stage

As summarized in Section 3.3 of Chapter 3, introducing assembly language as the intermediate stage gains some benefit while generating the RTL, but also brings troublesome issues. It is important to focus in detail if there is any better intermediate stage. Basically we expect an intermediate stage that could describe hardware behaviour and include data flow in it. While generating the hardware, the data flow and the possible parallel opcodes are determined so that no further processing is needed to get the reasonable RTL. Inspired by Data Flow Graph (DFG) which shows data

dependency between a number of operations, we found nice properties of it to fit our system. A DFG consists of two parts: execution nodes that refer to instructions in a assembly program and directed edges that show control flow of the system.

DFG provides a valuable short cut to fit the gap between software and hardware. Under this intermediate stage, the RTL generator does not need to care about branch or table access issues anymore. Instead, the compiler is responsible for branch or table access issues. Also the PPL program will be analysed and logical data flow will be sorted from a high altitude. Another advantage is the flexible pipeline strategy, since the initial DFG generated by the compiler does not contain any timing information, there should be a post processing step to change the timeless model to a timing model before sending it to the RTL generator. The additional processing step here is implemented by a DFG optimizer which will estimate delays of execution nodes and insert appropriate flip-flops when parallel execution occurs with different delay. In addition, the DFG optimizer could prejudge area, power consumption of all nodes, under different area or timing requirements, the initial DFG could be rearranged or pipelined through different ways.

Both the initial DFG and the optimized DFG can be described using the GraphViz dot file format [12] although it is possible to pass the data structure directly between compiler and optimizer without going through an intermediate file. With the help of GraphViz, graphs can be described in a simple text language and by default the text file name has a suffix `*.dot`. Considering the basic scenario as an example i.e. node A to node B (Figure 5.1), the two main components of a dot file are node definitions and directed edges. One limitation of this method is that all the execution nodes should have fixed latency, otherwise the optimizer cannot calculate required number

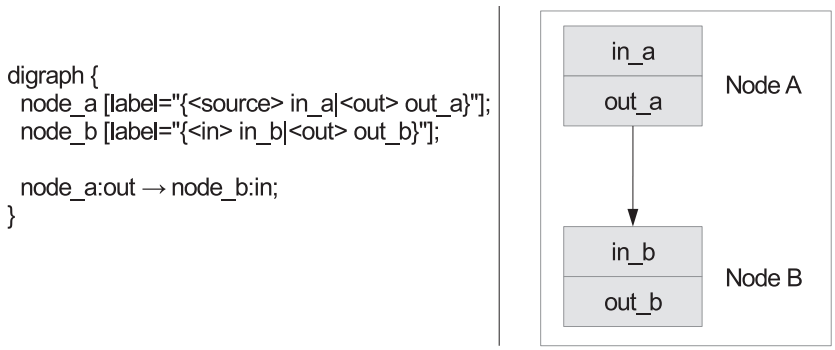


Figure 5.1: Example of a GraphViz dot file

of flip flops after this node. Currently latencies of all nodes that have been developed in packet processing are fixed, the only time consuming node is the lookup node and the latency is one clock cycle, rest of the nodes can be treated as combinatorial nodes.

Thanks to the object oriented RTL generator, it is convenient to transform its input from an assembly program to a dot file. The RTL generator still consists of a scanner, a parser and a builder, but new regular expressions are used in the scanner to capture token lists. Also in the parser and the builder, token lists containing directed edges will be interpreted to signal assignments.

Below is a PPL fragment source code about table lookup with if statement:

```
bit:16 da;

bit:16 tl;

bit:16 x;
```

```
da = pkt.read(0,15);
tl = pkt.read(96,111);
if (tl == 0'h1234)
   x = l2tab[ da ];
else
   x = 0'hffff;
pkt.write(16,31,x);
```

It contains two packet read nodes, one table lookup node, one flow control node and one packet write node. Its DFG format is in Figure 5.2

5.2 Infrastructure around Packet Forwarding

With the aim of reducing infrastructure complexity around the PFP, we decided to give up some adaptability and focus on necessary requirements of a Layer 2 switch. Previously, packet processing subsystem in Figure 3.8 has a separate `pkt` interface apart from the PFP. However, after changing from assembly program to DFG as the intermediate stage, parallel execution in the PFP comes true. Furthermore, even with the lowest pipeline depth, timing constrain is still met in PFP. In a typical Layer 2 switch, the depth is normally less then 10. Due to this fact, using `pkt` interface in 3.8 will not gain much in hardware cost. In the initial design, there is such a deep pipeline in the PFP that if the packet data is passed through PFP pipeline stages then a huge number of registers will be needed. Based on this premise, we separate packet processing flow and packet transmission flow. Once the pipeline depth in the PFP decreases significantly, it is worth integrating the packet transmission in the

