Faculty of Electrical Engineering,
Mathematics and Computer Science

Master Thesis

Ethernet implementation in Clash

Rik (H.W.) Strijker

January 2021

Supervisors:
dr.ir. A.B.J. Kokkeler
ir. H.H. Folmer
dr. ir. P.T. de Boer
dr. ir. J. Kuper

Computer Architecture for Embedded Systems
Faculty of Electrical Engineering,
Mathematics and Computer Science

University of Twente
P.O. Box 217
7500 AE Enschede
The Netherlands

Abstract

Connecting to the Internet is becoming more common by the day, sometimes users do not even know a device is actively connected to the Internet. The way in which devices connect and communicate on the Internet is written in so called Request for Comments (RFC)'s. The RFC's are constantly updated to fix security issues or to keep up with the scale of devices connected to the Internet by introducing new Ethernet standards to be adopted by the Institute of Electrical and Electronics Engineers (IEEE).
The growing scale of devices connecting to the Internet and communicating via an Ethernet protocol makes the use of Application-specific integrated circuit (ASIC)'s and Field Programmable Gate Array (FPGA)'s more common. In general ASIC's and FPGA's are used to perform tasks where large amounts of data are processed using the same steps. There are many Ethernet standards implemented on an FPGA to route messages to the correct receiver or even to handle (parts of) the communication and send a complete reply. However, large parts of the communication can be handled by the FPGA, there is no complete Ethernet stack supporting multiple protocols which is still flexible for future updates of RFC's.
This project explores the possibilities of making a flexible Ethernet stack by using Clash. Designing a complete Ethernet stack is too big of a task to complete within one project, however the stack should be easy to extend and update in the future. Security plays a crucial role during development, it should be possible to detect what protocol is sent and filter all information according to the latest RFC's. The implementation within this project focuses on both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6), although in the implementation other Ethernet types can be add as well, for example EtherCAT. The Ethernet stack is made and tested in Clash up and until Dynamic Host Configuration Protocol (DHCP).
The implementation of this project can be used to send and receive messages in both IPv4 and IPv6 using User Datagram Protocol (UDP). One important part of IPv6 are the extension headers that can be add to a message. The implementation can find extension headers in a received message even when the extension is not implemented in the system. To keep the implantation small, custom bit encoding is used for the Ethernet type, the IPv6 extensions and the DHCP options. For extensions and options with a fixed length this length is used in the data constructor of the corresponding option or extension.

Acknowledgments

Now I have finished my MSc-thesis, first of all I would like to thank my supervisors ir. Hendrik Folmer from the University of Twente and dr. ir. Jan Kuper from QBayLogic B.V. for their patience and for their feedback on my work and guiding me during this project. Thank you for the time you have invested in me in this process.
I would also like to take this opportunity to thank my wife Angela and my parents for reading my thesis over and over, to find all spelling mistakes, to suggest accurate wording and to find all correct words used in the wrong phrases and vice-versa. As a person with dyslexia I could not do without such support. Without their feedback this thesis would be much harder to read and to understand. I would also like to thank the graduation committee members dr. ir. André Kokkeler and dr. ir. Pieter-Tjerk de Boer for their valuable and friendly feedback on this thesis. And I would also like to thank the team working at QBayLogic B.V. for providing valuable advice during this project. I have had a great time working at QBayLogic B.V. and learned there how to use Clash and I learned about some of the problems you have to face within compiling technologies.
Finally I would like to thank Albert Steendam, my remedial teacher a long time ago. He has not assisted directly with this thesis, but without his efforts I never would have successfully finished my school career and my Master study. Without his help I would not have been able to read or write in English or even Dutch.

Contents

Abstract iii

Acknowledgments v

List of acronyms ix

1 Introduction 1
1.1 Problem statement . . . 1
1.2 Research questions . . . 2

2 Background 3
2.1 Ethernet network . . . 3
2.1.1 Getting an address . . . 3
2.1.2 Find the website . . . 4
2.1.3 TCP request . . . 4
2.1.4 Master Slave Network . . . 5
2.1.5 Ethernet hierarchy . . . 6
2.2 OSI-model . . . 7
2.2.1 Industrial Ethernet . . . 7
2.2.2 Properties of layers . . . 9
2.3 Aspects of Ethernet . . . 10
2.3.1 Delays . . . 10
2.3.2 Security . . . 10
2.3.3 RFC . . . 11
2.4 Ethernet frames . . . 11
2.5 Clash . . . 12
2.5.1 Types . . . 13
2.5.2 Parsing . . . 16

3 Related work 17
3.1 Network communication . . . 17
3.2 Hardware-based solutions . . . 18
3.3 P4 language . . . 20

4 Design **23**
4.1 Introduction . . . 23
4.2 Ethernet connection . . . 23
4.2.1 Implementation of the receiver . . . 23
4.2.2 Data Link layer . . . 24
4.2.3 Network layer . . . 32
4.2.4 Transport layer . . . 46
4.2.5 Session layer . . . 51
4.2.6 Checksum calculation . . . 59
4.2.7 Memory Management Unit . . . 60
4.3 Implementation of the transmitter . . . 63

5 Testing **65**
5.1 Introduction . . . 65
5.2 IPv6 extensions . . . 65
5.2.1 Chained extensions . . . 67
5.3 DHCP . . . 68
5.4 Sending VLAN . . . 69
5.5 Summary . . . 70

6 Conclusion and future work **71**
6.1 Sub-questions . . . 71
6.2 Main question . . . 73
6.3 Future work . . . 73

A Code **79**
A.1 Ethernet type . . . 79
A.2 Annotation IpHeader . . . 82
A.3 Ipv6Extention . . . 83
A.4 TransportType . . . 83
A.5 DHCPBody . . . 84
A.6 DHCPOptions . . . 85

B Wireshark **87**
B.1 HopByHop extension . . . 88
B.2 HopByHop and Fragment extension . . . 89
B.3 DHCP caption . . . 91

List of acronyms

ASIC Application-specific integrated circuit

ARP Address Resolution Protocol

CPU Central Processing Unit

DHCP Dynamic Host Configuration Protocol

DNS Domain Name System

ECN Explicit Congestion Notification

ESP Encapsulating Security Payload

FPGA Field Programmable Gate Array

HDL Hardware Description Language

HLS High Level Synthesis

HOPOPT Hop-by-Hop Options

LSB Least Significant Bit

ICMP Internet Control Message Protocol

IEEE Institute of Electrical and Electronics Engineers

IANA Internet Assigned Numbers Authority

I/O inputs and outputs

IP Internet Protocol

IPv4 Internet Protocol version 4

IPv6 Internet Protocol version 6

IRT Isochronous Real-Time

MAC Media Access Control

MMU Memory Management Unit

MPU Microprocessing Unit

MSB Most Significant Bit

PHY physical layer

RFC Request for Comments

OSI Open Systems Interconnection

TCP Transmission Control Protocol

UDP User Datagram Protocol

VHDL Very High Speed Integrated Circuit Hardware Description Language

VLAN Virtual LAN

Chapter 1

Introduction

1.1 Problem statement

In the modern world connecting all sorts of electronic devices to a network has become more important, not only computers and telephones but freezers, lamps and sensors in the soil of a plant are connected to the Internet. This is not only the case in a home or office environment but in industries as well. Examples in industries are where valves are used for controlling cylinders that can report when the cylinders need maintenance. There are multiple new industrial Ethernet protocols used in machines to read or write the state of or to a motor and other inputs and outputs (I/O) [1]. Industrial Ethernet is a special kind of Ethernet used in machines but which uses the same principles and cables as regular home or office Ethernet. The above implies more complexity of development and needs flexibility for later security updates. With the large growth of devices connected to the Internet and more companies making products which can connect to these networks, it is important to stay ahead of competitors. This means the time between development and shipment to the costumer has to be even shorter, this so called 'time to market' makes it more important to have a well tested and flexible network stack within a project.
There are multiple chips that can connect to a network. An example is a Central Processing Unit (CPU) in your PC, or a Microprocessing Unit (MPU) within embedded systems. In both systems the largest parts of the Ethernet stack is handled by software. It is easy to update parts of the software via the operating system running on the processor. This can result in an easy to configure and well updated Ethernet application. There is one practical problem when using a processor, on one single processor there can be interrupts, user applications and an Ethernet stack running all at the same time. These processors are not suitable for multiple handling tasks simultaneously. The network process will now be less predictable and could be interrupted at any point in the process.
For most applications interrupts are not a problem, however it can be for industrial networks and applications. Because lots of data is processed and predictability is not guarantied. To avoid these problems the CPU can be replaced by a so called FPGA. On an FPGA hardware can be configured for a certain task, for example to process Ethernet frames. Data within an FPGA is often processed in parallel, especially within systems where there is time critical data. One example of industrial Ethernet often implemented on an FPGA is EtherCAT [2].
To configure an FPGA a Hardware Description Language (HDL) is used, either directly written for an

FPGA or generated from another language. A benefit of HDL is full control over the hardware placement, but it results in more complex code. For large projects the complexity leads to difficulty for maintaining or expanding the code. The alternative is to use a High Level Synthesis (HLS) language to generate HDL. Using HLS increases readability by adding a level of abstraction from the hardware. Generally a HLS, like C, is not only intended for hardware design, which sometimes leads to complex hardware realizations. To design a multi protocol network stack running on an FPGA which is generic and easy to expand, HDL is not suitable. The HLS must give control over the bytes on the FPGA to provide a flexible and well tested communication stack.
Clash, on the other hand, offers the possibility for embedded languages as data types, a feature that seems useful to express protocols. Clash' type system provides the tools to create an embedded language easy to test and expand [3]. Clash will give full control over the generated hardware to prevent complex hardware realizations. The ultimate goal is to investigate the possibilities for data exchange which can support multiple Ethernet based protocols. Processing an Ethernet frame on an FPGA has been done before, however this thesis investigates a generic, expandable and testable multi protocol solution.
The desire for a full and extendable multi protocol solution is recognized widely, but to the author's knowledge only partial solutions exist. Hence, it is worthwhile to investigate whether Clash offers a suitable approach for a possible solution. The research of this thesis is carry out within QBayLogic B.V., the developers of the Clash compiler.

1.2 Research questions

In the forgoing it is stated that using Clash to read an Ethernet frame could be beneficial. This leads to the central question of this research:

What (dis)advantages will the Clash type system offer when investigating the implementation of a flexible, expandable and generic multi protocol network interface on an FPGA?

In support of the main question the following sub-questions will be addressed.

- Is the type system of Clash suitable for building a network stack?
- Can IPv6 extension headers be handled reliable on an FPGA using Clash?
- In what way does the Clash type-checker help to handle Ethernet frames?
- How can similarities between protocols be handled in the Clash type system?

To answer the questions above a proof of concept of an Ethernet stack has to been implemented in Clash. This system should be designed in a way that new protocols can be added easily. Important elements in the design are the IPv6 extension headers. In the future there will be new extension headers or patches for security problems released. Chapter 4 contains the analysis of the Clash implementation, where the components needed to make an Ethernet stack are described.

Chapter 2

Background

There are a large number of Ethernet protocols used in different applications. In the world of embedded systems an increase of Ethernet use can be seen in all sorts of operational fields [4]. This leads to more Ethernet solutions, not only in CPU but in FPGA's as well. The increase holds for known protocols and for industrial applications. With the increasing connectivity and the (re)use of standard building blocks there is a greater need for an easy to configure and well tested Ethernet stack.
Every message sent over for instance an Ethernet wire should comply to the RFC standard. This is the basis of Ethernet communication and describes the layout for every message and protocol. One model used to describe the position and functionality of different protocols is the Open Systems Interconnection (OSI)-model. The OSI-model is used to describe how a computer and server communicate with each other. Ethernet communication is possible on an FPGA, to describe this in Clash knowledge about the type system is needed.
The first part of this chapter discusses the basic functions used when visiting a website. The second part will describe how the OSI-model works, and describes the difference between home and office Ethernet and Industrial Ethernet. The last part contains the features in Clash that can be helpful building blocks in designing a parser.

2.1 Ethernet network

To understand how a computer is connected to a network take the following example. Someone is using the computer to visit a website, for instance *example.com*, and types in *http://example.com/* in the address bar. The computer has to find out where the server is located (on what IP-address), and should make a connection in order to get data back to be shown on the screen. Before doing so the device must have an IP-address.

2.1.1 Getting an address

When a device connects to a network, wired or wireless, it will try to get an address, for example an IPv4-address to identify the device within the network. When there is in this same network a DHCP-server(s) present, used to assign the IPv4 address, the device will use the DHCP-server. The device and server will send messages to each other to agree on what IP-address the device will live on. This

IP-address is a numeric address. The DHCP-servers can be seen as a municipality where one has to register when moving into a new house without an address. The municipality will take the registration and assign a unique address. In order to do so they need one's personal code to know who will live in the house. For devices the same methodology is used, the personal code for devices is called the Media Access Control (MAC)-address. This is a global unique number stored in the network adapter of the computer. When all registration is done and the addresses are known the real browsing will start.
To get the IPv4-address from the DHCP-server the device sends a message with his MAC-address to the DHCP-server. The DHCP-server will assign an IPv4-address to the device.

2.1.2 Find the website

Lets go back to the web browser, typing in *http://example.com/* will display a website. The address is plain text, however the device needs a numeric address to find the server before it can send your message. Figure 2.1 shows a flowchart of this process. The computer is connected to the Internet through a router. The device will first send a message to the Domain Name System (DNS)-server, via the router. The router contains a list of connected devices or servers, it will route messages to the destination. This is comparable to a logistics centre where packages are collected and brought to the correct truck via some complex system of conveyor belts. The truck will deliver the message, sometimes via different logistics centres.
Finally the textual address can be converted into a numeric one, this is done in the DNS-server, "the phone book" of the Internet. This number will travel back to your device, over a route via some routers and will arrive some time later back at the left side of the figure. Now all information needed by the device is present and can be used to load *http://example.com/*.
This process can be more complex; sometimes the DNS-server does not know where to find the webpage one is looking for. However, it will give you the address of a server which has more information. On the other hand most devices have their own list of addresses for servers that are most often used, like the list of contacts in a mobile phone. The numbers that are most often used are saved with some textual representation such as the name of a person.

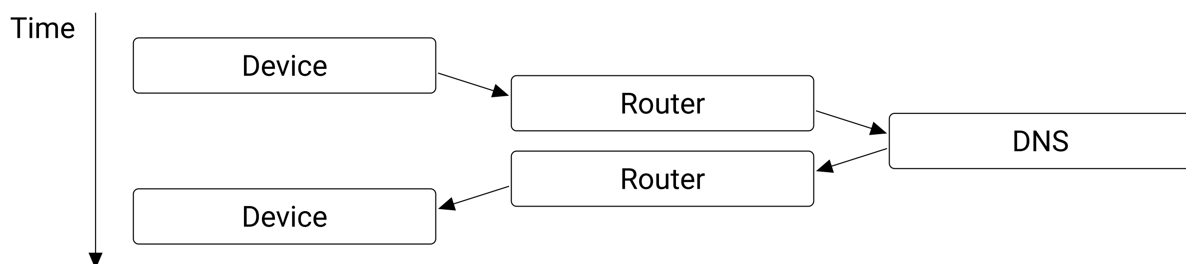

Figure 2.1: Sending a DNS request, with the steps and time.

2.1.3 TCP request

In order to simplify the description of the process outlined in the following example the routers will not be mentioned any more. A website is loaded via a so-called Transmission Control Protocol (TCP)

flowchart. Figure 2.2 shows what happens over time (without showing the details). First the device will send a request, a so called synchronize (SYN) package. Resulting in an acknowledgment (SYN-ACK) from the server. Sometime later the connection will be acknowledged by the device, the requested information is sent back to the device and can be shown to you, an acknowledgment is sent back to the server. Most servers will send data in bits and pieces which have to be reassembled in the device to form one webpage.
The description above is a basic example of a server-client system, all connected via routers. This type of network is request driven, servers will only send data on a clients request. There are innumerable servers and clients all over the world. Not all servers should reachable for all devices, to protect networks a firewall is used. Access to the network is organized with black and whitelisting in the firewall, all devices on the blacklist will be blocked, only the ones on a whitelist can get access. Network operators will manage this firewall.

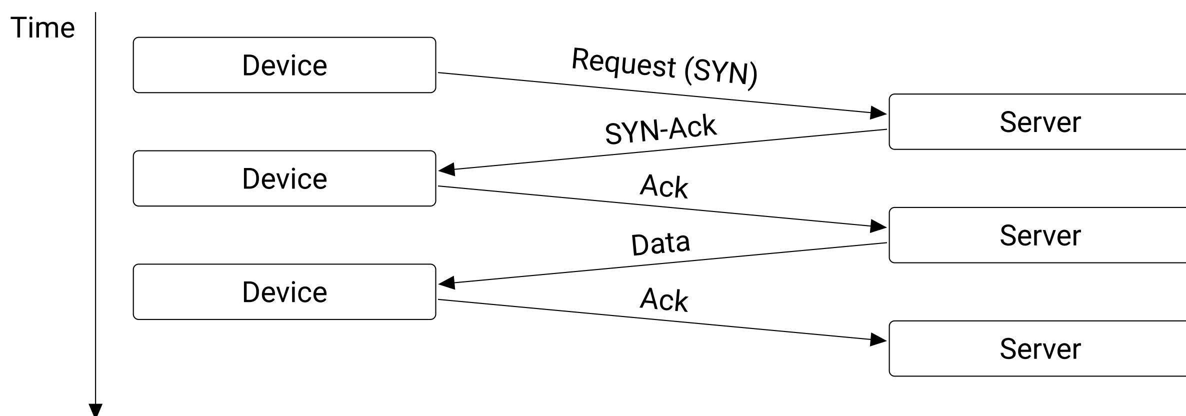

Figure 2.2: Sending a TCP request, with the steps and time.

2.1.4 Master Slave Network

The server-client networks are not the only network topologies, there are also Master-Slave systems. In this type of networks there is one Master connected to one or more Slaves. This system is primarily used in industrial system control. In contrast to a server-client network the Master-Slave topology has a cyclic behavior. When the communication starts, the Master first discovers the number of Slaves, the physical location in the network and how much data the Slave needs and can send back.
Take for example a simple system containing one Master and two Slaves like in Figure 2.3, the first Slave (in the middle) has 3 inputs and 5 outputs, the second has 4 inputs and 2 outputs. The Master will discover the amount of data needed by the Slaves before the communication starts, and will construct the message in a way both Slaves have exactly the space they need.
Using the example where there are 7 inputs (3 and 4 for Slave 1 and 2 respectively) and 7 outputs (5 and 2 for Slave 1 and 2 respectively) you would expect the message to have a length of 14 bits because this is the total amount needed. However, the Slave will first read the output data and then fill in the input bits. This is possible because Slaves will always send the same amount of data and Slaves can not get to another position when sending data. This has the advantage for the length of a message

because the bits for the output will be reused for inputs. The amount of data is the maximum size of the inputs and outputs of one Slave, the largest number will determine the bits needed for one Slave. This will result in 5 and 4 bits for Slave 1 and 2 respectively because Slave 1 needs at least 5 bits to represent all outputs, Slave 2 needs 4 bits to represent all inputs. Hence the total length is 9 bits instead of 14. The flowchart in Figure 2.4 shows that the Master will send a message to Slave 1, it will take out the outputs and fill in the input fields of the message and will forward it to Slave 2. Slave 2 will drain the outputs from the message and fill in the input fields and will send it back to the Master via Slave 1. Slave 1 will not update or add any fields, and therefore has been left out in the figure. Back at the Master all information will be processed and a new state will be calculated to repeat the process all over again. This does fundamentally differ from the server-client model where you need to send a request to every server you want information from. *http://example.com/* will never send data to a client that did not ask for data. A client will only ask for data when needed, in the Master-Slave network this decision to ask for updates is made by the Master, even if there is no update there will still be messages send.

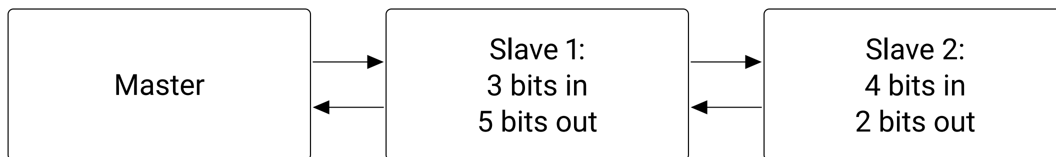

Figure 2.3: Message flow in a Master Slave system, there is always one Master and there can be multiple Slaves.

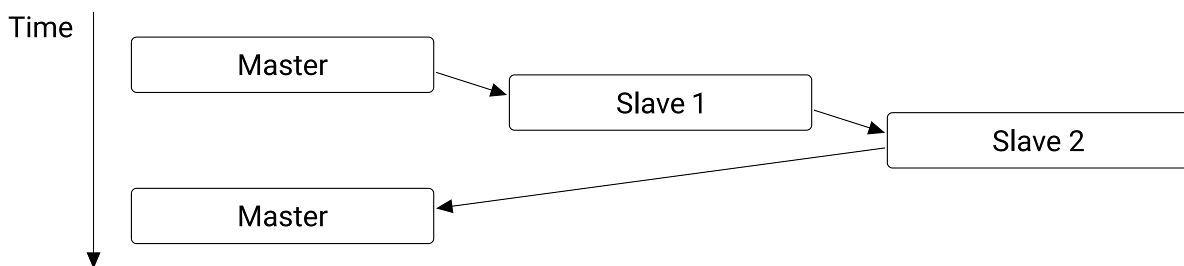

Figure 2.4: Flowchart of a message send over a network containing one Master and two Slaves, nodes are only shown when they process the data of a message.

2.1.5 Ethernet hierarchy

There are many different ways to send data over a network-cable or wireless, using all kinds of setups. This research will only focus on Ethernet frames. However, the data is not necessarily processed in one stack. A leading part of this research is how to combine the different protocols used on the Ethernet wire in one stack. Every device connecting to a network will use (parts of) the OSI-model. This includes clients, servers and routers. This OSI-model describes in layers how a device should interpret a message in seven abstraction levels.

2.2 OSI-model

The OSI-model has seven layers, containing: **Physical**, **Data Link**, **Network**, **Transport**, **Session**, **Presentation**, and **Application** [5]. In Figure 2.5 the OSI-model is depicted on the left, on the right there is an example depicted, both are explained below. The OSI-model is divided into three parts, the upper three layers are application dependent. These are often taken care of by specific applications. The lowest two layers are used by all protocols, data will always travel from one point to another, so a **Physical** layer is always present. Within the **Data link** layer the basic functionality is always being used, sometimes this layer is extended to get higher performances. For example in EtherCAT the **Data link** layer is used to make a connection between the **Physical** and **Application** layer, the **Data link** will handle the content from an incoming message and will then construct the outgoing message by filling it with application data.
The OSI-model is best described by the following example. Take the **Application** layer, where there is a letter send to someone. A message is written and the name of the receiver is included. An assistant (the **Presentation** layer) will encrypt the letter to prevent others from reading it and then gives it to the mail department. All letters are sent in a box to be sure that the paper will not be damaged during transport and delivery (**Session** layer). When sending a box it should be clear that the receiver is at home, because there are no neighbors to take care of it. The **Session** layer will check this before sending the box.
All information needed by the mail department is sent via the mail company, they have a delivery point where you can bring the package and where it is weighted to select the best service (**Transport** layer). There are two scenarios, either they can send the message really fast, but parts of the letter may be lost, or they send it more slowly in a more reliable manner, where after every page the receiver has to send an acknowledgment back to be sure every page is received.
The message will go to the logistics center to get it in the correct truck for delivery (the **Network** layer), there they will replace the name with a barcode (Ip address). The truck selected by the logistics center can be too small for your message, in that case it is cut in pieces to be reassembled at the receiver.
Next the mail man will put the packages in his truck (**Data link** layer) and adds the complete address of the sender and receiver (mac-address). The truck will bring the message to the receiver, there all steps are repeated but in reverse order.
The above works really well for loading a webpage or sending an e-mail. For streaming applications the checks are less important, the sender will not check if you are still home, it will just keep sending till it gets a letter back saying you moved away.
As discussed earlier, not all protocols will make use of all layers. Within some solutions where a low latency is a requirement, like industrial protocols, it is common to omit layers [6].

2.2.1 Industrial Ethernet

When using the complete OSI-model (for example when browsing to a website) many checks are performed to make sure all requested data has been sent and received correctly. These checks and retrying when data packages get lost, takes a lot of time and processing power. Within fields where

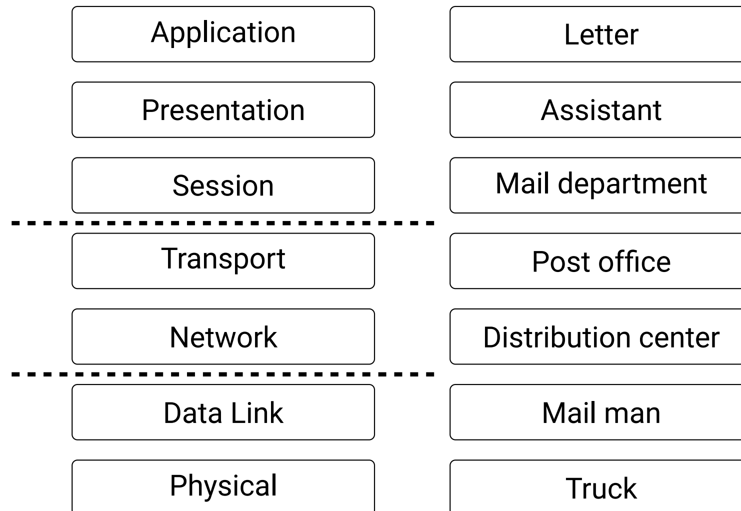

Figure 2.5: The seven layers of the OSI-model.

the amount of data is large and time is critical (for example in streaming audio or video) a delay can make it difficult to have a conversation, whereas missing a frame is not a critical problem. When in this case the majority of frames arrive on time you will still understand the conversation. Sometimes it is important to have no delay at all (for example in industry), missing a frame can be catastrophic when in this frame an Emergence-stop is set. To make the communication more robust the Master will send an update every millisecond, where most of the content of this message will be the same. Because of the redundancy within sending messages fewer checks are needed to still guarantee the arrival of data.

Less checks and more speed is the reason to choose for the use of less OSI layers. Let us go back to the example of sending a letter. Suppose you would like to know that it is delivered properly. But you need an ambulance and have to order it via a letter. In that case it is less important how data is represented as long as there is a guarantee that it will be delivered because in fact you only want to send an SOS. In this case, it does not matter if anyone between you and the receiver can read what the content of the message is. You will run to the mail man and give him the message, if needed it will be cut in pieces by the mail man. Because of the hurry he will even write the address on the envelop for you and put it in the truck. The downside to this approach is you will never know if the message has been received. But you can not wait for the receiver to answer your letter because the ambulance is really needed. You send the same letter over and over again until the ambulance has arrived. After all this stress you are in need for a coffee. You send a letter in the same way, asking for coffee and keep resending it till the coffee arrives. Again there is no conformation required whether the letter has been received, because at some point you will get a response in the form of a cup of coffee. This is what happens in high speed industrial Ethernet protocols. All time-consuming parts of sending information has been cut away. Messages will be short, mostly set or reset, for example a value like temperature. Messages will always travel a short physical distance, most of the time in a machine or factory, i.e. Within some (kilo)meters, in any case not to a server at the other side of the world.

2.2.2 Properties of layers

In the **Physical** layer there are only variations in speed, a message could be sent over a fibre optic line (for example a sports car instead of the truck) or an old telephone line (for example walking instead of using the truck), but it is still the same data, only the representation differs. In the **Data link** layer a few variations are possible, most packages will be in an Ethernet frame (the mail man could be a woman as well), only the package inside a frame differs. However, the **Network** layer offers many different implementations, there are all sorts of systems using an Ethernet frame for sending data. However, in the OSI-model only IPv4 and IPv6 are used (there are many different logistics centers, they are specialized in different types of goods). In the **Transport** layer there can be multiple ways to transmit data, e.g. UDP and TCP are the well known examples of protocols on the **Transport** layer. Above the Transport layer, shown in Figure 2.5, processing will be more complex and is done by applications to interpret data. The **Physical** and **Data Link** layer do not variate a lot in terms of implementation, the same basic hardware components are needed in an FPGA. In the implementation of the **Network** and **Transport** layer large variations are possible. This is why in Figure 2.5 these layers are between horizontal bars. However, some protocols will extend the **Data link** layer, for example in industrial Ethernet like EtherCAT. This is shown in Figure 2.6 where all the three layers used by EtherCAT are depicted. The **Physical** layer is exactly the same as a standard OSI implementation, but the **Data link** layer has a loop back function to create a ring bus structure and it has a synchronization mechanism [7]. The next layer used in EtherCAT is the **Application** layer, this layer provides the user with data or will switch the physical I/O. This is only one example of different implementations of the OSI-model where layers are omitted. The same type of modifications are done in the PROFINET Isochronous Real-Time (IRT) protocol. In most network implementations on FPGA's only the bottom four layers of the OSI-model are being used [8], [9]. From the **Session** layer upwards making decisions on what to do with data will be a user or application task, in most systems this will be done by a processor. Although there are lots of services using different protocols within Ethernet, they can all coexist on the same medium and most of them can be processed on the same hardware. Modern computers will have no problem with these kinds of multi protocol networks, with the correct software they can even be used as an industrial Ethernet Master.

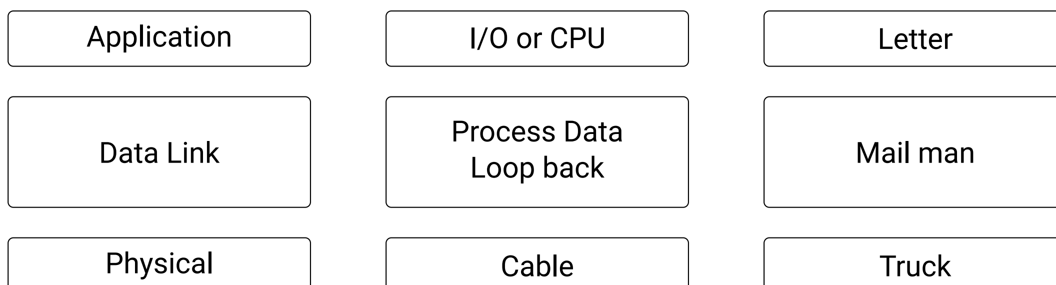

Figure 2.6: The OSI-model used in EtherCAT communication.

2.3 Aspects of Ethernet

The OSI-model offers a way to describe network communication is seven layers of abstraction. The model is used in all kinds of devices serving different needs. The model is generic, delays or security are not embedded in the OSI-model. However, for making an Ethernet stack security and delays are important to think about in advance. There is no description for a protocol in the OSI-model, all protocols are described in so called RFC's. The RFC's are important guidelines when developing an Ethernet stack. The Delays, security and RFCs mentioned above are described in more details below.

2.3.1 Delays

As said, above delays in Ethernet are not a real problem, computers are really good at general Ethernet communication. However, in specific applications a standard (Embedded) computer will introduce undesirable delays or has a large power-consumption [8]. This is one of the reasons for using an ASIC or FPGA in industrial applications. There are integrated micro controller cores for EtherCAT, the core will keep a connection with the Master. The EtherCAT core will provide real-time processing of industrial protocols, although there is no clear definition of Real-time Ethernet. There are calculations for the expected delays. So when a implementation is made in an FPGA it is important to keep delays in industrial Ethernet small.
There are different ways to generate hardware on an FPGA, depending on the manufacturer, Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Verilog are two widely used programming languages for FPGA's. They are used to design large systems at bit-level. It is less convenient to use them for data-flow problems. Data-flow modeling on an FPGA is easier when done in a language like Clash.

2.3.2 Security

One way to keep intruders out of a network is by using a firewall, which has to be configured so not all messages can go into the network. Recent research [10] revealed problems regarding correct configuration of firewalls in combination with IPv6. More details about IPv6 can be found in Section 4.2.3. This kind of misconfiguration can lead to hacking a network. Especially with more internet users, getting past a firewall is more beneficial for hackers.
When a problem is found in the guideline for implementing IPv6 a new RFC is published. RFC's are documents describing how a sender and receiver should behave within a network, in Section 2.3.3 this will be explained further. Writing a new RFC will take time and it has to be implemented in all running devices to make sure that the security problem has been solved, in practice this never happens because the scale of internet has become too large. When there is an implementation mistake in the FPGA or when a new RFC is published, an update of the FPGA is required. Updating an FPGA at run-time is not common for various reasons, so when there is a mistake in the Ethernet core this will be hard to fix. Therefore it is better to use a design language that can be easily verified and tested. A possible design language that has these characteristics is Clash. By using Clash instead of HDL most security and testing problems should be easier to tackle; using the strong type-checker of Clash will discard extension headers when faulty or unknown. To add a new extension header to the

Ethernet stack only the *type* should be extended, all other fields can be packed in the same *type*. These techniques can be a step to create a more robust Ethernet implementation using Clash compared to traditional FPGA languages.

2.3.3 RFC

The RFC's are important for implementing an Ethernet stack, Ethernet is an IEEE standard. Most of the recent RFC's are published by the Internet Engineering Task Force. The RFC's are written and reviewed carefully, however there is always a possibility of mistakes in (parts of) protocols [11]. Every message can be seen as a frame, built using the OSI-model, every layer has a number of possible protocols and there will be an RFC describing the function and layout. To send a frame multiple layers are used, so there is always more than one RFC involved. Not all protocols will have their own RFC, some are based on multiple RFC's [12].

2.4 Ethernet frames

For every website we visit, E-mail we send or when starting the computer, parts of the OSI-model are used. The **Physical** layer of the OSI-model (at the bottom of Figure 2.5), is the layer where a message is send over a medium, either wired or wireless. To get data from this connection an ASIC is needed. Before sending data the ASIC, known as the PHY, has to be configured. Building a physical layer (PHY) on an FPGA is not possible without using other electronics. When the configuration of the PHY is done, all data given to the PHY will be sent over the medium. The PHY and cables (or antennas) together form the *Physical* layer. In Section 2.2 the *Physical* layer is compared with the truck transporting a letter. From this wire the FPGA will receive an Ethernet frame, the structure is shown in Figure 2.7. The Ethernet frame is passed to the **Data Link** layer to be processed. The Ethernet frame starts with an Ethernet header followed by the payload, in Figure 2.7 an IPv4 or IPv6 is used as an example. The "*Payload (IPv4/6)*" can again be split in a header (IPv4/6 Header) and payload (Payload (UDP/TCP)). Interpreting the header is done in the **Network** layer, in Figure 2.7 this is the second line, starting with the IPv4/6 Header followed by a payload. Again, in the next layer a header is used to interpret the payload, the last line of Figure 2.7 shows this header and payload. This header is needed by the **Transport** layer to select the correct **Session** layer protocol send in the payload of the **Transport** layer. As can be seen, every layer needs information on how to interpret data send in the payload.
The first header of the Ethernet frame is needed to know where a message came from and where it should be send to. Every frame sent over a wire will start with a header, regardless what protocol is used inside a frame. The header will always be present in an Ethernet frame and will be used in the **Data Link** layer. This header is important, in this header the protocol used in the Payload is represented. The Payload is where the real data is stored.

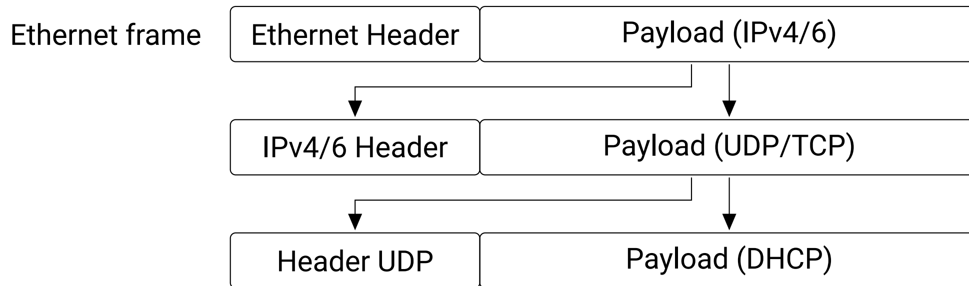

Figure 2.7: Stacking of protocols in an Ethernet frame.

2.5 Clash

FPGA's are configured (also called programmed) using a HDL. There are different FPGA's on the market, and different HDL's. Two standardized HDL's are VHDL and Verilog.
Processing an Ethernet frame on an FPGA has been done before, however never by using Clash. Using Clash could give a more flexible solution compared to a mainstream HDL, Clash has Powerful abstraction mechanisms, which may give more dynamic to design and test an implementation. When designing hardware on an FPGA it is efficient to reuse function blocks of hardware for the same calculation. This can only be achieved when the two parts of the system do not run at the same time, this is one way to get a small implementation on the FPGA. Similarities between layers can be used to achieve this goal, for example the *Header_Checksum* calculation of an IPv4 header and UDP frame are equal and will never be calculated at the same time.
It is common to design hardware of an FPGA using a HDL for the synthesis of the selected FPGA. There are different FPGA vendors, most of them make use of VHDL or Verilog to synthesize hardware. In this design process Clash can be used to get from a (mathematical) Haskell specification to hardware; in Figure 2.8 this design flow is depicted. The scheme starts with a mathematical specification in Haskell, like an algorithm that can be tested in isolation and as a complete system. The Haskell program is easily translated to Clash. The hardware behavior can be tested in Clash during the design, from this program and the tests HDL is generated. The generated HDL can be both VHDL and Verilog, for both languages tools are available to test systems. The same tests written in Clash will run in the HDL as well and can run using the vendor specific tools. Then the HDL can be used to synthesize hardware, again using the tools provided by the FPGA vendor. Using this flow makes it easier to find mistakes in the algorithm because all parts can be tested for functional correctness in isolation before being used as a system, where in HDL the design is about the timing in hardware. Clash has the advantage of a strong *type* solver, where a function will only produce the specified *type*. In Ethernet this can be useful, for example in IPv4 and IPv6 overlapping fields can be stored in the same type, in hardware this will result in using fewer registers. For example *time_to_live* and *hop_limit* have the same practical use case.

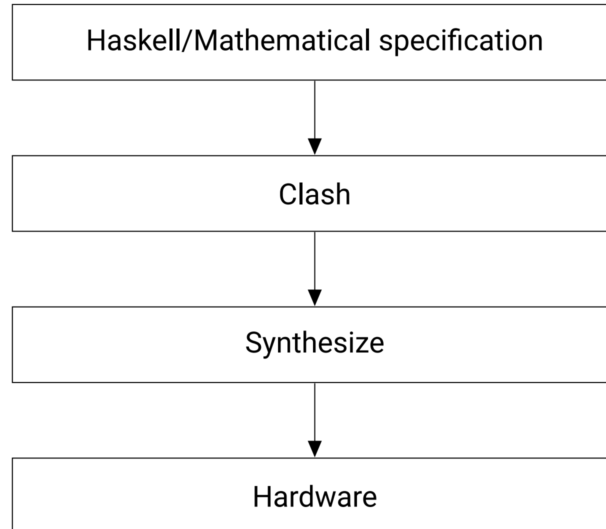

Figure 2.8: Design flow of hardware when using the Clash language.

2.5.1 Types

The frames described in Section 2.4 can have different EtherTypes, this specification will be used to only pass frames with a known protocol. This behaviour can be adopted in Clash. We can make a Clash *type* for Ethernet using the type value as an encoding for the data constructor, applying this the frame can be parsed without writing a parser. When a frame is unknown it will be discarded by the system using a wildcard. This behavior will help to avoid problems in wrongly parsing of IPv6 as described in Section 4.2.3.
The same structure will be used when handling options, the code used in the RFC to encode an option will be used in the data constructor as well. 'Packing' the Code with the Length and Data will result in a complete parser. In the case of an unknown option the RFC specification of dealing with this unknown option will be implemented as a fallback.
There are some important types in Clash to keep all types at a fixed length, Vec is one of them, it has a built-in length. Below are described the types often used in the realization of this Ethernet core. The type system of Clash allows the user to create his own types, when the new type follows a basic set of rules it can be translated to a HDL for programming the FPGA. Creating custom types makes the type system of Clash flexible, where the powerful strictness of the type checker will help to keep the design synthesizable. It is part of this research to investigate whether the Clash type system is useful for dealing with protocol handling

Maybe

One *type* used a lot in Haskell is the Maybe *type*, the Maybe type can have the value *Just A* or *Nothing*. In Clash it is common to use this when sending data in combination with a valid flag. For valid data *Just data* is used, when there is no valid data the sender will send *Nothing*. Using this data *type* makes creating a data bus much easier.

Undefined

One special value a type can have is undefined, in the simulation this will be represented by X, in hardware this means that there could be some voltage or non at all. When a system starts there will be a unknown state so all lines will be undefined, when a reset is applied the state will be known and will propagate through the system. There can be one more cases where an undefined value is used, namely when a *type* has more fields as there is data loaded into a system. Assume we know there are either 1 or 3 fields of valid information. This amount is determined by some number sent before the data: a kind of header telling the length. This will result in a structure like depicted in Figure 2.9, a length field followed by 1 or 3 bytes of data.
This *Input* data *type* in Clash will result in a *type* of 4 bytes and one bit for encoding A and B, the data constructor. In HDL there will be 33 bits allocated even if the incoming data has length 1. Assume the information stream into the system will come in portions of 4 bytes, not all information has to be valid, this depends on the first field. In this case the stream of data can be 'packed' in this *type*. Let us describe two cases for the information stream like the one in Figure 2.10. The first example is a message with length 3 and data a b and c. All fields have valid information because the length is 3. Packing this data all fields will be used and stored in *type* B. The information in the second case is the same, except the length is 1 so only field a is valid. Both examples have a box around the valid fields. When 'packing' the information coming from the first stream in the presented *type* it will fill all fields. The filled fields are shown at top of Figure 2.11 where the length is set to 3, the *type* is B and the data is stored in the data fields. For the second example there is only one field of data needed, only the length and data a are stored. The other data is not used. The first two fields will not contain valid data, so they are undefined. In the Clash simulation the values stored in these fields are not visible, in most simulators within HDL it will be an undefined value represented by a X.

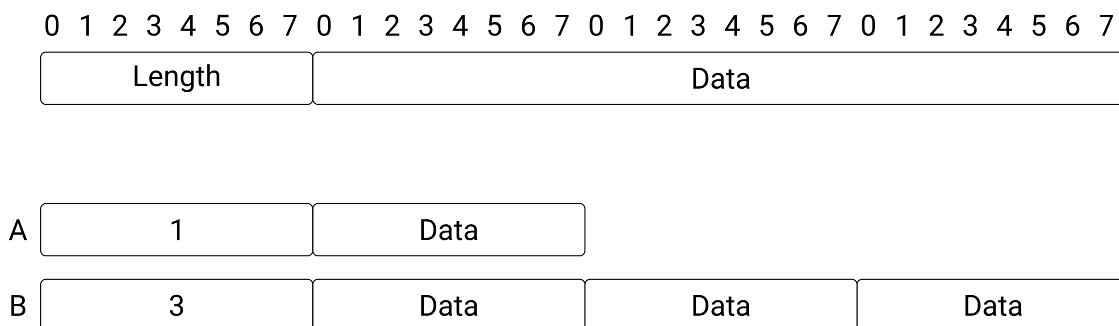

Figure 2.9: At the top, the input stream is depicted. Next is the schematic representation of the data type Input, built out of two entries, A with one length field and one byte of data and B with one length field and three bytes of data.

Custom encoding

Clash can 'pack' bits to other *data types*, this facilitates data transformation from one *data type* to another. Packing data can be used in combination with custom encoding, this is an important step in designing an embedded language. For the last example in Section 2.5.1 where there is an *Input data*

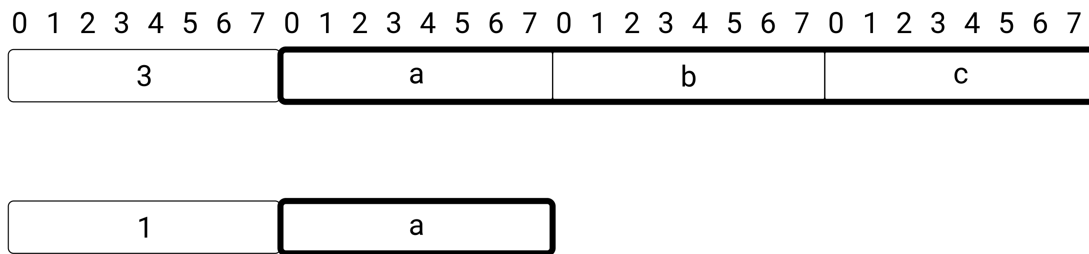

Figure 2.10: Two types of input data, one has a length of three, and one with a length of one.

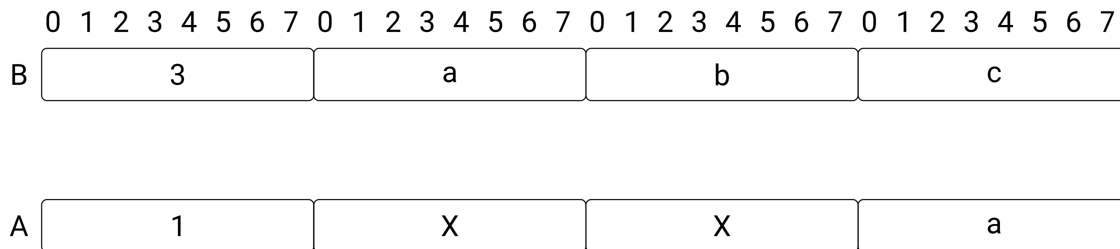

Figure 2.11: Undefined data stored in a variable.

type that can either have the length of 1 or 3 bytes, Clash will use 1 bit for encoding the *type* and 32 bits for the data, so a total of 33 bits. In this case the encoding will result in $0 = A$ and $1 = B$. However, in the example, *Input data type* has a length field, the value of this field is unique for all possible fields in this *data type*. There are two different lengths in this example, the length field can encode up to 255 fields in the *data type*, so custom encoding can be used. The custom encoding will result in a *type* of 32 bits where $1 = A$ and $3 = B$. In this example there is only one bit saved, however when there are 255 fields in a *type* there will be a saving of 8 bits. The second benefit is the direct translation from a bit patron to a Clash type, when the patron in the input has the value 3 constructor B will be used.