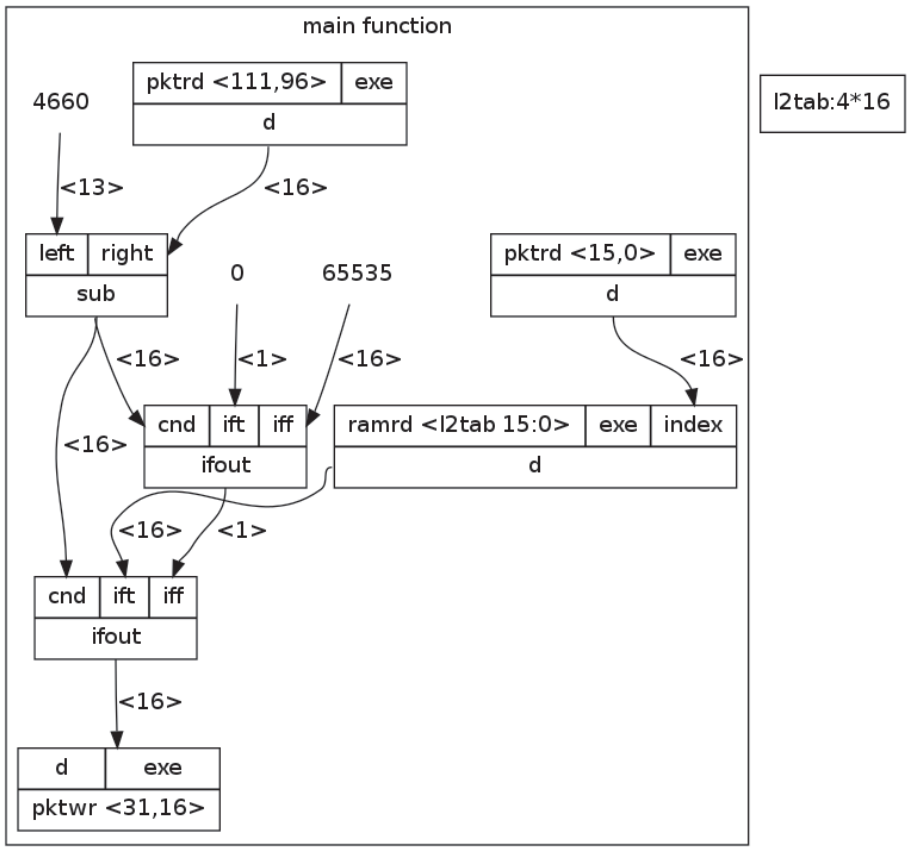


Figure 5.2: Example of a Graphviz DFG

PFP as well and get rid of packet FIFOs for the initial `pkt` interface.

Functionality of the PD is not as important as before once the packet FIFO has disappeared since the PFP will anyhow receive every single packet chunk. These chunks will have all the information that has to be decoded. Alternatively, merging functionality of packet decoding into the PFP can be a beneficial option. However, the critical issue of this proposal is caused by the location of decoding data because required packet information may have a possibility to be located in an unpredictable

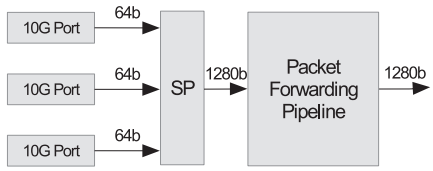


Figure 5.3: Move the SP to the head of the PFP

chunk number as the chunk size varies. To address this problem, Ethernet protocol must be taken into account. In a Layer 2 switch, packet decoding is actually the Ethernet packet header decoding. The Ethernet header features destination and source MAC addresses which have 6 octets each, the EtherType protocol identifier field and optional IEEE 802.1Q tag. As we know, in the initial design the processing unit in the packet processing subsystem is one packet chunk, and the SP between the packet processing subsystem and packet buffering subsystem collects small packet chunks to large packet cells. All in all, the size of the Ethernet header should be less than one packet cell (in our design it is 160 bytes). Suppose we move the SP forward and extend the input bandwidth of the PFP to one cell as shown in Figure 5.3, it is guaranteed that the Ethernet packet header is in the first cell of a packet. That is to say, in the new PFP module making packet forwarding decision is only triggered when the first cell of a packet comes, otherwise, it passes the packet cell through pipeline stages and output to the packet buffering subsystem.