Wildcard

When using the custom encoding with less fields than the maximum number of possible encodings ($255 - 2 = 253$), all other codes can be redirected to a wildcard. For example the *Input data type* can be extended with field C having 3 fields of data; in Figure 2.12 this extended type is shown. The fields A and B are still represented by 1 and 2 respectively but C is represented by all other values. This can be used as some kind of fallback to detect errors in the data.

Records

The types described above can be used in functions within Clash. A function will always have a number of input types and one output type. To get data from one type to another a function is needed. One way to make a function for a specific type is to use a record. Listing 1 is a minimum example of a record. There is a data type called Ip (line 1) with a record called *version*. When the information stored in this Ip type is needed, the function *version* can be used, this will return the *BitVector 4*. The function *version* can also be used the other way around, when there is a type Ip and a *BitVector 4* the *version*

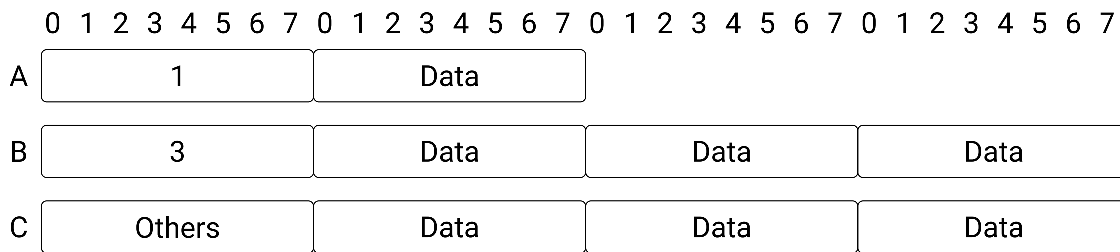

Figure 2.12: A schematic representation of the data type Input, built out of three entries, A with one length field and one byte of data, B with one length field and three bytes of data and at last the wildcard named Other with one length field and three bytes of data.

```
1   data Ip =Ip { version :: BitVector 4}
```

Listing 1: Example of a record called version in the datatype IP.

function can be used to update the Ip type.
The records are used a lot in the implementation of the Ethernet core as aliases for the fields of the different protocols on the different layers of the OSI-model. Updating one field of a type is easier when using this record syntax. In Listing 1 there is only one record field, in the implementation all fields of a protocol will have their own record.

2.5.2 Parsing

As discussed in section 2.3.3 RFC's are the building blocks describing the layout of Ethernet frames and protocols used on a network. However, an incoming message is a stream of bits of a certain length. When the stream of bits is interpreted using the RFC's a chain of protocols can be identified, one protocol is used for every layer of the OSI-model. The types in Clash can be made in the same way as the bit stream has to be interpreted. The translation between the bit stream and the Clash type is called parsing.
Writing a parser in any HDL can be complex. This is especially the case for large systems, as there will be a lot of cases and exceptions. When a new option is added, all paths leading to this path should be updated. When using Clash, parts of the parser can be encoded in the *data type*; adding an option or frame will only be an extra line in the data type to specify the length and data. The custom encoding requires an extra line to make the translation between the input stream an the Clash type. By adding this line in the encoding the extra option will be parsed automatically and stored using the specified *data type*. Of course functionality has to be added before the system can deal with a new option. So the parser in Clash will arise automatically, it has to be written only once. Adding the option only works with a fixed maximum length.
The above is a great advantage in IPv6, where currently extension headers are still under development, so they are likely to be adapted in the future. When a new extension header will be published, the Clash implementation can be extended and tested fast. As in HDL all bits from the bit stream have to be checked one by one.

Chapter 3

Related work

In this chapter other research concerning three aspects of network connectivity will be reviewed. Starting with the hardware architectures that are used for connecting to a network and the way different solutions are configured. The second part will discuss two FPGA-based solutions, one for home and office networks and the second for Industrial Ethernet. The last part of this chapter will discuss a HLS used for many Ethernet stacks, not only for FPGA's but in routers and switches as well.

3.1 Network communication

Devices all over the world are connected via Ethernet and all modern operating systems can communicate via Ethernet [13]. There are all sorts of devices connected to the Internet for the purpose of reading data from sensors or turning things connected to the network on or off. In some cases CPU's are replaced by FPGA's to get more computational power in the system [14].Replacing CPU's by FPGA's is common when the amount of data that needs to be processed is large.
Using FPGA's for processing and sending data over a network is possible, for example in a sensor network [15], [16] where the FPGA is used for processing and sending data. The problem with sending data over a network via an FPGA is the configuration of the network core in the FPGA, this configuration has to match with the configuration of the network. Devices sending data over a home or office network need an Internet Protocol (IP)-address. In most research the IP-address is configured by a controller or CPU. Configuration of the IP-address via a controller or CPU means there is still an FPGA and a CPU needed in one application. In all examples the FPGA is only running in the core of the network to process and send the data. At the user side of the network a micro controller or CPU is used to collect and show the data. This same approach is found in particular data centers, where the connection is managed by the FPGA [17].
For connecting to an Ethernet based network a number of IP-cores are available. The IP-core will connect the user application in the FPGA to a network interface. An example of a IP-cores for connecting to a Ethernet based network is the *Intel Triple-Speed Ethernet* IP-core [18], this IP-core can be used when connecting to a home or office network and will handle all Ethernet problems. All data from the user application connected to the IP-core will be send using an IPv4 or IPv6 frame. The core is made for two Ethernet protocols: UDP and a limited TCP implementation. The core can not handle IPv6 extensions, adding an implementation for this is not possible. The *Intel Triple-Speed Ethernet* IP-core

is a good option for most applications, however when you want to have an Industrial Ethernet application you need another Intel IP-core [4]. With this IP-core the largest Industrial Ethernet protocols are covered, however the Intel IP-core for Industrial Ethernet applications can not be used for sending frames over a home or office network. The IP-core is used in combination with the Nios processor on the Intel FPGA, the IP-core is used for the real-time communication while the processor provides the configuration and data.
The combination of hardware for Industrial Ethernet in combination with a processor is used in dedicated chips as well. An EtherCAT example where an ARM Cortex-M4 is combined with EtherCAT hardware in a single chip is the XMC series from Infineon [19]. The EtherCAT core is connected to the ARM via the internal databus. This bus is used for configuration and data exchange between the ARM and the EtherCAT core. The combination of the ARM and EtherCAT core has the advantage that an ARM Cortex running at 144MHz can communicate over a real-time industrial protocol where a cycle time is less then 100 µs without much load on the controller. There are more FPGA based solutions where this approach is taken. For example in the Softing core [20] where there is a micro-controller needed for configuration and to provide data to the Industrial Ethernet application. In this IP-core there is some overlap between the hard real-time Industrial Ethernet and more soft real-time EtherNet/IP or MODBUS TCP. The downside of this IP-core is that the user has to pick one protocol at the start of the project and can not work with two protocols at the same time.
In the Industrial Ethernet example the configuration is done using a micro controller, this gives the user flexibility to configure the protocol without using HDL. There are also measurements done on EtherCAT networks where the frame size is optimized [21], [22]. In some cases the optimization of the frames will give a higher throughput. As described there are all kinds of solutions to connect an FPGA to a network.
However, there is a problem with the more generic IP-core found, the support for TCP or DHCP is missing. There is a lot of coding needed to get TCP or DHCP to work. There are cores supporting DHCP and TCP, one example is found in the *open-cores* website [23], the IP-core can request an IPv4 address and make a TCP connection to a server. For TCP only one connection at a time can be made by the hardware implementation. The TCP or UDP frame will be send over SPI to a micro controller putting it in an IPv4 frame and sending it to the destination. All examples above resolve some part of an implementation for a complete Ethernet stack. However, non can handle both Industrial and home or office network communication at the same time. Most cores are connected via the Avalon bus or a custom bus. So there is space for a core where the networking is done in the user project and where expanding the stack is easy.
From the above we can conclude that there are advantages in using an FPGA in network applications. Especially when there is a lot of data processed in this network or when a low latency is crucial. In the given examples there is a trade off being made between flexibility and expandable.

3.2 Hardware-based solutions

In this research the Ethernet core is intended to be flexible and expandable. There has been research done on flexible and expandable Ethernet cores by others as well. Processing of UDP is done on an

FPGA in other research [24] where all configuration of the Ethernet core is done in one project. In the research data is send from the FPGA to a computer where the data is shown to the user. In the hardware implementation UDP is send over a gigabit link to stream video from a PC to the FPGA. The implementation presented in [24], is written in HDL so can be expanded, the implementation is some what flexible and generic. However, the implementation is only suitable for IPv4, there are no IP addresses used because the connection is made using a cross cable.

So there are parts of a complete Ethernet implementation missing. The second thing missing is a TCP implementation, for implementing TCP a state machine is required. There is no such state machine in UDP. When the given implementation needs an extension of TCP the state machine has to be implemented as well. So expanding this core with extra protocols will take more effort. In other research there is a complete TCP implementation [25]. This hardware based TCP processor called SiTCP runs a minimum TCP implementation on an FPGA. The core can handle both UDP and TCP as well as Address Resolution Protocol (ARP) and a limited Internet Control Message Protocol (ICMP) implementation. Everything is handled by the FPGA, the user application can write data to the core via a data bus. The core will make the TCP connection and will send the data over a Gigabit Ethernet connection. For testing the user application test data was stored inside the FPGA, however in principle this could be logic sampling data and provided it to the core. When a message is received correctly a PC connected via RS232 will be notified by the FPGA. For proving the speed to be $1Gb/s$ the system ran for 800 seconds at this speed. The measurement is done between a PC and the SiTCP where test data was send in both directions and between two SiTCP running on different FPGA boards. In the research a constant throughput of $949Mb/s$ between the two FPGA's was found, but only 60% of the data between the PC and FPGA had a speed of $949Mb/s$, the rest had a lower speed. The paper does not investigate why there is a difference, but the suspicion for the lower speed are the used coding technique.

The measured $949Mb/s$ seems to be low for a $1Gb/s$ connection. In the paper the difference is explained by the measured data throughput, the data is send in frames of 1460 bytes or 1518 bytes. The frames are Ethernet frames with all required headers, when the overhead produced by these headers is add to the measured data speed you will get a $1Gb/s$.

The paper shows that it is possible to create a complete Ethernet stack on an FPGA, even on a $1Gb/s$ connection. The paper also highlights the fact that making a reliable Ethernet stack on a PC is more complicated when the chosen coding technique is not sufficient. So all steps are taken to make a reliable Ethernet core in HDL on an FPGA, but only for UDP and TCP. The proposed Ethernet core could be expanded with other upper layer protocols, however implementing IPv6 will be a challenge because in the deframing standard patrons are used which will not variate, matching patrons works well on IPv4.

When we take a look at industrial Ethernet the speed is normally set to a $100Mb/s$ connection, the low speed is sufficient for industry right now, networks are relatively small and data send from nodes to the master is small. However, there is research done where a EtherCAT core is made on an FPGA using a $1Gb/s$ connection [26]. In this specific research the EtherCAT core is written in HDL with all aspects needed for EtherCAT implemented. User data can be send to the core and will be send to the master by the EtherCAT core. In principle Ethernet and EtherCAT can be combined on the same network, however this core only filters on EtherCAT, all other messages are discarded. Via a protocol

checker the core can make a distinction between EtherCAT and non EtherCAT frames. Between the so called MAC and the protocol checker is a loopback function implemented.
The loopback function is used in industrial Ethernet to forward messages to the correct port. In master slave systems discussed is section 2.1.4 forwarding is important. Messages from the master have to be processed by the slave core. When the slave core is done processing, the message will go back to the loopback function and will be forwarded to the next slave. Messages from the last slave going to the master have to be forwarded without processing, this message has been processed by the slave when it came from the master. The logic for forwarding messages to the correct port or slave core is located in the loopback function. In the last slave messages from the master are forwarded to the slave core. However, in this case there is no next slave so the message is 'forwarded' back to the previous slave. This forwarding is fundamentally different to standard Ethernet. The problem with the implementation given by [26] is that the loopback function does not check the message type. It just forwards all messages from the port connected at the master to the slave core without checking the type. This forwarding is no problem because the Protocol Checker can filter between EtherCAT and other protocols. The other way around, messages going back to the master are forwarded to the next slave without checking the type and are not processed again, because they where processed on the way from the master. In EtherCAT a slave can not make a message, it can only modify a message. So an Ethernet frame coming from a previous slave going to the master will never be processed. For a Ethernet frame, for example containing a IP message, which can be made by every node, the loopback function will have to check if the message has to be forwarded. The previous means that the slave will be a 'wire' for an Ethernet frame even when addressed to the node. To implement the forgoing a structural design change is necessary.
In all implementations discussed above there is one type of Ethernet messages playing a central role. However, according to the Ethernet standard it is possible to have different types of messages on one network. Although it is possible, it is difficult to make an implementation for different Ethernet types because there is a difference in implementation needed for the Ethernet types. Extending the implementations found to support multiple Ethernet types will be difficult in HDL, a solution is to use some HLS. Reading Ethernet frames from a network can be done by the P4 language. P4 is specially designed for Ethernet applications.
The above forms a good base for investigating whether Clash can be used to make a more complete Ethernet Core on an FPGA. The role of Clash will be comparable to the role of the P4 language, where a language with a high level of abstraction is used to generate code for programming an FPGA.

3.3 P4 language

One HLS used for Ethernet applications to generate HDL for FPGA's is the P4 language, it is not only used for generating HDL but also for configuring switches and other network appliances as well. In [27] the FPGA ia used with the HLS P4 to make an Ethernet parser. In the presented terabit implementation IPv6 extensions are combined with Virtual LAN (VLAN). The P4 language is only used to extract data from an incoming Ethernet frame. The user application can be connected to the data output and is written in HDL. The research presented in [27] is a good example of a HLS used to make receiving an

Ethernet frame easier. Unfortunately, not the complete user application is specified in P4 so there are still parts of the system that need to be coded in HDL.
P4 can also be used for measurements on networks, in [10] two large networks are monitored via a P4 implementation. From the research we know that there is a possible misconfiguration in IPv6 implementations which can result in security problems and vulnerabilities in the network. For finding the misconfiguration and to find the number of extensions used in IPv6 the two networks where monitored. There are two things that can be extracted from the research presented in [10]. The first is that interpreting an Ethernet frame using an HLS is possible, writing the complete user application in this HLS is not always visible. The second is that in some situations misconfiguration can lead to security problems, so it is important that the Ethernet application can be easily tested.

Chapter 4

Design

4.1 Introduction

In this project, the Clash language will be used for the decomposition and composition of an Ethernet frame to test its usability and stability. To do so, common patterns of different Ethernet protocols will be bundled together. The decomposition and composition will be discussed in the same order as the OSI-model presented in Chapter 2. Before discussing security problems of extension headers and options, some other parts of an Ethernet message have to be parsed. To our best knowledge there are currently no other projects in Clash where there is an Ethernet parser produced and tested. This has to be done first before solving the security problems. The structure of this chapter is as follows. First the connection between the cable and FPGA is described. The second part is about an Ethernet receiver on an FPGA, build layer by layer following the OSI-model. The last part describes the transmission of an Ethernet frame.

4.2 Ethernet connection

Connecting an FPGA to Ethernet is only possible via a so called PHY as explained in Section 2.4. In order to communicate with the PHY a special data bus is needed (MII or rMII). There are of the shelf building blocks available to do so on an FPGA. They can be used in Clash, however there is a project already available at QBayLogic where there is a connection made between a PHY and the FPGA using an ARM based processor running Linux.

4.2.1 Implementation of the receiver

Now all conditions are met to connect an FPGA to a network, the design of an Ethernet stack in Clash can start. To test whether it has advantages to use Clash for making a generic Ethernet stack, a basic implementation has been made. This implementation is not a complete system, however all parts of the implementation are constructed to meet the minimum requirements from the corresponding RFC's. The system must be able to identify whether a message is incorrect or whether a frame is unknown and should be discarded.
In this basic implementation the DHCP should work without any action from the user. The system will

try to receive an IP address from the server and will use the IP-address to check whether a message is addressed to the system. This way the user only has to send a message and the system will take care of calculating any necessary checksums and will add all addresses. In most systems the given implementation will save some the processor work and will speed up sending messages.

4.2.2 Data Link layer

The **Data Link** layer of the OSI-model handles the Ethernet header. An important functionality of the Data Link layer is the addressing of a package. The Data Link layer is not only part of servers and clients but is part of routers as well. To better understand the functionality of an Ethernet frame, the following sections will describe what an incoming frame can look like and what fields can be expected. The process of explaining what parts can be expected in a Ethernet frame will follow the structure of the OSI-model, starting at the bottom and going up, from cable to application. The way a frame is structured will be shown in Clash as well. The first layer found in the OSI-model is the **Physical** layer, this layer is taken care of by either an ARM controller or an IP-core. The data coming from the **Physical** layer will be processed in the **Data Link** layer and wil start with the Ethernet header. Data coming from the **Physical** layer will be processed per byte.

Ethernet header

The Ethernet header has a linear structure, and there are only two variants of this header. In Figure 4.1 a standard Ethernet frame is depicted, all field lengths are shown in bytes. In the figure there is a bold line around the header fields, only *Payload* is not considered as part of the Ethernet header. From left to right the first field is the *Preamble*, this field is used to find the start of a frame. Next are the *Destination* and *Source* addresses, both given as MAC addresses. These are used to identify the sender and the receiver. The last header field, the *Ethertype*, contains information about the protocol used for sending the *Payload*. Every protocol adopted as an IEEE standard has his own *EtherType* of 2 bytes. There is one exception, the so-called VLAN. In this case the header has been extended with 4 bytes; or 8 for double tagging (where two VLAN-fields are used in combination). As can be seen in Figure 4.2 there are two extra fields of two bytes added to the frame, together they are called the VLAN tag. Information in this field can be used to create different networks on one cable.
VLAN tags can also be used to give some frames priority over other frames, like a priority vehicle on the road. Figure 4.2 shows the VLAN tag, consisting of two parts: the *TPID*, which is the *EtherType* of VLAN and the *TCI*, which is used to identify the different virtual networks. To correctly interpret the *Payload*, all fields mentioned above are needed.
Both possible headers in Figure 4.1 and 4.2 start with block 2 and 3: the MAC address. The MAC address is an unique number stored in hardware. The MAC address is used to deliver the message to your computer. Like an address line on the envelope, although there is more information needed to get a letter to the correct person. This additional information is available in the higher layers of the OSI-model and will be discussed later in this chapter.
Figure 4.1 and 4.2 both contain a Header and a Payload, every upcoming layer will also have a header and a payload. So immediately after this header the payload will start with the header of the **Network**

layer. For the test the focus is on IP messages, so the content of the payload will be an *(IPv4/6) Payload* as shown in Figure 4.3. The Header of the Data Link layer, also called Ethernet header, in this figure refers to the header shown in Figure 4.1 and Figure 4.2. The Ethernet Header is located at the **Data Link** layer of the OSI-model as shown at the right of the figure.
The Clash implementation should contain the different fields mentioned above. The type made in Clash should represent three fields always located on the same position in the input stream: Preamble, Destination and Source. The *Ethertype* holds information about the structure of the Payload and can variate in length, so will be considered part of the payload.

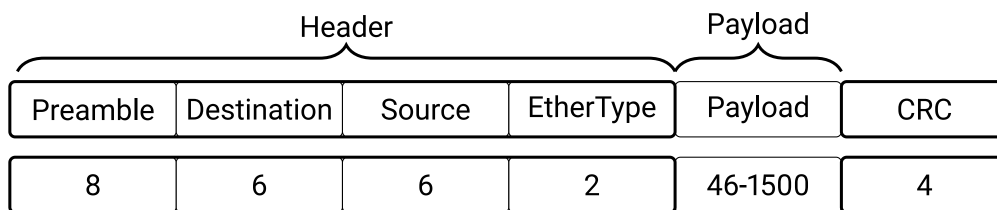

Figure 4.1: Common Ethernet frame in the data link layer, in Bytes.

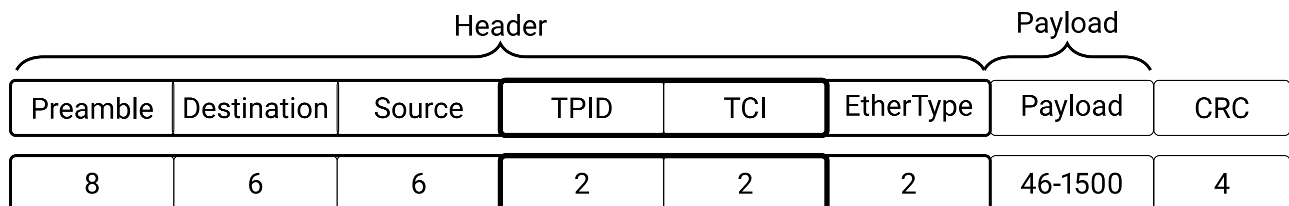

Figure 4.2: VLAN Ethernet frame, in Bytes.

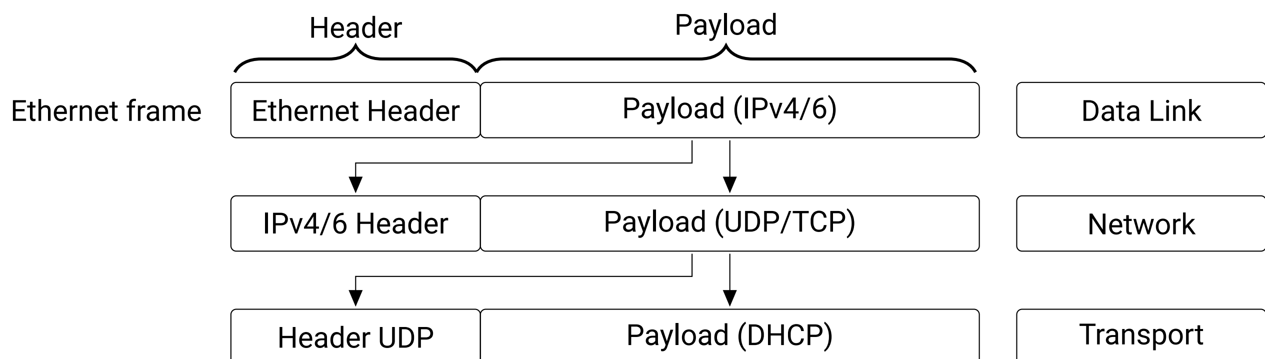

Figure 4.3: Stacking of protocols in an Ethernet frame, every layer has a header and a payload. The next layer is stored in the payload of the last layer.