The new PFP structure is shown in Figure 5.4, the entire PD is replaced by a direct extraction block in the PFP and the `first` flag is used as enable for the forwarding

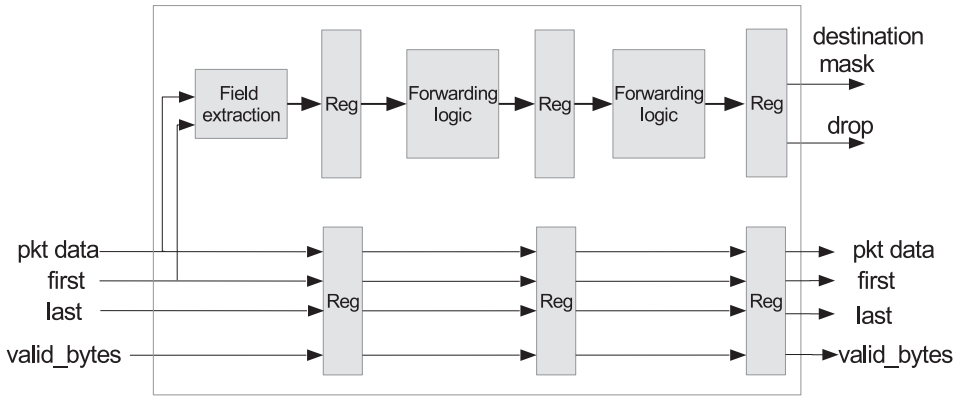


Figure 5.4: Simplified Packet Forwarding Pipeline Structure

logic.

Hardware infrastructure around PFP is hence incredibly decreased and turns out to be a structure in Figure 5.5, it is quite straightforward and has been compacted as much as possible. Obviously this architecture sacrifices flexibility to a certain extent, it cannot handle packet headers larger than a cell and the protocol cannot be too complicated. However, for a traditional Layer 2 or even Layer 3 switch, we consider it is sufficient.

Derived from software supervised DFG, the clumsy wire connection and buffer insertion in the PFP RTL module is disappeared. Previous `pkt` interface can be merged to PFP module effectively. As a consequence, the generated switch from this tool chain becomes more compact and relieve the routing complexity.

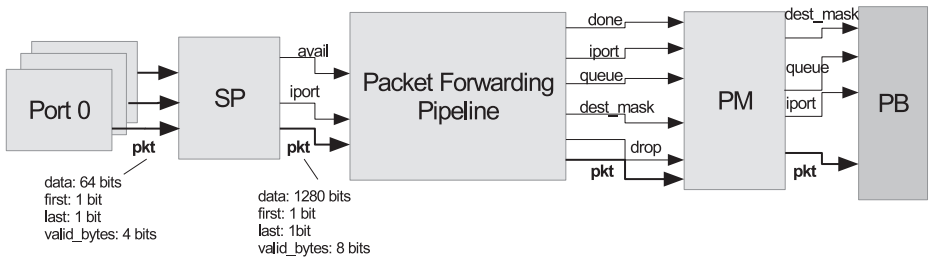


Figure 5.5: Simplified Packet Processing Subsystem

Table 5.1: Timing Comparison

		2 port	6 port
Initial design	Maximum frequency	129.655MHz	106.205MHz
Optimized design	Maximum frequency	134.803MHz	110.803MHz

5.3 Optimized Implementation Result

This section compares the implementation results between the optimized design and the initial design proposal. With the purpose of judging improvements we gained from the optimized design, below we list the synthesis results for the optimized one as well as the results we get for the initial design in Chapter 4. The comparison includes timing, device utilization and power consumption.

Table 5.1 lists maximum frequency in two designs and they all meet the required 105 MHz core clock frequency to get full cell rate. Apparently when it comes to 6 port the timing slack is limited, if we are targeting more ports design, the critical path which resides in the packet buffering system should be optimized.

Table 5.2: Device Utilization Comparison

		2 port	6 port
Initial design	Number of slice registers	4.87%	11.61%
	Number of slice LUTs	19.30%	50.09%
Optimized design	Number of slice registers	2.42%	5.19%
	Number of slice LUTs	5.07%	12.99%

Table 5.3: Power Comparison

		2 port	6 port
Initial design	On-chip power	1.955 W	4.030 W
Optimized design	On-chip power	1.585 W	3.427 W

Device utilization is another result that we focus on. As in Table 5.2, the area saved from optimized design is remarkable, with a 6 port implementation, resource occupation in the optimized design is still less then a 2 port design with initial structure. We should admit that the initial design is more flexible and leads to more redundancy, but since we are developing a tool chain from software to hardware, we have the capability to push flexible part under software control as much as possible while keeping hardware structure clean. On-chip power consumption shown in Table 5.3 is also an evaluation of the optimized design. As device utilization decreases, the power consumption drops accordingly.

The gains we get from the optimized design are due to using DFG as the intermediate format. It offers an opportunity to simplify hardware infrastructures a lot. Firstly, low pipeline depth in the PFP allows us to merge packet transmission into the PFP and hence the packet FIFO per port is replaced by registers in the PFP. Since PFP is shared by all ports, the hardware resources in this part will not change if number of port increases. Secondly, the unified packet transmission path in the packet processing subsystem provides possibility to get rid of the complicated PDs

in the initial design. In the original PD, extracted packet fields need to be stored in memories in advance, the packet read instruction is in fact a RAM read operation. In the optimized design, all the field storage logics are disappeared but a direct bit extraction operation is left. Without the complex interface between packet decoder and field memory, routing difficulty is reduced and device utilization is dramatically dropped. Above all, the valuable feature we can get from the two optimizations is that the hardware occupation of these logics are not dependent on port configurations.

Chapter 6

Conclusions and Future work

This chapter summarizes the thesis project, analyses worth of the project and lists further steps to make it generating real functioning switches.

With the aid of the compiler developed by PacketArc AB, we bridge the gap between software and hardware, developed a tool chain to automatically create hardware for Ethernet switches from a high level C-like language. This addresses the challenges to find an efficient solution to meet different port and bandwidth requirements. With the flexibility of high level packet processing language, the forwarding architecture can be generated by various forwarding algorithms very quickly. The output from the tool chain is synthesizable RTL code and based on different implement scenarios, a corresponding wrapper is provided to fit the target board.

So far at the end of the thesis project, we implemented a 6-port 10G Layer 2 switch with minimum forwarding functionalities. We have simulated the switch through random test cases successfully and finally come to an on-board test. In the throughput test 100% throughput is reached for fixed size packets and more stress tests for mixed

size packets are in progress. At this point, we may say the architecture and the entire tool chain has been proved feasible. Currently the tool chain is aiming at generating a pure Layer 2 switch, since packet processing subsystem is assembled by individual blocks, if a library with enough hardware infrastructure is established, the hardware architecture of the target switch could be determined more aggressively and wisely.

In future work, more packet processing scenarios need to be supported, this includes tool chain update from the compiler to the RTL generator as well as new hardware infrastructures. To easily observe the switch status like number of dropped packet, internal overflow or underflow, a set of internal observation points is required. Also, a user-friendly configuration interface is necessary so that the lookup tables in the switch can be configured conveniently. Eventually, a PCIe interface shall be hooked to the generated switch for user monitoring and switch inner control. Finally, the robustness of the switch should be guaranteed. if catastrophic error occurs in the switch, some assertion checkers are required to be able to reset the switch automatically.

Finally, based on this thesis project and the results that have been analysed, we have reason to believe that it is possible to use the developed tool chain to create efficient packet processing solutions for a broad range of requirements with less effort.

Bibliography

[1] ITU-T, "Recommendation x.200," accessed 20-April-2014. [Online]. Available: http://www.itu.int/rec/T-REC-X.200-199407-I/en

[2] B. A. Forouzan, *Data Communications and Networking*, 4th ed. McGraw-Hill Higher Education, 2006.

[3] U. V. Burg and M. Kenny, "Sponsors, communities, and standards: Ethernet vs. token ring in the local area networking business," *Industry and Innovation*, vol. 10, no. 4, pp. 351–375, 2003.

[4] J. Yoshida, "Ethernet backbone in car: Hype or reality?" June 2013, accessed 20-April-2014. [Online]. Available: http://www.eetimes.com/document.asp?doc_id=1319157

[5] D. Comer, *Network Systems Design using Network Processors*, 1st ed. Prentice Hall, 2004.

[6] A. K. Kloth, *Advaned Router Architectures*, 1st ed. Taylor and Francis Group, LLC, 2006.

[7] T. Sridhar, "Layer 2 and Layer 3 Switch Evolution," *The Internet Protocol Journal*, vol. 1, no. 2, 1998.

[8] L. Kleinrock, "Analysis of a time-shared processor," *Naval Research Logistics Quarterly*, vol. 11, no. 1, 1964.

[9] J. Aycock, "Compiling little language in python," accessed 20-April-2014. [Online]. Available: http://pages.cpsc.ucalgary.ca/~aycock/spark/paper.pdf

[10] J. Decaluwe, "Myhdl manual release 0.8," accessed 20-April-2014. [Online]. Available: http://docs.myhdl.org/en/latest/manual/index.html

[11] Xilinx, "Logicore ip 10-gigabit ethernet mac v11.4 product guide," 2012, accessed 20-April-2014. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ten_gig_eth_mac/v11_4/pg072-ten-gig-eth-mac.pdf

[12] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, "Graphviz and dynagraph static and dynamic graph drawing tools," in *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2014-417

http://www.eit.lth.se