EthernetHeader Clash

In the implementation the structure of a Clash type should follow the structure of an Ethernet frame. An Ethernet frame will contain a header and a payload. The first data type in the implementation is called *EthernetFrame*. The type *EthernetFrame* has two fields, one for the *EthernetHeader* and one for

the *EthernetType*, the payload of the message, this is shown in Listing 2. The type for the *EthernetFrame* has the same structure as presented in the top of Figure 4.3.
The type made for the *EthernetFrame* has two fields: the first field, *EthernetHeader*, should have a structure in which the header fields can be identified. In Figure 4.1 and 4.2 the first three fields are always on the same position. In the type for the *EthernetHeader* made in Clash the three fields are shown in Listing 3.
The name of the records in the code shown on lines 3 to 6 in Listing 3 are the same as the first three fields in Figure 4.1 and 4.2. The preamble in Listing 3 line 3 has the type *Vec 8 Byte*, this Preamble is made to store 8 bytes. In both Figure 4.1 and 4.2 the Preamble has a length of 8 bytes, so the record and the field match in length and name. The destination(MAC) and source(MAC) are present in both figure 4.1 and 4.2 and Listing 3 (lines 4 and 5).

```
1   data EthernetFrame = EthernetFrame EthernetHeader EthernetType
```

Listing 2: The Clash data type used to store the Ethernet frame.

```
1  data EthernetHeader =
2    EthernetHeader {
3      preamble :: (Vec 8 Byte),
4      destinationMAC :: (Vec 6 Byte),
5      sourceMAC :: (Vec 6 Byte)
6    }
7
8  headerDestin :: Byte -> EthernetFrame -> EthernetFrame
9  headerDestin byteIn (EthernetFrame header xs) =
10   (EthernetFrame (header{destinationMAC = (dMAC <<+ byteIn)}) xs)
11     where
12     dMAC = destinationMAC header
```

Listing 3: Header data type with a part of the state-machine used to fill the destination MAC address of the header type.

An Ethernet message will be received over time, in advance the length is unknown, there is only a maximum and a minimum length. Not knowing the length makes it hard to buffer the complete Ethernet frame. The first three fields of the frame will always be present in every frame, this means buffering the first three fields is possible because the position and length will always be the same. Although the implementation could fill a buffer with 20 bytes and then split it and fill the correct fields, the real implementation will immediately fill the fields of the header with the received bytes. The choice for filling the fields directly with the received bytes instead of using a buffer is made because some PHY's filter the preamble from a message. In the current implementation the reset value of the Ethernet core can be changed so the state-machine will start at the destination address.
When we take the other possible implementation, proposing a buffer of 20 bytes and then splitting the data and filling the records of the data type and changing the PHY, we will encounter a problem. Changing the PHY from filtering the preamble to not filtering the preamble, results in adapting the buffer size or shifting the data in the buffer to the correct position and updating the reset state of the

state-machine. In the current implementation only the reset state of the state-machine needs to be changed, resulting in less work to be carried out.
A small part of the state-machine code is shown in Listing 3 line 8 till 12, one byte of data is shift into the record holding the destinationMAC. The current destinationMAC is taken from the Ethernet header (line 12) and is filled with the incoming byte from the PHY (line 10). The EthernetHeader is the first type in the EthernetFrame shown in Listing 2, the EthernetType (the second field of the Ethernet-Frame) is not used and will be placed back in the Ethernet frame.
The above is done for the first 20 (or 12 when the PHY filters the preamble) bytes of a Ethernet message. So the first three fields of an Ethernet header as presented in Figure 4.1 and 4.2 are stored in the *EthernetHeader*, however the *EtherType* or VLAN-tag (presented by TPID and TCI in figure 4.2) is missing. To simplify VLAN, Figure 4.2 only has 4 bytes for the tag (TPID and TCI), however this can be extended to 8 bytes. Taking al three possible messages in consideration the payload can start between 2 and 10 bytes after the source MAC-address. The amount depends on the number in the first two bytes, for Figure 4.1 this is the EtherType, for Figure 4.2 the first two bytes after the source MAC address is the TPID.

TCI is not needed for information about the length of a payload. Information in TCI will tell something about the virtual network where the message is on and can give a priority to the message. TCI has a length of two bytes, these bytes can be split into three fields. The three fields all are represented by three records in a separate Clash type, the data type is named TCI as shown in Listing 4 (line 3 till 6). For now the three fields are only recognized in an Ethernet frame, there is no functionality implemented to use the information.
To receive an Ethernet frame the system should be able to identify all three cases: a EtherType, a VLAN-tag and a double VLAN-tag. The latter contains two VLAN-tags after each other, so for the base case there are two possible paths. Both paths are shown in Figure 4.4, all blocks represent a 2 byte number, except for the payload. The bytes are received over time, one byte per clock cycle. Translating the time aspect to the bytes received in Figure 4.4 means the payload will start 4 clock cycles later for the bottom message. To know when the payload starts depends on the EtherType or TPID, when the TPID is received the header will need 4 extra clock cycles. The EtherType will tell what **network** layer protocol is used, for example IPv4, IPv6 or EtherCAT. For the implementation a type in Clash should capture both VLAN and the **network** layer protocol. Incoming bytes on the FPGA should match with one of the **network** layer protocols or the TPID of the VLAN-tag, so the constructor of the type should match the number given to recognize the **network** layer protocol or the TPID.

```
1  data VlanTag =  VlanTag TCI
2
3  data TCI = TCI {
4    pcp   ::  BitVector 3,
5    dei   ::  Bit,
6    vid   ::  BitVector 12}
```

Listing 4: The Clash type of VLAN called VlanTag, consisting of a TCI. The TCI has three records as specified in the RFC, the total size is 2 bytes.

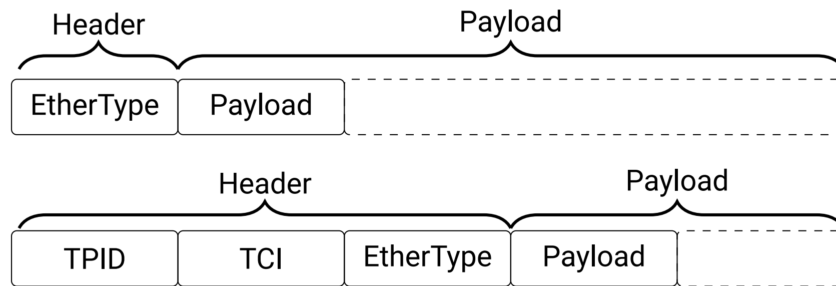

Figure 4.4: Common Ethernet frames in the data link layer, in bytes.

EthernetType

Constructing the Clash type for an EthernetType will result in a number of constructors, one for every protocol implemented in the system. The constructor can have a number of data fields, for IPv4 and IPv6 there will be a type for the header and one for the **transport** layer protocol. For VLAN there will be one field for the VLAN-tag and some fields for the other protocols, the other protocols could be a header for IPv4 or IPv6 but for some other protocols as well. The above is drawn in Figure 4.5, where there are constructors for IPv4, IPv6 and VLAN. In the figure all constructors will have an *IpHeader* and *TransportType*, VLAN will have an additional field for the VlanTag.
In the Clash implementation VlanTag is a data type with one field for TCI as shown in Listing 4 (line 1). For double VLAN, two VlanTags are in the type so there will be four data fields and one constructor. Figure 4.5 shows a structure to store data, every constructor matches with unique Ethertype value. The bit pattern of the Ethertype needs a translation to the corresponding constructor of the Clash data type.

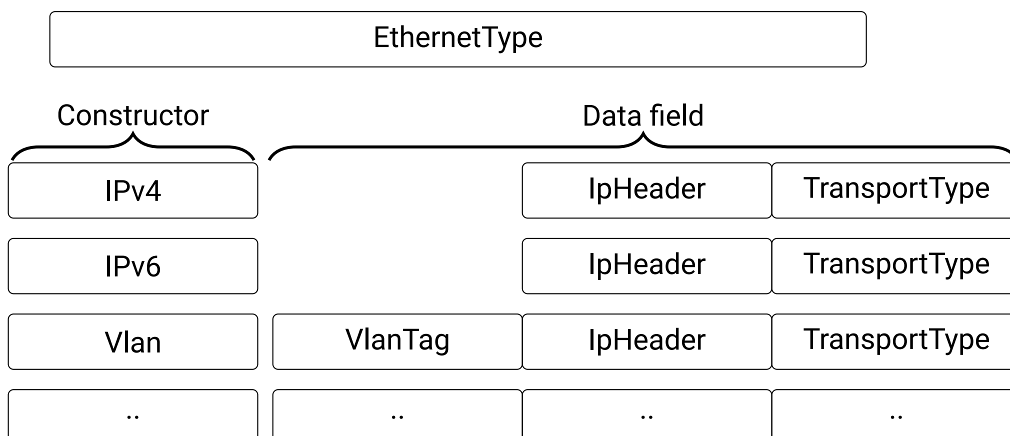

Figure 4.5: Schematic representation of the Ethernet type. Every message has a header combined with one IP-Header and a transport layer field.

```
data EthernetType
  = IPv4           IpHeader TransportType
  | IPv6           IpHeader TransportType
  | Vlan   VlanTag IpHeader TransportType
  | ...
```

Listing 5: EthernetType containing three constructors including the data fields for these constructors.

The Clash data type for the Ethertype in Listing 5 has the same structure as drawn in Figure 4.5. There are again only three constructors given, although there will be many more protocols. The given data type in Listing 5 is only a type in Clash, on the FPGA data is a bit pattern, *annotations* are used to make the translation between the two. Part of the *annotation* for the IPv4 constructor is given in Listing 6, this block of code is needed to make an *annotation* for line 2 of Listing 5. The complete type can be found in Appendix A.1, Listing 25. The complete annotation can be found in Appendix A.1, Listing 26. Part of the annotation shown in Listing 6 starts on line 2 with a formula to the length of a frame, which will be the length of the IP header and the length of the transport layer. The formula to calculate the length for the IP header can be found in section 4.2.3, for the transport layer the formula to calculate is given in section 4.2.4. So the lenFrame are two known lengths added up with the number 16, where 16 is the length in bits of the *EtherType*, the field present in the data that revers to a specific constructor in the Clash data type is called EthernetType. The EtherType will tell how the payload should be interpreted and will therefore be part of the EthernetType and is needed in the formula to calculation the length. Line 4 of Listing 6 will tell Clash the length of the data where the *annotation* is for, in other words how many bits are needed to fill the data structure.
A way in Clash to give meaning to the bits-stream in the FPGA is offered by the annotation, to get a better understanding of the annotation for IPv4 Figure 4.6 contains a graphical representation of the code given in Listing 6. The bit-stream drawn in Figure 4.6 on the line of the FPGA starts with $0x80$ followed by $0x00$, Clash will translate this exact bit-pattern to a IPv4 constructor, followed by an IpHeader and finally a TransportType. To do so, Clash needs to know the amount of bits there are in a stream, in the figure this length will be 16 bits, the lenIpHeader and lenTransport. The length formula is the same as represented by line 2 of Listing 6.

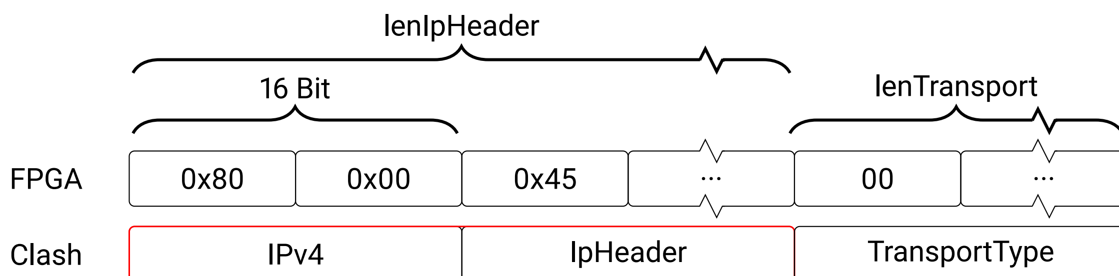

Figure 4.6: Schematic representation of an IPv4 bit stream translated to a Clash type.

Line 5 to 8 of Listing 6 represent how many bits are needed for every Data field in the Clash type. Starting at line 5 for the IPv4 constructor $0xffff$ bits will be used from the first 16 bits of data, so $0xffff$ will be shifted all the way to the left minus 16 bits ($lenFrame - 16$). Line 6 will tell Clash to

```
# ANN module (
  let lenFrame = lenIPHeader+lenTransport+16
  in ...
  (lenFrame)
    [ ConstrRepr 'IPv4 (shiftL 0xffff (lenFrame-16))
      (shiftL 0x0800 (lenFrameame-16))
      [ (shiftL ((2^lenIPHeader)-1) (lenFrame-lenIPHeader)),
      (shiftL ((2^lenTransport)-1) (lenFrame-lenIPHeader-lenTransport))]
    , [...]
]) #
```

Listing 6: Annotation for the IPv4 constructor of the EthernetType.

only use the IPv4 constructor (presented in line 2 of 5) when the first 16 bits of the data have the bit value $0x0800$, this number is registered by the IANA [28] as: "Internet Protocol version 4 (IPv4)". So line 6 will tell Clash that the following bits have to be interpreted as line 2 of Listing 5, because the constructor IPv4 is selected when the first 16 bits have the value 0x0800.
In figure 4.6 the first 16 bits on the FPGA are $0x80$ and $0x00$ so Clash chooses the IPv4 constructor. So the first 16 bits of data on the FPGA can be interpreted by Clash as an IPv4 constructor, but there are two data fields specified in the type as well.
Every field needs one mask to tell the width of bits needed in the data type. As presented in figure 4.6 by the red line around IPv4 and IpHeader the position will include the bits of the constructor in the IpHeader, this will come clear in section 4.2.3. So the IpHeader type will start at $0x80$ and will take all bits till lenTransport and has the length lenIpHeader. The mask can be calculated by $2^{lenIPHeader} - 1$ as used in line 7 of Listing 6. The mask has a width of *lenIPHeader* bits, and is positioned at the Least Significant Bit (LSB), however the fields in the figure are located between the Most Significant Bit (MSB) and the TransportType. So to get the mask at the correct position it will be shifted to the left by $lenFrame - lenIPHeader$ bits. In figure 4.6 all data in the first 4 blocks of the FPGA line can be interpreted by Clash. The positioning is in the code on line 7 of Listing 6 represented by the formula $lenFrame - lenIPHeader$.
There is a distinction between the constructor on line 5 and 6 of Listing 6 and the data field on line 7, there is no specific pattern needed of the data, only the constructor will take a pattern to match with. Figure 4.6 has a zigzag line in the curly brace below lenIpHeader and in the fourth node of the FPGA to visualize there are many more bytes. The same is done with the zigzag line at the lenTransport. The Transport type is the second data field of the IPv4 constructor, the position in the bit-stream is behind the bits of the IpHeader. Line 8 of Listing 6 will do the same for the transport layer as line 7 for the IpHeader. Line 8 of Listing 6 starts with the mask $2^{lenTransport} - 1$, which is the number of bits needed for the transport type. The bits in the transport type ($lenFrame - lenIPHeader - lenTransport$) are positioned behind the bits after the lenIPHeader and have the length of the Transport type. The code in Listing 6 will tell Clash how many bits and what bit order on the FPGA corresponds with the IPv4 constructor in line 2 of Listing 5.
There are two more constructors in Listing 5, of course IPv6 will have the same structure as IPv4, but IPv4 in Listing 6 line 5 will be IPv6 and 0x0800 on line 6 will be 0x86DD. So Figure 4.6 is the same for

IPv6, only the first two blocks of the FPGA will have the value $0x86$ and $0xDD$, for the line with Clash the first block will have IPv6 as the constructor name.
For the VLAN constructor of line 4 in Listing 5 the *annotation* will be different, there are three fields instead of two. Listing 7 shows the *annotation* for one VLAN tag. The length of a frame is identical to the IPv4 (and IPv6) length and is given in line 3 of Listing 7. Line 4 is the bit pattern ($0x8100$) for a VLAN message with two extra zero bytes at the position where TCI is located. Line 5 is the mask telling Clash only the first 16 bits are used for the encoding, the TCI bits "do not care" of the encoding. Line 7 will tell Clash the Vlan constructor is located in the first 32 bits, however only the mask (first 16 bits) are needed to select the correct constructor, the last 16 bits will not influence the choice of the selector. The complete annotation can be found in Appendix A.1, Listing 26.
Take for example Figure 4.7, the VLAN pattern is located in the first 16 bits of the bit-stream on the FPGA, the mask will select only the first 16 bits. Clash will match the pattern as being VLAN and expects the next 16 bits to be the TCI, after that the IpHeader and TrasportType will follow. Comparing the VLAN with IPv4 from Figure 4.6, the same bits for the IpHeader and TransportType are taken by Clash.

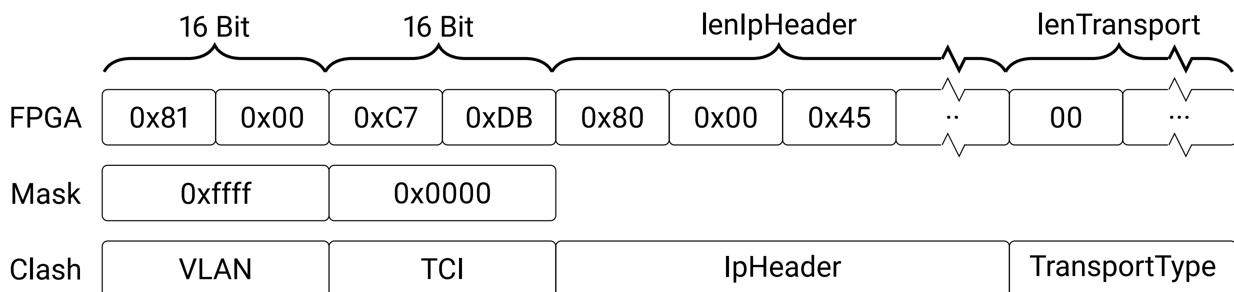

Figure 4.7: Schematic representation of VLAN in a bit stream translated to the Clash type.

```
1
2 # ANN module (
3   let lenFrame = lenIPHeader+lenTransport+16
4     vType   = 0x81000000
5     vTypeM  = 0xffff0000
6     ...
7     [ ConstrRepr 'Vlan (shiftL vTypeM (lenFrame-(32)))
8       (shiftL (vType) (lenFrame-(32)))
9       [ (shiftL(0xffff) (lengthFrame-32)),
10      ...]
11    , [...]
```

Listing 7: Annotation for the Vlan constructor in the EthernetType.

Line 8 of Listing 7 will position $0x8100$ correctly. Line 9 will give the position of the VlanTag (the first field of line 4 in Listing 5). The data will be in bit 16 till 31 (counted from the MSB). Line 10 of Listing 7 will be a copy of line 7 and 8 from Listing 6. There are 32 bits added to the formula to calculate the position because the type will be shifted 32 bits to the right. The 32 bits are the Vlan type and VLAN-tag, together they will shift the payload to the right as can be seen in Figure 4.4 where the payload

starts 32 bits later. The shift to the right is clearly visible in Figure 4.7 where the data starts 32 bits later compared to where the data in Figure 4.6 started. Clash will use the annotations shown in Listing 6 and 7 to get a bit-representation for the EthernetType shown in Listing 5. Every constructor in the type will have a separate annotation with approximately the same formula to calculate the length and position of the mask and data fields. There is no better readable solution to do this kind of annotation in Clash. For the Network layer comparable annotations are needed to select the correct constructor and make a distinction between IPv4 and IPv6.

4.2.3 Network layer

The Payload of the Ethernet frame will be used as an input for the Network layer. The choice for a particular Network layer protocol depends on the Ethertype. In a home or office environment this will most likely be an IPv4 frame, however IPv6 is increasingly common [10]. When designing a type for any Ip header it is good to know about similarities between IPv4 and IPv6 for the reuse of functions and data structures. Every line in the frame has a width of 32 bits, grouped by 4 groups of 8 bits (4 bytes). IPv4 and IPv6 can both contain options added after the header.
Options have to fill one or multiple lines of 32 bits. However, options can have variable lengths which can result in a line less then 32 bits. To get the option aligned to 32 bits, an option can add zeros (padding zeros). The padding zeros do not add extra information to the option but fill the empty space.

IPv4

The IPv4 standard was published in 1981 in RFC 791 [29], and has a linear structure. Figure 4.8 presented in RFC 791 is used to illustrate an IPv4header. Every header has at least five horizontal lines of 32 bits, however there is the possibility to add options at the end of a header. This will be elaborated in Section 4.2.3. The first line (line 0 in the Figure) starts with the *Version* field. The next part will briefly describe the function of the fields shown in Figure 4.8.

4 bits	**Version**	Should have the number four, because this header is for IPv4.
4 bits	**IHL**	Represents the length of the header in words of 32 bits. When there are no options (standard) this field should be the number a five, because a standard header has a length of five words containing 32 bits. At most it can be $15 \cdot 32 = 480$ bits.
8 bits	**Type_of _Service**	Can be used to get less delay or more throughput. In most packets this field will be set to zero. [a]
16 bits	**Total_Length**	Gives the total packets size. [b]

16 bits	**Identification**	In combination with the Source and Destination address and the protocol field this identification will help to identify frames which are used in the same fragmentation.
2 bits	**Flags**	Are used to avoid fragmentation or to identify when a packet contains the last fragment of a fragmented frame.
14 bits	**Fragmentation _Offset**	Is used to give the position of a fragment in the original packet. And can be used to rearrange the frames to form the original message.
8 bits	**Time_to_Live**	Is used to remove packets that can not get to the receiver within the specified time. [c]
8 bits	**Protocol**	Used to identify the upper layer protocol.[d]
16 bits	**Header _Checksum**	An addition of all header fields to check for transmission errors.[e]
32 bits	**Source _Address**	IP address used to know where a packet came from and where a reply should be sent to.
32 bits	**Destination _address**	Is the IP address where a packet is sent to.

[a] It can be used in a router to select a path where for example the throughput is higher, which is the logistics centre from the example of Section 2.2. This logistics centre can route the packet to a delivery service that can handle larger packets because they have larger trucks.
In RFC 3168 Explicit Congestion Notification (ECN) has been presented, this field is used to tell the receiver that there is a congestion on the path from the sender to the receiver. The ECN field uses the last two bits of the *Type_of_Service*. ECN can be used to select a new path to the receiver.

[b] This includes the Ethernet header and data. Historically the minimum supported *Total_Length* is 576 byte, however most nodes in a modern network will support larger packets as well.

[c] Every node will decrease the value by one (or the amount of seconds used to process, with a minimum of one). When this field is zero the packet will be discarded.

[d] This does refer to the next layer, the **Transport** layer. There are many protocols that can be used in the **Transport** layer, one is described in Section 4.2.4.

[e] The *Total_Length* and the *Identification* are not included in this addition. A more detailed description can be found in Section 4.2.3.

The IPv4 protocol is not hard to translate to a Clash type, because it is a linear protocol. The only tricky part is the *Header_Checksum*. To calculate the checksum a special adder has to be developed for adding any overflow bits. Adding the overflow will truncate the result to a 16 bit word. The underlying protocol for IPv4 and IPv6 will be the same, so the same function will be used for both. In section 4.2.6 a more detailed description for the checksum calculation is given.

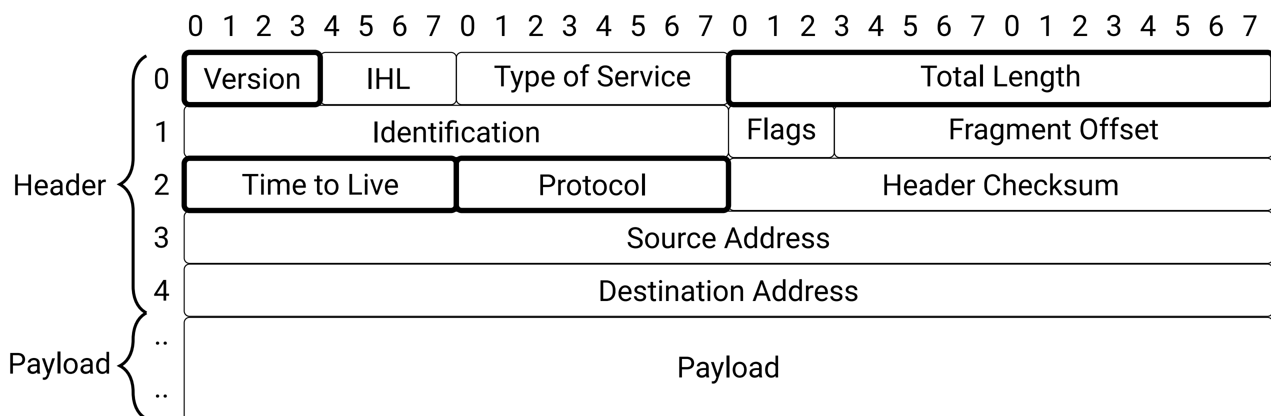

Figure 4.8: IPv4 frame

IPv4Header in Clash

The Clash type to store an IPv4Header is called IpHeader, a small part of the type is shown in Listing 8. Only the IPv4 part of the header is shown, all fields given in Figure 4.8 are in the listing, only the last two bits of *Type of Service* is split in ECN as described in RFC 3168. The order in Listing 8 and Figure 4.8 are the same, as well as the number of bits. Note, Byte and EthernetWord are 8 and 16 bits respectively.
The IpHeader type will have a similar annotation as the EthernetType type in section 4.2.2, a possible bit patron is shown in Figure 4.9. The line starts with 0x8000, the Ethertype for IPv4, and is used to select the IPv4Header constructor. The first record in Listing 8 is the version field, 4 bits holding the number 4. In Figure 4.9 the first number after the type is $0x45$ (8 bit), so the first 4 bits is $0x4$, matching the block in Figure 4.9 called ver. (short for versionIp) in the Clash line .

```
1  data IpHeader =
2    IPv4Header {
3      versionIp              :: BitVector 4,
4      ihlV4                  :: BitVector 4,
5      dscpV4                 :: BitVector 6,
6      ecnV4                  :: BitVector 2,
7      lengthIp               :: EthernetWord,
8      identificationV4       :: EthernetWord,
9      flagsV4                :: BitVector 3,
10     fragmentOffsetV4       :: BitVector 13,
11     timeToHop              :: Byte,
12     nextProtocol           :: Byte,
13     headerChecksumV4       :: EthernetWord,
14     sourceIPAddressV4      :: Vec 2 EthernetWord,
15     destinationIPAddressV4 :: Vec 2 EthernetWord
16   }
17   | IPv6Header ...
```

Listing 8: IpHeader header type, both IPv4 and IPv6 are combined in this type. Only the records for IPv4 are shown.

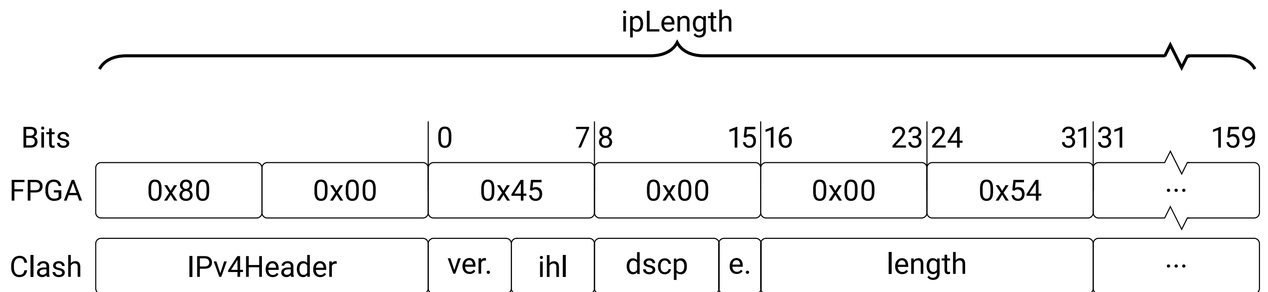

Figure 4.9: Schematic representation of the IPv4 bit stream translated to the IPv4Header constructor, part of the IpHeader type in Clash.

```
1  ipLength = 336+lengthExtension+lengthExtensionVec
2  ...
3  [ ConstrRepr 'IPv4Header (shiftL 0xffff ipLength) (shiftL 0x0800 ipLength)
4    [(shiftL(0xf) (ipLength-4)),
5     (shiftL(0xf) (ipLength-8)),
6     (shiftL(0x3f) (ipLength-14)),
7     (shiftL(0x3) (ipLength-16)),
8    ...
```

Listing 9: Annotation for the IPv4Header, part of the IpHeader header type.

The IpHeader has a similar *annotation* as the Ethertype, in Listing 9 a small part of the *annotation* of the constructor for IPv4Header is shown. Line 1 starts with a formula for the length of the IpHeader, 336 is the maximum number of bits for a IPv6 constructor including the 16 bit *annotation*, for IPv4 160 bits would have been sufficient. On the same line *lengthExtension* and *lengthExtensionVec* are added to ipLength, both are for IPv6 only. Line 3 of Listing 9 selects the constructor based on the first 16 bits, when the number is $0x0800$ the IPv4Header will be selected. Comparing Line 3 of Listing 9 to Figure 4.9 where the first two blocks are $0x80$ and $0x00$ on the FPGA, and because they are $0x0800$ Clash will treat the rest of the bits being the Clash IPv4Header constructor. Line 4 of Listing 9 selects the first record of the IPv4Header which is *versionIp* as shown in line 3 of Listing 8. The record *versionIp* is 4 bytes, or 0xf in hex, and is positioned after the constructor so these are the first 4 bits of data. In Figure 4.9 ver. revers to versionIp and is located at the first 4 bits of byte 0x45, so versionIp will have the value 0x4 in the stream on the FPGA.
Line 5 of Listing 9 will take the bits for the record called ihlV4, the value will be 0x5 in the stream shown in Figure 4.9. Line 6 is the dscpV4 record, the field is 6 bits and so the mask is 0x3F (0011 1111 in binary) and is shifted 14 positions from the MSB, so to Bits 8 to 14 in Figure 4.9. The last two bits of the same byte, bit 14 and 15, are the ecnV4, the fourth record of line 6 in Listing 8. To filter ecnV4 from the stream of data the mast 0x3 is used and shifted 16 bits from the MSB, shown in line 7 of Listing 9. There is a annotation for all records shown in Listing 8, so all 160 bits of the stream can be translated to a Clash type as shown in Figure 4.9. The complete annotation can be found in Appendix A.2, Listing 29.
Between the IPv4-header and a Transport layer protocol can be IPv4 options, in this implementation IPv4 options are not implemented. Although they are not implemented the basic structure is impor-

tant to understand because there are more protocols using comparable techniques for options and extensions.

IPv4 options

Figure 4.8 can have options, for example the option "Internet Timestamp", which can be used to record the time at which a system processed a datagram. There are two ways to format an option, a single *Option_Type*, shown in Figure 4.10, or an option containing data starting with a *Option_Type*, *Option_Length* and *Option_Value* (This is shown in Figure 4.11). The *Option_Value* can have a variable *Option_Length*, this is represented by the dotted line behind the *Value* field. Combining Figure 4.8, Figure 4.10 and Figure 4.11 will result in Figure 4.12. In this Figure the first five lines (0 till 4) are the same as in Figure 4.8. Line six (number 5 in the Figure) is used for option X. There are 3 bytes used for this option: *Option_Code*, *Option_Length* and *Option_Value*. To complete the line to a word of 4 bytes, the zero option is used. This option has the *Option_Code* 0 and an *Option_Length* of 1 byte. However all devices are allowed to use options, they are hardly ever used in a packet. Because of security issues most routers do not use options or drop packets with an option. RFC 7126 has a list of what should be done with IPv4 options and what the risks of using an option are [30]. So in most practical cases Figure 4.8 will be a complete IPv4-frame.
Both Figure 4.8 and 4.12 have a Header and a Payload, these are the same Header and Payload as shown at the second line in Figure 4.13. The payload of the IPv4 frame will be used in the **transport layer** and will again start with a header, this is a transport layer header.

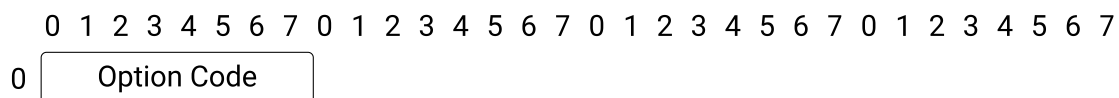

Figure 4.10: Single option of IPv4, used to get a line of 4 bytes.

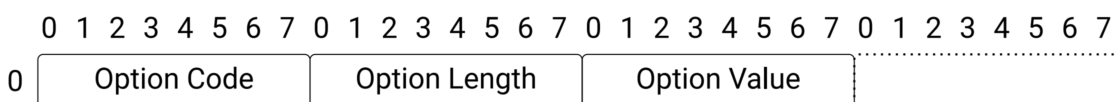

Figure 4.11: Option containing data of IPv4, the option value depends on the value in Option Length.

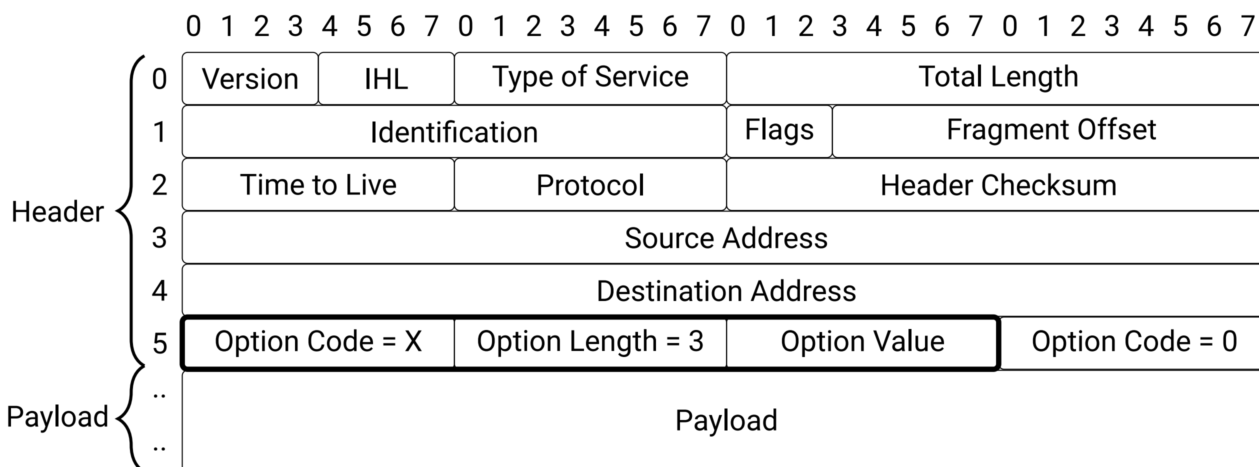

Figure 4.12: IPv4 frame with a option added between the Header and the Payload.

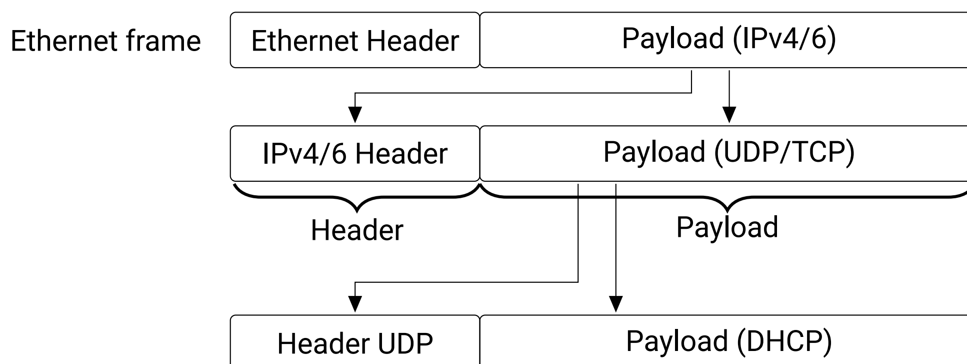

Figure 4.13: Stacking of protocols in an Ethernet frame, the Payload of IPv4 in the input for the upper layer protocol.

Checksum

The IPv4 *Header_Checksum* (introduced in Section 4.2.3) is the addition of all header fields, the *Total_Length* and *Identification* are not used in this addition. One of the fields used in this *Header_Checksum* calculation is *Time_to_live*. Every node handling the message has to decrease the value of *Time_to_live* by at least 1, as explained in Section 4.2.3. The consequence of this will be that the value of the *Header_Checksum* field has to be recalculated and decreased by the same amount. For an end node this will not be a problem, however a routing application is responsible for the recalculation
All information of the header is needed to parse the Payload of an IPv4 frame. The next layer of the OSI-model will handle this Payload, the *Option_Length* field of the IPv4-header will tell the amount of bytes send in the Payload. As described above there is a newer Internet standard, called IPv6, which can send the same type of Payload.

Address exhaustion

One of the reasons to start the development of IPv6 was the prospect of address exhaustion. There are only around 4,3 billion addresses available for all IPv4 nodes. In IPv6 the number of available

addresses is much larger, because the addresses are larger. It is expected that IPv6 will be more commonly used in the future. When a communication stack in hardware is made, implementing an IPv6 parser is a must.

IPv6

RFC 2460, released in 1998 [31], introduces IPv6. In 2017 it became standardised via RFC 8200 [32] to become the new IPv6 Internet Standard. In the future the standards will replace the functionality of IPv4 and will be used in the same environment and for sending the same type of payload. The most obvious differences between the IPv6 and IPv4 header is the length of the IP-address, so fulfilling the aim of having more addresses available. Secondly, the number of fields in IPv6 is less compared to IPv4. Figure 4.14 from the RFC presents this header, which can be extended by so called extension headers. The next part will briefly describe the function of the fields shown in Figure 4.14.

4 bits	**Version**	Should be six instead of four in case of an IPv6 frame.
12 bits	**Traffic_Class**	Used to define the priority or class in a frame. [a]
16 bits	**Flow_Label**	Used for real-time applications, and to track packet used in the same connection with a specific server.
16 bits	**Payload_Length** ‘	The length of all data after the header, including extension headers and Payload.
8 bits	**Next_Header**	Used to point to the protocol after the current header. [b]
8 bits	**Hop_Limit**	The number of routers that can be passed during the trip. [c]
128 bits	**Source_Address**	Used to know where a packet came from and where a reply should be sent to. [d]
128 bits	**Destination_address**	Is the address where a packet is sent to. [d]

[a] This field is comparable to the Type_of_Service from the IPv4 specification.
[b] It could be an extension header or a upper-layer protocol as used in the **Transport** layer, described in Section 4.2.4.
[c] This field is equal to *Time_to_Live* from IPv4 and is the number of hops a packet can make before it gets destroyed. Hops are the number of routers that can be past during the trip. There is one change made to the IPv4 *Time_to_Live*, every hop will decrease the counter by one, so when the processing takes more than a second the *Hop_Limit* will still be decreased by one.
[d] It has the same purpose as the IPv4-addresses but is longer.

Both headers serve the same purpose and have the same type of fields. IPv6 is mainly introduced to deal with the IP-addresses exhausting of IPv4 as described in Section 4.2.3. All fields of the IPv6 frame have to be parsed and stored on the FPGA, this is done in the *IPHeader* type. This is shown

in Listing 10, where the constructor for *IPv6Header* (on line 3) has less fields compared to IPv4. The fields shown in Figure 4.14 are in Listing 10 line 3 to line 16 as well, including all lengths. The names of the records differ from the ones in the figure.

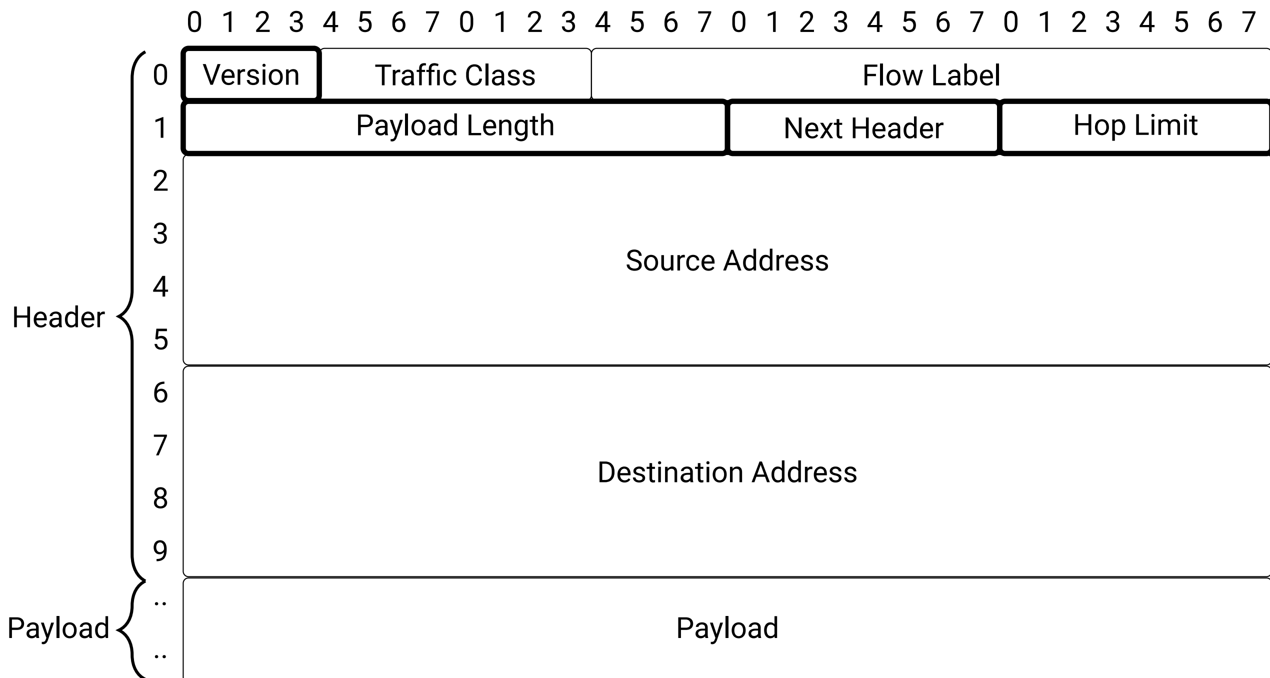

Figure 4.14: A schematic representation of a IPv6 frame, as presented in RFC 2460.

IPv6Header in Clash

The Clash type to store a IPv6Header is called IpHeader, a small part of the type is shown in Listing 10. Only the IPv6 part of the header is shown, not all records in de code are shown in Figure 4.14. The records *extensionHeadersIPv6Parser* and *extensionHeadersIPv6* are only in the code and not in the figure. All other records are present in the figure. To select the constructor for IPv6 the Ethertype is used, for IPv6 this is $0x86DD$. Figure 4.15 shows a possible stream of data on an FPGA representing a IPv6 message. The data starts with 16 bits: $0x86$ and $0xDD$, so a IPv6Header. Bit 8 till 32 are all zero and are not all shown. The last part of the message is cut as well. The IPv6 header can be followed by an extension header, this is where *extensionHeadersIPv6Parser* and *extensionHeadersIPv6* are used for. Bit 48 till 56 hold the *nextProtocol* or *Next Header*, in this example $0x2C$, meaning a *fragmentation header of IPv6*.

The length of an IPv6Header will be 336 bits, after the header zero or multiple extension headers can be send. The system will need space to store the extension headers, *extensionHeadersIPv6* will store all extensions received. The code in Listing 11 is the annotation for the IPv6Header constructor of the IpHeader data type in Clash. Line 1 will start with the length of 336 bits for the IPv6 header and will reserve space for the extensions. *lengthExtension* and *lengthExtensionVec* both have a value that will be calculated later in this chapter, for now assume they have a fixed value. The annotation in Listing 11 will continue on line 3 by matching the IPv6Header constructor to the patron $0x86DD$, the same value

is in the FPGA bit-stream of Figure 4.15. The first 4 bits of the IPv6Header is the *versionIp* in Listing 11 line 4 the first 4 bit ($0xf$) are taken from the MSB. In Figure 4.15 the FPGA stream has the value $0x6$ on bit 0 to 4, according to RFC 8200 the Version field should be six in case of an IPv6. The complete annotation can be found in Appendix A.2, Listing 29. Listing 11 line 5 will fill the *trafficClassV6* of the *IpHeader* in Listing 10. The *trafficClassV6* record is 8 bits starting 4 bits from the MSB till bit 12, in Figure 4.15 *trafficClassV6* is abbreviated by tra. and is $0x00$ in the input stream. All other record of the IpHeader have a comparable annotation.
There are some records in both the IPv4Header constructor and IPv6Header constructor, the fields can be of interest when the value has to be changed by the system, in this case one implementation can be used for both constructors.

```
1  data IpHeader =
2    IPv4Header ..
3    | IPv6Header {
4      versionIp                  :: BitVector 4,
5      trafficClassV6             :: Byte,
6      flowLabelV6                :: BitVector 20,
7      lengthIp                   :: EthernetWord,
8      timeToHop                  :: Byte,
9      nextProtocol               :: Byte,
10     sourceIPAddressV6          :: Vec 8 EthernetWord,
11     destinationIPAddressV6     :: Vec 8 EthernetWord,
12     extensionHeadersIPv6Parser
13       :: ((Vec (IPv6ExtensionLength NumberIPv6Extension) Byte),Byte),
14     extensionHeadersIPv6
15       :: (Vec SizeEthernetFrame   (IPv6Extention),Index (SizeEthernetFrame+1))
16   }
```

Listing 10: IpHeader header type, both IPv4 and IPv6 are combined in this type. Only the records for IPv6 are shown.

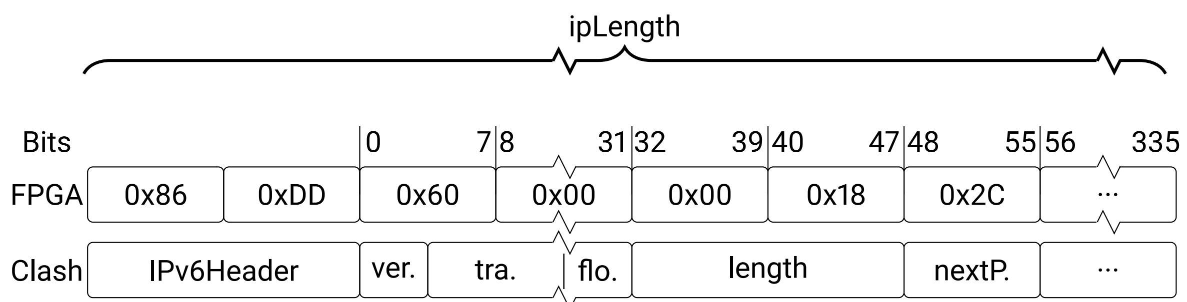

Figure 4.15: Schematic representation IPv6 bit stream translated to IPv6Header Clash type.

```
1      ipLength = 336+lengthExtension+lengthExtensionVec
2      ...
3    , ConstrRepr 'IPv6Header (shiftL 0xffff ipLength) (shiftL 0x86DD ipLength)
4      [(shiftL(0xf) (ipLength-4)),
5       (shiftL(0xff) (ipLength-12)),
6    ...
```

Listing 11: Annotation for the IPv6Header, part of the IpHeader.

Similarities between IpV4 and IpV6

There are four IPv6 header fields adopted from IPv4. These four are highlighted in Figure 4.8 and 4.14, they have matching lengths.
Version has the same position, length and purpose in both protocols, only the value differs (from 4 to 6).
Total_Length and *Payload_Length* both tell the number of bytes in a frame but for IPv4 the header is included, this is not the case for IPv6. For IPv6 the length of any extension header present is added to the *Payload_Length*.
Time_to_Live and *Hop_Limit* are in practice equal to one another but only when every node on a path takes less then two seconds to process a message. This will be the case in normal operations.
Protocol and *Next_header* have a same purpose, but whenever there is a extension header used in IPv6 this field will tell what the first extension header will be.
The four fields mentioned above can be found in Listing 8 and 10 for both constructors, the IPv4Header and IPv6Header.
Version is the record *versionIp* (Listing 8 line 3 and Listing 10 line 4).
Total_Length is te record *lengthIp* (Listing 8 and Listing 10 line 7).
Time_to_Live is the record *timeToHop* (Listing 8 line 11 and Listing 10 line 8), this is a combination of time to live and hop limit.
Protocol is the record *nextProtocol* (Listing 8 line 12 and Listing 10 line 9).
The **Source Address** and **Destination Address** in both headers have a separate record because the length does not match.

IPv6 extension header

A possible extension header is given in Figure 4.16. This header always has a *Next_header* field, equal to the *Next_header* field of the IPv6 header. This field tells whether the next part of the frame holds an extension header or an upper-layer protocol. One upper-layer protocol is described in Section 4.2.4. The *Ext_Data_Len* specifies the length of an extension header, this can be variable between extension headers or for the same extension header in different frames. There is a dotted line around the *Extension_Data* in the figure to represent the unknown amount of extension header data. In principle there is no limit on the number of extension headers, however the total length of all extension headers may not exceed the maximum frame size. The extension header length (in bytes) can be calculated by

Formula 4.1.

$$HeaderExtensionLength = (n * 8) + 8 \tag{4.1}$$

When going back to Figure 4.16 every extension header has to have a minimum length of 8 bytes, this is the mandatory offset $+8$ in the formula. All extension headers have to align in a 32 bits word so the size will increase by 8 bytes, this is why $n * 8$ is used.
The minimum extension header length should be 6 bytes, combined with the *Next_header* and *Ext_Data_Len* field this will result in 8 bytes. This minimum length is not included in the *Ext_Data_Len* field [33]. The n can be an arbitrary number, however the total *Ext_Data_Len* should fit within the maximum packet size. For good alignment a zero extension header should be used. The *Ext_Data_Len* field is an unsigned 8-bit number, it can never exceed 255 bytes, this is the upper-boundary for every extension header.
Assume there is no space limit in hardware, there is the possibility to have multiple extension headers in one frame, take them together as being m extension headers. The length can be calculated using the n defined in the paragraph above. Using this definition the space in hardware to store all possible extension headers can be calculated using the m and n. However, this can only be done at run-time. There is no way to know the *type* of a frame sent over a network at compile-time but hardware has to be allocated when compiling. This means m and n should be chosen sufficiently large. The value of m can be estimated by viewing the current adoption of IPv6 extension headers in common network-setups. There have been measurements done on two large networks, from this research the possible amount of extension headers in a frame is known. Most frames do not use any extension headers, less than 1% has one header and close to 0% have two headers [10]. So a practical choice of m would be two or more extensions, in most cases this will be sufficient. In the same research the *type* of headers send over the network has been captured, there are three commonly used headers: Ipv6-fragmentation, Encapsulating Security Payload (ESP) and Hop-by-Hop Options (HOPOPT). Only in an ESP extension header the length is variable [34], the minimum mandatory size equals 16 bytes. For Ipv6-fragmentation the length is fixed to 8 bytes. So $n > 4$ will be a good start for the system, however $n < 64$ will be a good upper-bound to stay within 255 bytes. There is a risk to select m and n at compile-time to the minimum estimated, especially when an implementation is used for several years, there is no way to predict the adoption of headers in the upcoming years.

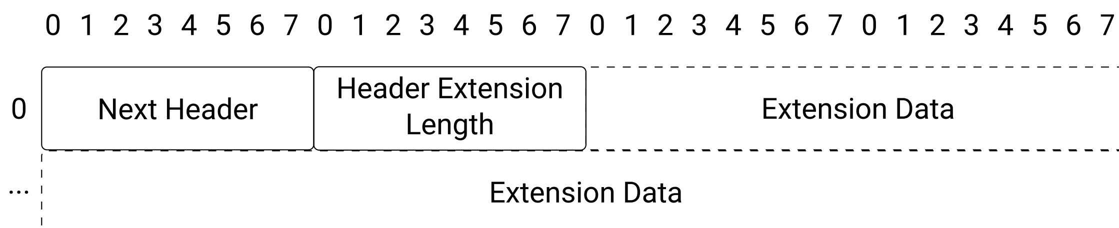

Figure 4.16: Schematic representation of an IPv6 extension header.

From the same research we know that there are security problems with parsing the extension headers used in IPv6. There are different problems but most have to do with incorrect configuration. In for example firewalls this can result in unwanted behavior like accepting a frame when it should be

discarded [10].

When using an IPv6 extension header two situations can be identified, a fixed length or a variable one. There are always two fixed fields present in an extension header as described above.
The forgoing is best explained by an example, in the section about IPv6 Figure 4.15 the nextProtocol has the value $0X2C$. The Fragmentation extension has the value $0X2C$, so the next header in the IPv6 frame given in Figure 4.15 is a fragmentation extension. The fragmentation extension has a fixed length of 0, using Formula 4.1 the length will be 8 Bytes. Every extension has 2 bytes reserved for the next header and length field as shown in 4.16, so the fragmentation extension will have 6 bytes of data. Figure 4.17 shows a possible fragmentation extension, there are 8 bytes of data in the header from bit 0 up to bit 63. The value $0X2C$ is a copy of the IPv6 header. The next protocol has the value $0X11$, the length has the value $0X00$, so a total size of 8 bytes. Bit 32 up to 47 are all $0X00$ and have been shortened for a better fit. The forgoing is a good example of an extension with a fixed length, however there are extensions which have a variable length.

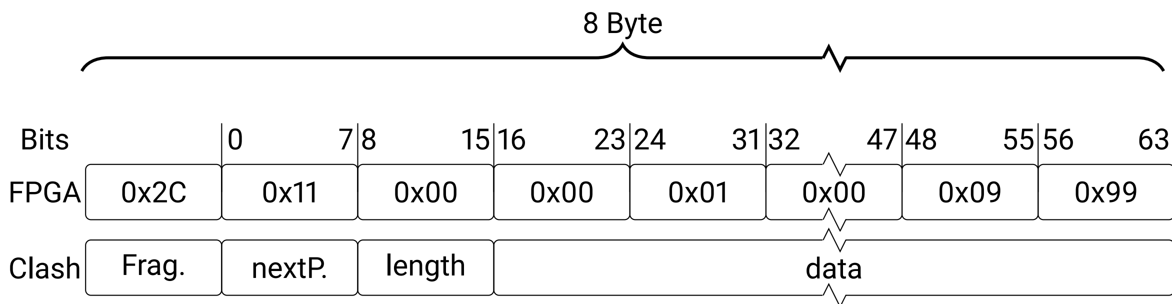

Figure 4.17: IPv6 fragmentation extension, the length is fixed to 8 bytes, every rectangle on the FPGA represents one byte.

For extensions with a variable length n Bytes will be reserved, an example is the Hop-By-Hop extension. In Figure 4.18 a stream of bits on the FPGA is coming in, the length is not known by Clash so the data field will have a length of *length extension*. The first byte on the FPGA will again tell what the next protocol will be, in the example $0X11$. The length of the extension is $0X00$ so again 8 bytes when using Formula 4.1, however there are *length extension* bytes reserved. What will happen in this case is shown in the line 'Stored' in Figure 4.18, the next protocol and length will be stored first. The extension data will be shifted from the right in the bytes reserved for the extension data. The result will be a extension type starting with two 8-bit values, followed by $lengthextension - 8$ bytes of unknown data and finally the 6 bytes of data send by the extension.
There is also the possibility to send more than one extension in a IPv6 frame, these is called chained extensions. The concept of receiving is equal to the examples in the previous section, so this is repeated as often as there are extensions present in the message.

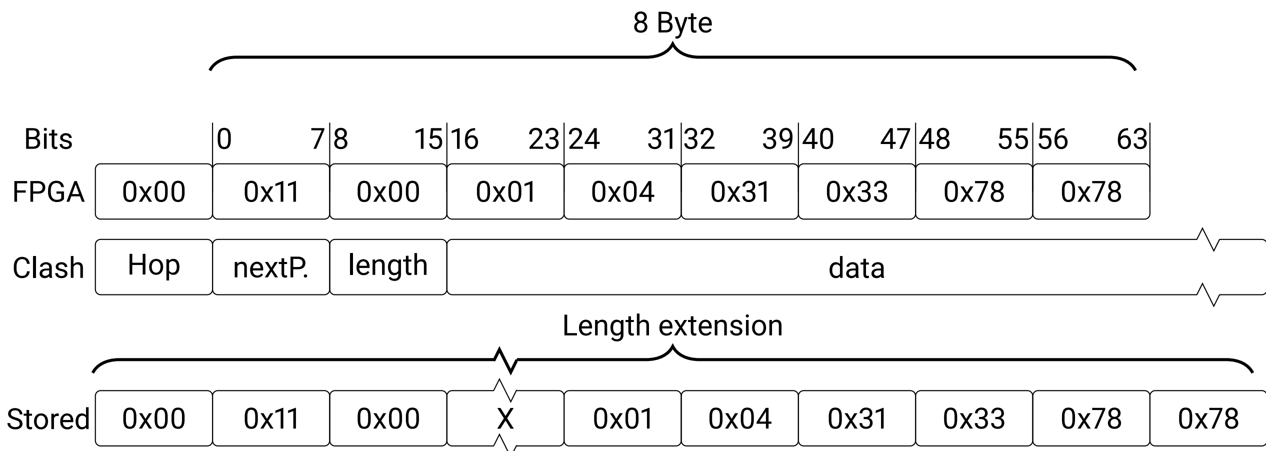

Figure 4.18: IPv6 extension Hop-By-Hop, the length of the extension is unknown, so it is possible to store undefined data.

```
data ExtensionHeaderIPv6 m = ExtensionHeaderIPv6 {
  nextExtensionHeaderIPv6 :: Byte,
  headerLengthIPv6        :: Byte,
  headerPayloadIPv6       :: Vec m Byte
  }

data IPv6Extention
  = FragmentIPv6 (ExtensionHeaderIPv6 6)
  | HopByHopIPv6  (ExtensionHeaderIPv6 IPv6ExtensionLength)
  | WildcardIPv6 (ExtensionHeaderIPv6 IPv6ExtensionLength)
  ...

  ...
  let maxOptionLength = (snatToNum (SNat @(IPv6ExtensionLength)))
    maxLen = maxOptionLength*8
    posLen = (shiftL 0xff (maxLen))
    nexLen = (shiftL 0xff (maxLen+8))
    ...
    [ConstrRepr 'FragmentIPv6 (shiftL 0xff (maxLen+16)) (shiftL 0x2C (maxLen+16))
      [nexLen + posLen + (2^(8*6)-1) ],
    ConstrRepr  'HopByHopIPv6 (shiftL 0xff  (maxLen+16)) (shiftL 0x00 (maxLen+16))
      [nexLen + posLen + (2^(8*maxOptionLength)-1) ],
      ...
```

Listing 12: IPv6Extention type and annotation, the data will be part of the clash type IpHeader.

IPv6 extension header in Clash

The Clash implementation has support for three extensions, two examples are the fragmentation extension shown in Figure 4.17 and the Hop-By-Hop extension shown in Figure 4.18. Both extensions have the same structure, one field for the next protocol, one for the length and one for the data. The latter can have a variable length as explained above, Listing 12 line 1 to 5 shows the *ExtensionHead-*

erIPv6 type with the three records. Line 4, *headerPayloadIPv6*, is a vector of bytes storing the extension data, in both Figure 4.17 and 4.18 this will be bit 16 up to and including bit 63 on the FPGA stream. *nextExtensionHeaderIPv6* is the next protocol, represented by bit 0 to 8 in both Figure 4.17 and 4.18. *nextExtensionHeaderIPv6* will tell what protocol will follow after the extension, the protocol can be a upper layer protocol or an extension. *headerLengthIPv6* tells the length of the current extension, represented by bit 8 to 16 in both Figure 4.17 and 4.18. There is no constructor for the different extensions in the data type, because the length of an extension can variate. The *ExtensionHeaderIPv6* extension will have a length m for the number of bytes in the data field.
The next type in Listing 12 line 7 to 11 will add the length to *ExtensionHeaderIPv6*, for example line 8 will construct *FragmentIPv6*, the fragmentation extension, with a length of 6 bytes. The second example is *HopByHopIPv6* on line 9, this extension will have a length of *IPv6ExtensionLength*, or in other words the maximum length an extension can have in the implementation. The last important constructor is the *WildcardIPv6*, this will be used for all unknown extensions. The unknown extensions will be saved by the system and the upper layer protocol in the frame can still be processed, the system will reply to the sender with a error. The complete IPv6Extention type can be found in Appendix A.3, Listing 28, Listing 29 is the complete annotation for the IPv6Extention type.
The last part of Listing 12, line 13 to 23, is the annotation of *IPv6Extention*. The *maxOptionLength* is the number of bytes the data can at most have. *maxLen* is the number of bits the data can at most have. *posLen* (line 16) is the position where the length field is in the data, so bit 8 to 16 counted from the MSB, shown in both Figure 4.17 and 4.18. *nexLen* (line 17) is the position where the next protocol field is in the data, so bit 0 to 8 counted from the MSB, shown in both Figure 4.17 and 4.18.
The constructor for FragmentIPv6 and HopByHopIPv6 are in line 19 and 21 of Listing 12 respectively, both correspond to the first byte of Figure 4.17 and 4.18. Line 20 and 22 of Listing 12 will give the position of the next protocol and length field, corresponding to bit 0 to 8 and 8 to 16 in Figure 4.17 and 4.18. The data field on line 20 of Listing 12 takes the last $8*6$ bits from the stream, for Figure 4.17 these are bit 16 to 64. For line 22 of the listing the bits taken for the data field are $8 * maxOptionLength$, so in Figure 4.18 this is the width of the data field.

Chained extensions

There can be multiple extension headers used in a single frame, this is shown in Figure 4.19, there are two examples shown. The one on top of the figure is a normal IPv6-header and some upper-layer UDP protocol, the *Next_Header* field will point to the UDP Payload. At the bottom of this figure the same UDP Payload is sent but there are two extension headers between the IPv6-header and Payload. The *Next_Header* field of the header will hold the value for "Routing header", this is the first extension header. There again is a *Next_Header* field in the "Routing header" and this will point to the "Fragment header". This extension header will start with a *Next_Header* field as well and points to the upper-layer protocol, in this example a UDP Payload. There could be any number of extension headers used in a frame. The last *Next_Header* field will always be an upper-layer protocol, there are various upper-layer protocols but in this example only UDP is used.
In Figure 4.19 the "Routing Header" comes first, this is recommended, but the nodes must accept any order . There is one exception to this, the "Hop-by-Hop Option" must always appear right after the

IPv6-header [32].

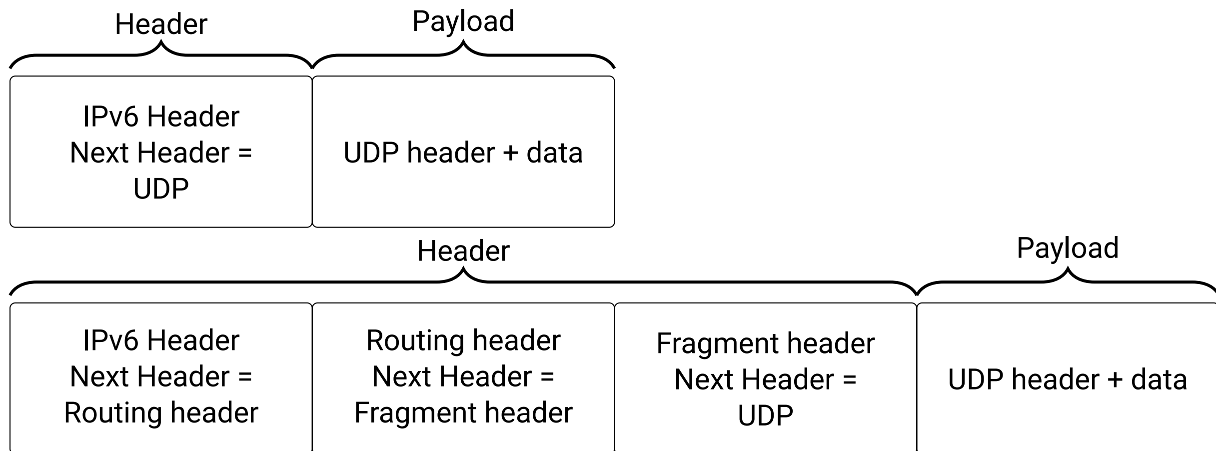

Figure 4.19: IPv6 chained extensions, the top image has no extensions, the bottom image has two extensions between the Header and the Payload. The Next Header field will point to the next extension header or Payload.

Header and payload

For IPv6 all header fields and extensions are part of the header. IPv6 and IPv4 share the same layer in the OSI-model. The header in Figure 4.14, 4.16 and 4.19 all refer to the same 'Header' as in Figure 4.13. The same holds for the Payload. The payload is used in the next layer of the OSI-model. In the IPv6 implementation all extension headers are considered part of the payload. For the implementation a clear distinction between different layers is necessary. For this research, extension headers are considered part of the IPv6 header.

Other protocols

In a structure like the one above more protocols can be included without changing the parser. For example when a new protocol has the same implementation for the Transport protocols, the same selector can be used. So, whenever there is a new IPv standard, only a parser for the header information needs to be written, assuming the same Transport layer will be used.

4.2.4 Transport layer

The next layer in the OSI-model (Section 2.2) is the Transport layer, the so called upper-layer protocols are used to handle data. UDP is one of the upper-layer protocols, and is used to transmit data in an Ethernet frame without much verification. UDP will send a message and the receiver does not have to send any acknowledgement whether the message has been received correctly. There are also upper-layer protocols that do use verification, an example is TCP where a state-machine is used to open a connection, to send data and to close the connection again. Every step will be acknowledged by the receiver to inform the sender that a message arrived correctly. There is a list of "Assigned Internet

Protocol Numbers" where over a hundred registered upper-layer protocols are listed [35], [36]. This same list has all registered IPv6 extension headers. From this list the UDP protocol is one of the easiest to explain and implement.

UDP

UDP published in 1980 in RFC 768 [37] is one of the upper-layer protocols. Figure 4.20 presented in RFC 768 is used to illustrate an UDP header. The header has two lines having four fields of two bytes. The next part will briefly describe the function of the fields shown in Figure 4.20.

16 bits	**Source_Port**	Is optional and can be used to indicate the port of the sending process. [a,b]
16 bits	**Destination_Port**	The port where a message is addressed to. [b]
16 bits	**Length**	Holds the length in bytes of the header and data of the UDP message.
16 bits	**Checksum**	An addition of all header fields and data to check for transmission errors.

[a] When a reply is sent, this port is the destination for the replier, when the Source_Port is not used it will be set to zero.
[b] There is a list of "Transport Protocol Port Number" registered by Internet Assigned Numbers Authority (IANA) [38]. The ports are used to give the information about the application where a frame is used.
[c] Calculation for IPv4 is not mandatory, because there is a *Header_Checksum* in the IPv4-header. For an UDP frame sent vian IPv6 it is mandatory to calculate the *Checksum*, because it can not be calculated in an IPv6 header. For the UDP *Checksum* calculation some fields from the IPv6 header have to be taken in account as well, this is done using a pseudo-header [32].

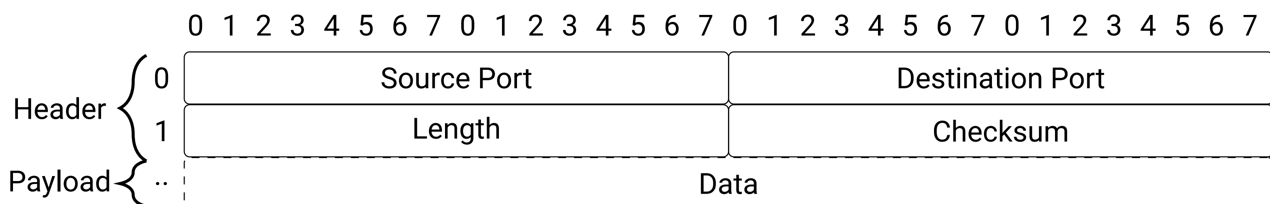

Figure 4.20: UDP frame, the first four fields are the UDP header, data inside UDP can have a variable length.

UDPHeader

The datatype for the UDPHeader shown in Listing 13 (line 5 to 9) has the same fields as Figure 4.20. The complete type can be found in Appendix A.4, Listing 30. There is not a custom bit encoding in Listing 13 because the position where the bits for the encoding are differs between the two IP versions. The encoding can be based on the *nextProtocol* field for IPv4. In the case of IPv6 it can be either in the *nextProtocol* field or in the *nextExtensionHeaderIPv6* field. The position of the *nextProtocol* will be different between IPv4 and IPv6. So the encoding will be based on information present on one of the three positions. The *UDPHeader* is part of the *TransportType* (line 1 to 3). There is also a fallback type for the case of an unknown *TransportType*, in this case all non UDP packages. This fall back only works for IPv4, in the case of IPv6 all not implemented transport types will be seen as an extension header and will be parsed using the *WildcardIPv6* as discussed in section 4.2.3. Parsing the transport layer header as an extension header is one of the downsides of having an incomplete implementation, but it could be prevented by adding all transport types or all extension headers to the selector for the state of the parser.

```
1  data TransportType=
2    UDP UDPHeader
3    ..
4
5  data UDPHeader = UDPHeader {
6    sourcePortUDP        :: EthernetWord,
7    destinationPortUDP   :: EthernetWord,
8    lengthUDP            :: EthernetWord,
9    checksumUDP          :: EthernetWord}
```

Listing 13: UDPheader type in Clash, every record is two bytes.

Data inside a UDP message will be sent to an FPGA blockram. This data can be used in the Session layer. Every byte sent to the blockram will be added to the Checksum. After receiving all data from a message, the system will set the validation flag for the other layers. This will speed-up the process for a Session application because it can start processing before each byte is checked.
There will be multiple applications in the Session layer; these could be on the FPGA but parts could be on a CPU as well. The Transport layer will write data to blockram without knowing the upper-layer protocols. This will be a problem for the next layers, because the blockram has only one output port to read data from the blockram and multiple readers can be present in the upper-layer. To manage this a Memory Management Unit (MMU) has been developed; implementation details can be found in Section 4.2.7. When writing data to the blockram there is no problem because only the receiver state-machine will write data to the blockram. This state-machine will only parse incoming messages from one Ethernet port, so it is not possible to write multiple messages.

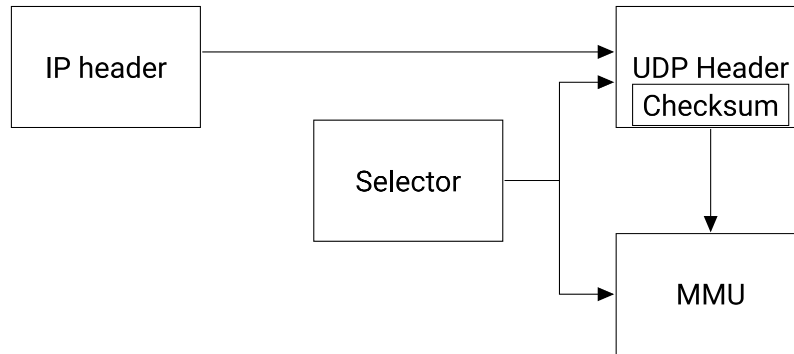

Figure 4.21: Block diagram of the UDP system in combination with the MMU.

The block diagram in Figure 4.21 shows all components connected to the UDP system. Data going from the selector will be used in both the MMU and Checksum. To calculate the last one some information from one of the IPv-headers is needed to create a pseudo-header as described in Section 4.2.4. The header will keep track of the Checksum and will update it, the MMU will only get a signal when the check is not successful. At this point the Session layer will need to flush all data read from the MMU.
Up till this point all data was always aligned in two groups of 16 bits, but a message could have an odd number of bytes. The 16-bit adder used for calculating the Checksum can only handle 16 bits words, so the last byte will be extended with a zero byte to meet requirements as specified in the RFC [37].

Checksum UDP

To calculate the *Checksum* all fields from the UDP header and pseudo-header (a number of fields from the IP header presented in section 4.2.3) have to be added together. The fields from IPv6 that should be used in this calculation are shown in Figure 4.23. This *Checksum* does not include the *Hop_Limit* or *time_to_live* field like in the IPv4 *Header_Checksum*, so there is no need for a hop to calculate or update the *Checksum*. In section 4.2.6 a more detailed description for the checksum calculation is given. Although it is not mandatory for IPv4 this same calculation can be performed on an IPv4 header. The pseudo-header to perform this calculation on an IPv4 header can be found in Figure 4.22.

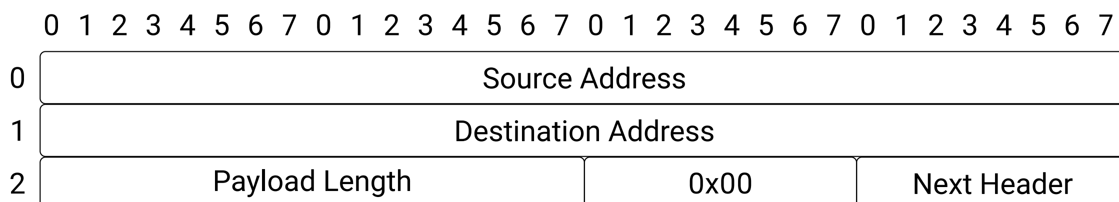

Figure 4.22: UDP IPv4 pseudo header needed to calculate the UDP checksum.

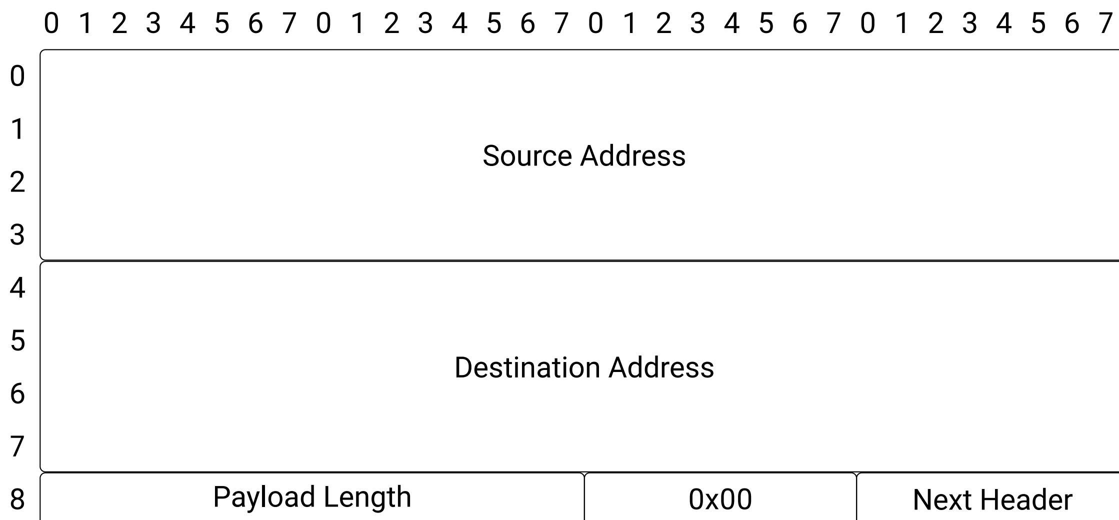

Figure 4.23: UDP IPv6 pseudo header needed to calculate the UDP checksum.

Other protocols

There are many more protocols on the Transport layer but only UDP, TCP, DCCP and SCTP share the same list of port numbers. There are of course more upper-layer protocols. in most of the upper layer protocols the checksum calculation is similar to the calculation presented above.

Header and payload

The Header in Figure 4.20 is the same Header as indicated on the last line of Figure 4.24. The Payload is the same as well. This payload will be used in the Session layer.

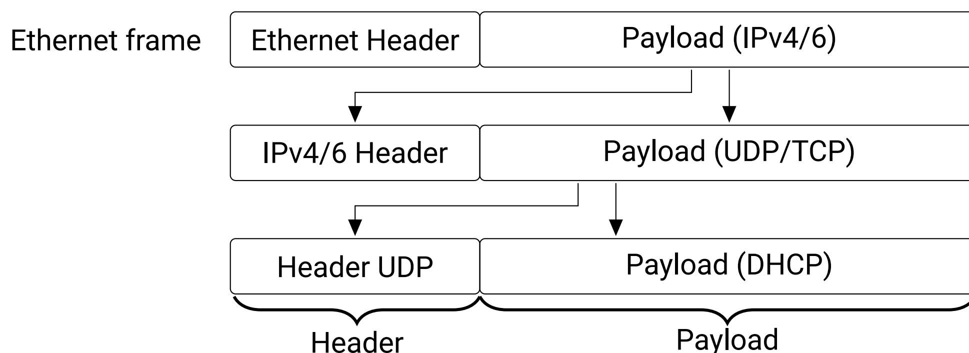

Figure 4.24: Stacking of protocols in an Ethernet frame UDP header and frame.

New protocols

For adding a new protocol the MMU we will need an extra output port to provide data for the new protocol. This will take more effort compared to extending the Network layer.
Now all protocols to send or to receive a message are implemented. Now there is a way to get a message from an Ethernet wire all the way to an UDP parser. The system can even handle extension

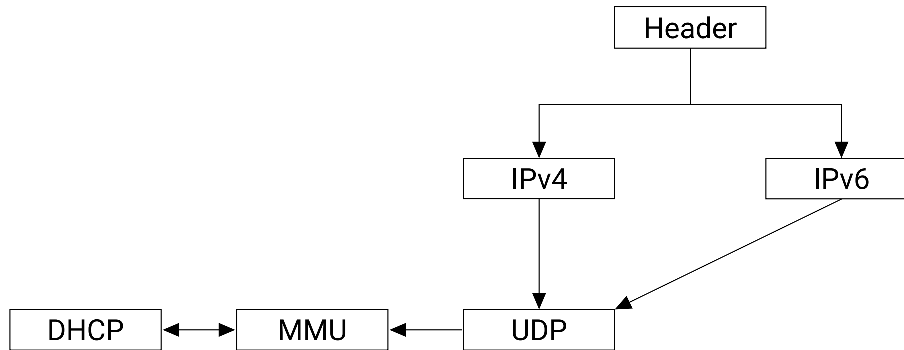

Figure 4.25: Schematic representation of the Ethernet type, every message has a header combined with one IPv header, one Transport layer header and is used by DHCP.

headers and will filter out incorrect messages. A test message is needed to proof that the system is able to communicate with a server. This could be a request from the system to get some data from a server. In this test case the DHCP protocol will be used. This DHCP protocol has four states: discovery, offer, request and acknowledge. It uses the UDP protocol so the system should be able to handle this UDP message. In the DHCP parser all information obtained by all other parsers is needed to send and receive a message, this is shown in Figure 4.25. There is not a direct connection between UDP and DHCP because the MMU will handle all communication from one to the other. This will be two different types in the system. For a sender or receiver information stored in both of the types will be needed to get from one state to the other.

4.2.5 Session layer

The next layer is the Session layer, most parts of this layer are handled by the user application, except DHCP. The DHCP is needed to get a IP address from the system and is implemented as a test-case for the system.

DHCP

The first thing done by most devices connected to a network is registering an IP-address, as explained in Section 2.1.1. For IPv4 this can be done via DHCP. In Section 2.1.1, a simplified version of a DHCP request is shown. There are four states distinguished, which is shown in Figure 4.30. In this Figure the *Client* is a device trying to obtain an IP-address, the *Server* is a DHCP-server. Every state has the same structure of a message. Figure 4.26 shows a complete frame, the first 109 rows are mandatory and a number of options will follow. The amount of options depends on the request and support of the options, and the total length of the DHCP message can never exceed the *Length* given in the UDP header (Section 4.2.4). The next part will briefly describe the function of the fields shown in Figure 4.26.

8 bits	**Op**	Message op code or message type. [a]
8 bits	**Hdr_type**	The hardware type, 1 for Ethernet.
8 bits	**Hdr_add_length**	The length of the address, this is 6 in the case of an Ethernet MAC-address.
8 bits	**Hops**	Set to zero by the client. [b]
32 bits	**Xid**	The *Transaction_ID*, a random number chosen by the client.
16 bits	**Secs**	Holds the *Seconds_Elapsed* since the client started the process.
16 bits	**Flags**	Flags,have to be set to zero.
32 bits	**Client_IP_address**	IP-address used by the client. [c]
32 bits	**Your_IP_address**	IP-address assigned by the server.
32 bits	**Server_IP_address**	IP-address of the server.
32 bits	**Relay agent_IP_address**	IP-address used in booting via relay agent.
48 bits	**Client_MAC_address**	MAC-address of the device requesting an IP-address.
592 bits	**Panding hostname/server**	For Bootstrap Protocol, set to zero for DHCP.
1024 bits	**host/File**	For Bootstrap Protocol, set to zero for DHCP.
32 bits	**Magic_Cookie**	Four bytes are the start of the *Option* field.
	Options	There are a number of options that can be requested by the device. [d]

[a] This field should be 1 for a request, and 2 for a reply.
[b] This field is optionally used by relay agents.
[c] When newly connected to the a network this field is set to zero, used in BOUND, RENEW or REBINDING state.
[d] One mandatory *Option* is the "Message Type" but there are a lot more.

DHCP type

Most of the fields listed above are in the DHCP type, the complete type can be found in Appendix A.5, Listing 31, part is shown in Listing 14. The type has three constructors, only two are shown (line 2 and 6). *Panding hostname* and *host* are not included in the type because they will always be zero. In the receiver the bytes on this position are ignored, the sender will add the missing bytes at the

correct position. *DHCPBody* (line 1 to 5) is used to choose the correct constructor depending on the IP version. The *sourceDHCP* field (line 3 and 8) holds the source port from the UDP header, the source port is needed to test if the last received message is indeed DHCP. This check is needed because the message is stored in the FPGA blockram, where the information stored in the EthernetType is not present. The previous will be discussed in more detail in section 4.2.7. The *lengthDHCP* (line 4 and 12) is the UDP length needed for the state machine to know the maximum length of the DHCP message. Most fields for the *DHCPBodyV4* constructor are in the list above, except for *serverAddress* (line 20). This is a separate type storing all four IP addresses (line 23 to 27). For IPv6 this type can be reused but with a length of 16 (for the IP-addresses being 128 bit). This way the type is generic and can be used for both IP versions, although DHCP is rarely used for IPv6.
The last record for the *DHCPBodyV4* constructor is the *optionsDHCP* (line 21), which is similar to the IPv6 extension vector shown in section 4.2.3. However, parsing the options is done in another way.

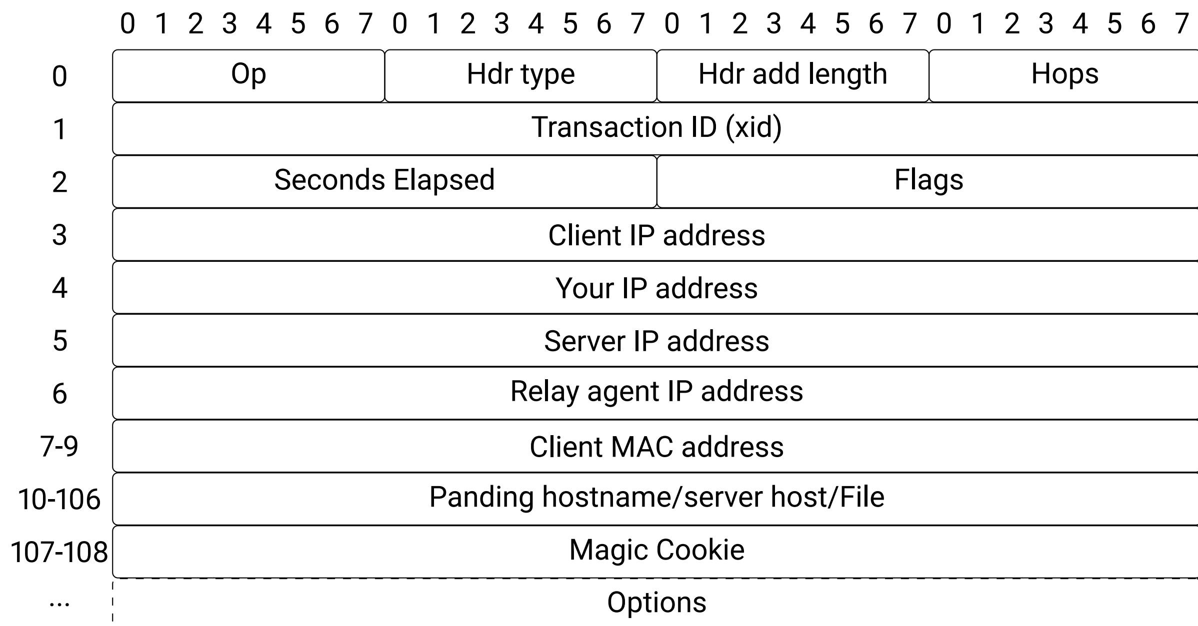

Figure 4.26: A DHCP frame as presented in the RFC.

```
data DHCPBody n=
  DHCPBody{    --a body
    sourceDHCP           :: EthernetWord,
    lengthDHCP           :: EthernetWord,
    versionDHCP          :: EthernetWord}
```

```
6     | DHCPBodyV4 {  --a IpV4 body
7       messageTypeDHCP       :: Byte,
8       sourceDHCP            :: EthernetWord,
9       typeDHCP              :: Byte,
10      hardwareTypeDHCP      :: Byte,
11      hardwareAddressDHCP   :: Byte,
12      lengthDHCP            :: EthernetWord,
13      versionDHCP           :: EthernetWord,
14      hopsDHCP              :: Byte,
15      transactionIDDHCP     :: Vec 4 Byte,
16      elapsTimeDHCP         :: Vec 2 Byte,
17      flagsDHCP             :: Vec 2 Byte,
18      cookieDHCP            :: Vec 4 Byte,
19      gatewayAddress        :: Vec 6 Byte,
20      serverAddress         :: (DHCPServer 4),
21      optionsDHCP           :: Vec n DHCPOptions}
22    ..
23
24  data DHCPServer n = DHCPServer {
25    clientIP    :: Vec n Byte,
26    yourIP      :: Vec n Byte,
27    serverIP    :: Vec n Byte,
28    gatewayIP   :: Vec n Byte}
```

Listing 14: DHCP type in Clash, the number of supported options is variable and chosen at compile-time.

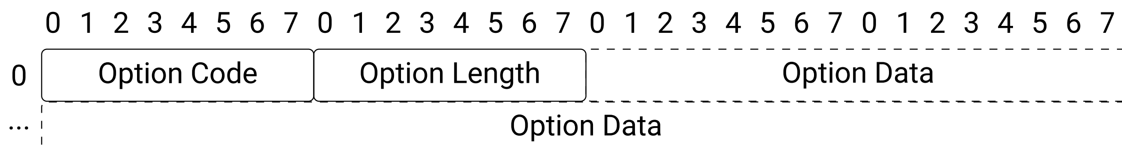

Figure 4.27: A common DHCP option as presented in the RFC.

DHCP options

In Figure 4.27 a possible DHCP *Option* is drawn, this structure looks a lot like the structure presented in the section about IPv6 extension headers (Section 4.2.3). The maximum length calculation DHCP differs from the extension headers, the maximum is one byte. This leads to an upper limit of 255 bytes as the maximum option size. The options can have different lengths and can be calculated by $2 + (n)$, where n has to fit in a byte.
The difference between DHCP options and IPv6 extension headers are the amount of options they contain. The number of DHCP options is limited by the requested number. The server will only reply to the options in this list, but only to the options it supports. Where in IPv6 the client sends all extension headers needed for the data without knowing if the server can handle this extension header. When the server can not handle the extension header it will reply with an error message containing the header number and the client will make a new message without the rejected header.

The DHCP server will only reply to a requested *Option* when the option is supported. There can never be more options in the reply from the server than requested by the client. From this we can conclude a client should never request more options than the reserved space. Combining all this information the total maximum length to reserve for DHCP options can be calculated at compile time by $requestedOptions * (2 + 255)$. Most of the options will have a smaller length. For example option 23 has only 1 byte of data. In an IPv6 extension header the data has to be aligned for fitting in a 32 bit word, in DHCP this is not a requirement. The server should try to keep the same order of options in the reply as was requested by the client, however this is not mandatory. As stated in the RFC: "The DHCP server is not required to return the options in the requested order, but MUST try to insert the requested options in the order requested by the client." [39]. Only the *Magic_Cookie* has a fixed position in the option list.

DHCP option type

The system can handle 16 different options and a wildcard option, four options are shown in Listing 15. The complete type can be found in Appendix A.6, Listing 32, Listing 33 has the complete annotation. In Listing 15 the *DHCPOption* type (line 1 to 3) has two records, one for the length and one for the data. The option code is embedded in the custom bit encoding for the different options. The *DHCPOptions* (line 5 to 10) specify the option and the length. Most options make use of a fixed length (line 7 and 9), for other options the length can be variable and is set to the maximum given by the user: *DHCPMaxOptionLength DHCPOptionLength* (line 6 and 10). The calculation for the length can be found in Formula 4.2.

$$DHCPMaxOptionLengthn = (n * 2) + 8 \tag{4.2}$$

The offset in the formula is 8 because the longest fixed length option is 7, but in some variable options IP-addresses are send, so a multiple of 4 is taken as a base. The calculation of $n * 2$ in the formula is introduced to keep the maximum options always even because data coming from the blockram is a 16-bit word. A possible stream of options is shown in Figure 4.28, there are three options in the stream translated to a Clash type.

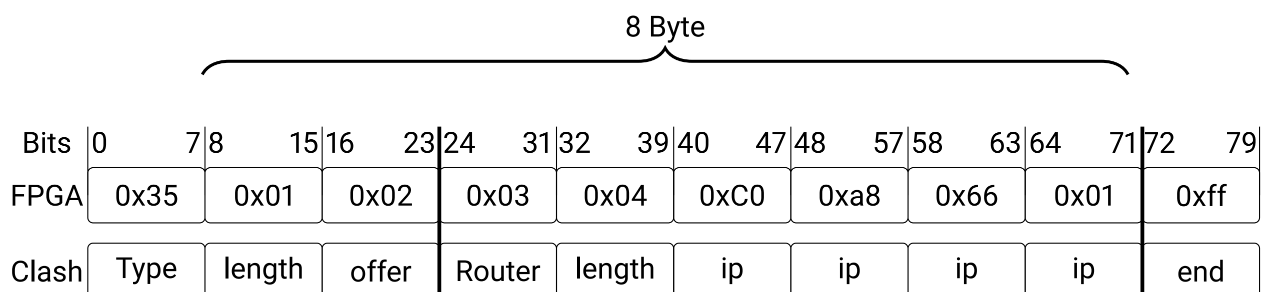

Figure 4.28: Stream of DHCP options on a FPGA, stored and encoded to a Clash type.

```
1  data DHCPOption n = DHCPOption {
2    optionLength  :: Byte,
3    optionValue   :: Vec n Byte}
4
5  data DHCPOptions
6    = DHCPOptionRouter  (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
7    | DHCPOptionMessageType  (DHCPOption 1)
8    ..
9    | DHCPOptionEndmark  (DHCPOption 0)
10   | DHCPOptionWildcard  (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
11
12     ANN module (
13   let maxOptionLength = DHCPMaxOptionLength DHCPOptionLength
14       maxLen = maxOptionLength*8
15       posLen = (shiftL 0xff (maxLen))
16   in DataReprAnn
17     (maxLen+16)
18     [ConstrRepr 'DHCPOptionRouter (shiftL 0xff (maxLen+8)) (shiftL 0x03 (maxLen+8))
19       [(2^(8*maxOptionLength+8)-1) ],
20     ConstrRepr 'DHCPOptionMessageType (shiftL 0xff (maxLen+8)) (shiftL 0x35 (maxLen+8))
21       [posLen + (2^(8*1)-1) ],
22     ..
23     ConstrRepr 'DHCPOptionEndmark  (shiftL 0xff (maxLen+8)) (shiftL 0xff (maxLen+8))
24       [posLen + (2^(8*0)-1) ],
25     ConstrRepr 'DHCPOptionWildcard (shiftL 0x00 (maxLen+8)) (0)
26       [2^(maxLen+8)-1]
27   ..
```

Listing 15: DHCP option type including the annotation.

In the custom encoding (line 13 to 27) the same mask calculation is used of all four options (line 18, 20, 23 and 25). For the mask calculation the option length is taken, this means the data is taken from the right most site of the option (the MSB). The *maxOptionLength* is defined on compile-time using Formula 4.2 (line 13 of listing 15). The *maxLen* is the length of the option data in bits (line 14). All unknown options are handled by the wildcard, this should not be needed for DHCP because only the requested options will be returned by the server.
The first option in a DHCP message should be the DHCP Message type (shown on line 7, 20 and 21 of Listing 15). In the bit-stream shown in the FPGA in Figure 4.28 the first option is $0x35$, in Listing 15 Clash will match the encoding on line 20 and 21 to the first byte on the FPGA. From Line 20 Clash will match the $0x35$ to line 7, the *DHCPOptionMessageType*. The length of the *DHCPOptionMessageType* is set to be 1. The length of 1 matches byte 2 of Figure 4.28 where the length is $0x01$. The next byte is the data, according to the RFC $0x02$ in the message type revers to a DHCP-offer. The next part of the stream, from bit 24, will be a new option with the number $0x03$. Clash will match this option to line 18 of Listing 15, the router option. The length of the router option is variable (line 6), so the length in the option will tell the number of bits. In line 19 of Listing 15 the length is set to the *maxOptionLength*. Figure 4.28 bit 32 to 40 hold the length of the router option, $0x04$, so 4 bytes of data. The data will

be in bit 40 to 72, so in the example $0xC0$, $0xa8$, $0x66$ and $0x01$. Clash will store the information in *DHCPOptions* using the constructor *DHCPOptionRouter*. The last byte in the FPGA bit-stream of Figure 4.28 is $0xff$, the Endmark. Line 23 of Listing 15 matches $0xff$ and will take 0 bits for the length and 0 bits for the data, the result in *DHCPOptions* is an empty constructor, *DHCPOptionEndmark* with no data and length. The *DHCPOptionEndmark* can be followed by a number of zero options, also called pad option. A zero options option is a 0-byte without a length or data field and will not be parsed by the system.
In IPv6 extensions there is alignment between the extensions, all will have a length which is a multiple of 32-bits. For DHCP options this is not the case. Take for example *DHCPOptionMessageType* (line 7), the fixed length is one, adding the option field en length field will result in a size of 24-bits. However, in the UDP checksum calculation words of 16-bits are expected. So when all options together do not end on a multiple of 16-bits the pad option will be used. The state-machine in the system will also receive data in 16-bits words from the memory because for the checksum calculation the data is stored in 16-bit words.

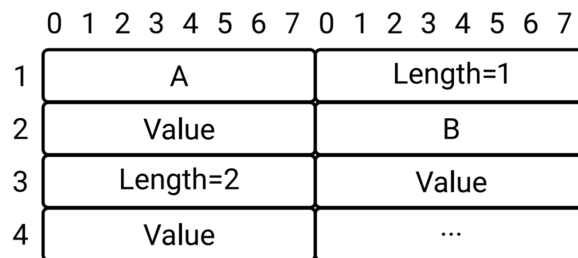

Figure 4.29: An odd UDP Option stored in blockram.

Odd and even

The solution to deal with the odd length is best explained by using an example. Suppose there are two options, A has a length of 1 and B is 2 bytes. The fields are read from memory as 16 bit numbers as depicted in Figure 4.29. The first line is parsed in the first cycle and gives the result $A, 1$. The second read will give $value, B$. This value belongs to option A, but code B represents the start of a new option. The parser has to put the complete options to memory and check whether option B is the *End_Option* tag. This is not the case and a next read will give $2, value$, and so on. In this example the start position of an option in the data stream has to be tracked by the parser. The parser will have a state variable to remember where the *Option_Code* and length are located in the word. This state variable will handle the input word as being odd or even, depending on the previous messages.

DHCP protocol

The DHCP protocol has four states: Discovery, Offer, Request and Acknowledge. The client will try to get an IP-address from a DHCP-server, this is illustrated in Figure 4.30. The client will first send a Discovery to the server. In the Discovery frame all options supported by the client are requested. All servers will send an Offer, to simplify the Figure only one server is drawn. The client will take one of the

offers and reacts using a Request to tell the server which IP-address he will be using in the network. This request needs to be acknowledged by the server. DHCP is one of the most basic but functional protocols to test, it is simple and easy to test because nearly every network has a DHCP server. To check whether the correct IP-address is used by a device, it has to send a frame. In the header this new IP-address should be used.
On the FPGA an user will give a list of option to request, the list will be should not be longer than the maximum amount of options. As described above options can have a variable length so could be longer than the maximum specified by the used. In this case a option will take multiple positions in the vector containing the options.

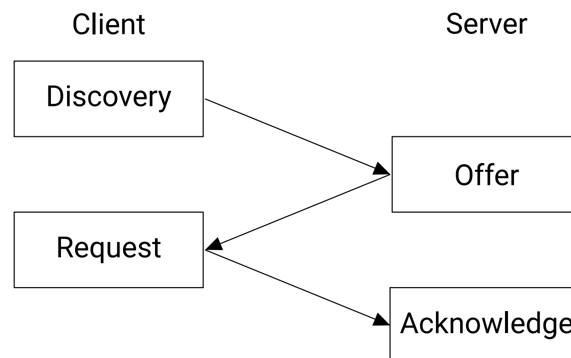

Figure 4.30: The DHCP state-machine, to get a IP address the client will send a Discovery to the server.

Long options

The behavior described above is depicted in Figure 4.31. The user has to give a list of options supported by the system (at compile time): as seen at the top block in the figure. Before making a request all duplications will be filtered and when the list has more entries than the option data type, the remaining part will not be used; the list is truncated. This is done to make sure that the parameter list will not exceed the maximum storage capacity. In order to have some extra space available for the *End_Option*, the truncate block will make the list of the request one option smaller compared to the available storage capacity. In the third block, named Request, the message is send to the server and a reply will be sent back. The parser will read all options and will check and keep track if they are too long. When the *Option* is twice the maximum length it will not be parsed at all to save space (left side of the *Option_length* block in the figure). When there is still space but the *Option* is too long it will get two slots in the options storage. The first entry of this *Option* will hold the total *Option_Length*, the second will hold the remaining length. When the system is in the ’Fill option’ state and there is no storage left all options using two slots will be put on the "too long list". When there are options in this list and the parsing is finished a new request will be made but the options on this "too long list" will not end up in the new request (the ’Delete long option’) state in Figure 4.31. This method will result in a list containing all options that can be used in combination with the available storage.
Not all options will have the same alignment and can have a odd length. In IPv6 this is solved using the zero options but in DHCP this option does not exist, so the system should handle this.

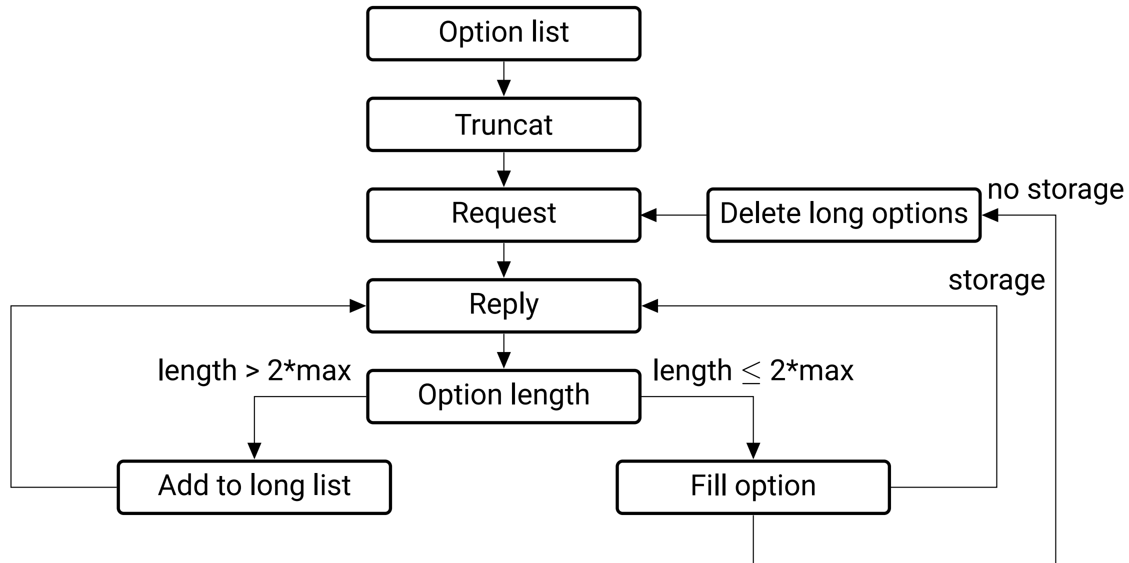

Figure 4.31: State machine for handling long DHCP option and will make sure the maximum storage capacity is exceeded.

4.2.6 Checksum calculation

The calculation for the IPv4 and UDP checksum are the same. To calculate the checksum all fields needed for the calculation are placed in a vector of 16-bit numbers. All numbers in the vector will be add together as shown in Listing 16 line 1 to 3. The outcome should be a 16-bit number, however when adding two 16-bit numbers the outcome will be a 17-bit number. To truncate the number back to a 16-bit number the MSB must be added to the 16-bit number. In Listing 16 line 5 to 12 the *addCarry* function is shown. The function will add the two 16-bit numbers (line 11) resulting in the *summed* result of 17-bits. The MSB will be taken from this result (line 12) and will be add to the truncated result (line 9). The outcome will be a 16-bit number. The checksum calculation is done when all positions in the vector are added together (line 3). One important part of the calculation is the *foldl1* (line 3), this will force the calculation of to be done in one clock cycle, this could be a problem in the performance.

```
1  addVec :: Vec (n+1) (EthernetWord)
2  -> EthernetWord
3  addVec xs = foldl1 addCarry xs
4
5  addCarry :: KnownNat n
6  => BitVector n
7  -> BitVector n
8  -> BitVector n
9  addCarry a b = truncateB summed + resize (bitCoerce carry)
10 where
11 summed = a `add` b
12 carry = msb summed
```

Listing 16: The checkum calculation will add two sixteen bit words and give a sixteen bit word as a result.

A solution would be to calculate the checksum when parts of the data are received. Parts of the checksum calculation should in that case be done in the Network layer, however this will give extra design problems when packing the IP-headers. Timing problems will most likely arise for IPv6 frames because of the size of the pseudo header being 192-bits longer compared to IPv4.
The IPv4 checksum calculation will be done when all header fields are received. For UDP the calculation is split in two parts, when all fields of the header are received the calculation for the first part will be done. The second part will be done when the data is written to memory.

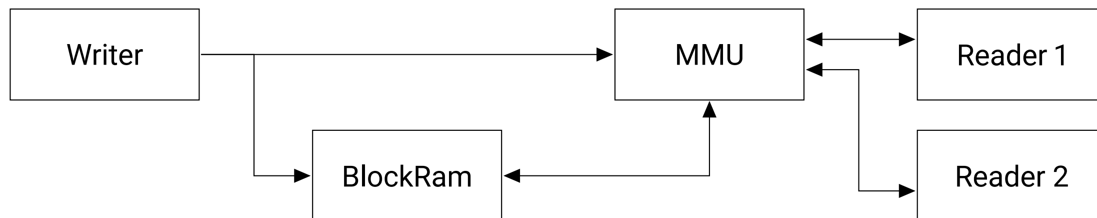

Figure 4.32: Block diagram of a Memory Management Unit including two readers.

4.2.7 Memory Management Unit

To connect the Session layer to the receiving system a MMU will be used. This MMU will handle the communication between multiple readers and the blockram. Suppose we have a system containing one unit writing to memory and two units trying to read at the same time. This is not possible for a normal blockram, and therefore a MMU will be used. The general structure can be seen in Figure 4.32. Left in the Figure is the writer writing data to the blockram. This same information will be written to the MMU in order to track the state of the blockram. When a reader wants to receive data from the blockram a request will be sent to the MMU. When there is data available the request will be forwarded to the blockram. When there are two readers requesting data at the same point in time the MMU will decide to forward the request from the reader to the blockram coming from the reader that did not read last time, also known as round robin. The other reader will get data in the next clock cycle. This will prevent one reader from getting all reading time but distributes data over the readers.

Example of using the MMU

Let us take the situation as described above. First introduce the principle and then discuss the internal states of the MMU. The principle is shown in the diagram in Figure 4.33. Both readers are sending requests to the MMU and there is data available for both of them. In this example the two readers will get the Maybe data type (see Section 2.5.1) *Just data* or *Nothing* depending on who did get the data during the preceding cycle. To simplify the figure *Just data* and *Just request* are data and request respectively. In the first clock cycle reader 1 does get data, the other one has to wait and gets Nothing. In the second cycle reader 2 will get data because the MMU knows that reader 1 did get data in the previous cycle. In the third cycle this is flipped again and this continues till both readers have received all data.

Reader 1 to MMU	request	request	request	request	
MMU to Reader 1	data	Nothing	data	Nothing	
Reader 2 to MMU	request	request	request	request	
MMU to Reader 2	Nothing	data	Nothing	data	
	1	2	3	4	

Figure 4.33: Timeline multiple readers with a Memory Management Unit constantly requesting data from the blockram.

Position

Apart from the fair use of the blockram managed by the MMU, the MMU has a second function: keeping track of the position read by every reader. Using this information a read request is only forwarded when there is data available. Let us expand the example above with this position information. In Figure 4.34 this diagram is drawn, both readers have added a 'position/available' row above the request line. In the example this 'position' is used to point at the location last read by the reader, 'available' holds the maximum position of where data is stored in the blockram. To simplify the figure *Just data* and *Just request* are data and request respectively. As shown in the diagram, reader 1 does tries to read data but it is not available yet. When the fair use system has to handle this the readers will have alternating turns. Using the position information and combining it with the fair use system this will result in reader 2 getting data in cycle 1 and 2. In cycle 3 there is new data available so the fair use system will allocate the timeslot to reader 1. Reading data from the blockram in this way is more efficient and the readers will get the data when it is available.

position/available	2/2	2/2	3/3	3/3	
Reader 1 to MMU	request	request	request	request	
MMU to Reader 1	Nothing	Nothing	data	Nothing	
position/available	0/2	1/2	1/3	2/3	
Reader 2 to MMU	request	request	request	request	
MMU to Reader 2	data	data	Nothing	data	
	1	2	3	4	

Figure 4.34: Timeline with multiple readers constantly requesting data from blockram, MMU will block a read when all data has been received.

Most readers will not always request data; they will have to process the data before reading the next line. This is shown in Figure 4.35. The 'position/available' field is removed to simplify and we will assume there is data available for both readers. To simplify the figure *Just data* and *Just request* are data and request respectively. Reader 2 requests data all the time because it does not need time to process the data. Reader 1 on the other hand needs time to process two cycles for one line of data. In the first cycle reader 1 will get data, in the next two cycles it will not request any data. Reader 2 will get data at this point of time and will continue requesting more. In the fourth cycle reader 1 will request new data and will get it immediately because reader 2 did read last time, so his request will have a lower priority compared to reader 1. The same will happen in cycle 7 when reader 1 requests data again.

Reader 1 to MMU	request	Nothing	Nothing	request	Nothing	Nothing	request
MMU to Reader 1	data	Nothing	Nothing	data	Nothing	Nothing	data
Reader 2 to MMU	request	request	request	request	request	request	request
MMU to Reader 2	Nothing	data	data	Nothing	data	data	Nothing
	1	2	3	4	5	6	7

Figure 4.35: Timeline of two readers where one needs two cycles to process one line of data. The MMU will give this read a higher priority.

The examples described above can also be used for more than two readers. The number of states will get larger and the number of readers that have to wait for data will increase. But in the end they will get the data.
In Listing 17 a small part of the code is shown. Take the example where only one reader is requesting data: in the code this is Reader 1 (line 2). This is the only reader requesting data, the address is not important, so there is only a match of *Just* _ (line 2, second position of the second tuple). The *mmuReadSystemReceiver* will send the requested address to the *mmuRead* (line 5). The *mmuRead* will check if there is data at the given address and will forward the request to the *BlockRam*. The reply will be send back to the *mmuReadSystemReceiver* and will be send to the correct reader (line 3). The *mmuReadSystemReceiver* will update the state (first position of the tuple on line 3), so Reader 1 will not receive data when two readers are requesting at the same time.
Now assume Reader 2 and 3 are requesting data (line 7 of Listing 17). The *mmuReadSystemReceiver* will check if Reader 2 did request data the last time (line 9). When this is the case Reader 3 will get data, but only when there is data (valid3 has to be true in the guard line 9). Otherwise Reader 2 will get the data (line 10). When none of the two readers or Reader 3 received data last time (line 11) Reader 2 will get data. Once again, there should be data available for this reader, otherwise Reader 3 will get the data (line 12). This function will follow the same procedure for all possible combinations of zero, one, two and three readers requesting data at the same time.

```
1  ..
2  mmuReadSystemReceiver(_,s1,s2,s3,memReady) (memState,m1@(Just _),Nothing,Nothing)
3    = ((1,s01,s2,s3,(valid1,False,False)),(m1Out,memReady))
4    where
5    (s01,(m1Out,valid1)) = mmuRead s1 (m1,memState)
6  ..
7  mmuReadSystemReceiver(x,s1,s2,s3,memReady) (memState,(Nothing),m2@(Just _),m3@(Just _))
8    = case x of
9    2 | valid3==True  -> ((3,s1,s2,s03,(False,False,valid3)),(m3Out,memReady))
10     | otherwise     -> ((2,s1,s02,s3,(False,valid2,False)),(m2Out,memReady))
11   _ | valid2==True  -> ((2,s1,s02,s3,(False,valid2,False)),(m2Out,memReady))
12     | otherwise     -> ((3,s1,s2,s03,(False,False,valid3)),(m3Out,memReady))
13   where
14   (s02,(m2Out,valid2)) = mmuRead s2 (m2,memState)
15   (s03,(m3Out,valid3)) = mmuRead s3 (m3,memState)
16 ..
```

Listing 17: The implementation for the MMU Reader.

4.3 Implementation of the transmitter

The transmitter uses the same data types as the receiver, including the state variables. The transmitter can have two states, at startup it will be in the DHCP state. In this state the system will try to request an IP address. When the system has an IP address it will send the messages stored in the MMU.

DHCP

For the initial state a DHCP-discover message is stored in the FPGA. Clash will use a provided list of options to send the discover. Before sending the discover Clash has to calculate the length and UDP checksum. The discover will be send to the MMU by the DHCP sender. The UDP-sender will get the message back from the MMU. UDP will add the port numbers and will calculate the rest of the checksum.
When the UDP-sender added all necessary information the IPv4 or IPv6 sender will add the IP addresses in the case there is an IP address known.
Adding the IP addresses is done using the identifier, the system has a list of identifiers with corresponding IP addresses. When an identifier is not in the list and the system can not give a destination IP address, the default address is used. In case of IPv4 255.255.255.255 is used when starting the system, otherwise the DNS address obtained from DHCP will be used. IPv6 will use the predefined DNS address. The source IP address is set to the system address except for DHCP, in this case there is no source address so the default (0.0.0.0) will be used.
When all IP addresses are set the checksum can be calculated for IPv4 and for UDP. In case of IPv4 the checksum for both protocols have to be updated. For IPv6 only the pseudo header presented in section 4.2.4 will be added to the UDP checksum.
Now the IP header contains all required information and the message will be parsed to the header-sender. This header-sender will add the MAC-addresses, this will work in a similar way as adding the IP addresses. The difference for the MAC is that the IP-addresses are compared with a list of combinations of MAC and IP. When there is no MAC address, the system will fill the destination with 255 at all 6 positions. For every layer at this point the header is filled. There is no data send jet.

Sending a frame

The sender will send the data byte by byte, starting with the header. The first 12 byte can always be send right away, these are just the MAC-addresses. For the next part there are three possibility to send, the etherType, VLAN tag or double VLAN tag. In the last 2 cases the sender will jump back four positions, in the case of double tagging this jump will be made twice. The double tag or VLAN tag will be removed from the EthernetType, a small part of the code is shown in Listing 18. First the VLAN tag will be separated from the *dataIn* (line 5 and 6), now the *dataIn* is 4 bytes to short, the system will replace the gap by 4 undefined bytes and pack it back in the EthernetType (line 7 and 8). The custom encoding shown in section 4.2.2 Listing 3 will make this possible. Take for example *VlanV4*, and feed it to the code in Listing 18 (line 5 and 6) the result from de encoding will be something $(0X8100xxxx, 0X0800...)$. When the second part of the tuple is pack using the same encoding it will

```
1 sendingVLanFrame ::(SendingEthernetFrame, EthernetFrame, ..
2 sendingVLanFrame (_,(EthernetFrame header ethernetFrame), ..)
3   = (SendingFrame (11,0),(EthernetFrame header eType) ..
4     where
5     (tag,dataIn) = unpack (pack ethernetFrame :: (BitVector LengthFrame))
6       :: (Vec 4 Byte,Vec (LengthFrame-32) (Bit))
7     eType = unpack (pack (dataIn++ (replicate d32 undefined))::
8       (BitVector LengthFrame) ) :: EthernetType
```

Listing 18: The implementation for sending the VLAN tag.

result in *IPv4*. The outcome from pack can be put back in the frame while the header is kept untouched (line 2 and 3).

Chapter 5

Testing

5.1 Introduction

In this chapter the results of the simulations are discussed. Unfortunately there was no time left to test the implementation on an FPGA. All Clash code compiles on version 1.3.0 at the moment of writing the latest Clash available.

5.2 IPv6 extensions

One important part of the implementation are the IPv6 extensions. There are two scenarios for sending an extension, one where a correct header is received and one where a for the system unknown header is received.

Correct Hop by hop

One of the extensions supported by the system is a HopByHop extension. In Figure 4.18 of section 4.2.3 a HopByHop extension was taken as an example, where there was more space reserved than needed by the extension. In this section the same experiment is repeated, this time using real data. In a program called Wireshark a caption of a HopByHop extension is made. The full caption is shown in Listing 34 in Appendix B.1. Part of the extension is shown in Figure 5.1, in the caption three things are visible: the next header is UDP (17, decimal), the length is 1 and there is some data. Using the length calculation given in formula 4.1 presented in section 4.2.3 the total length is $8 * 1 + 8 = 16$. On line 4 of Figure 5.1 *[Length: 16 bytes]* is shown, this is the same result as formula 4.1 gave. The extension contains data, starting with 0x01 (hexadecimal), followed by the decimal number 12 and some PadN containing hexadecimal.
The output given by the Clash simulation is shown in Listing 19, there is a *HopByHopIPv6Hex* found on line 2. There is *Hex* in every data field to get the output printed in hexadecimal values. The same three parts from the caption (Figure 5.1) are visible in the output of Listing 19 as well. The next protocol is 0x11 (line 4), revering to UDP ($0X11(hexadecimal) = 17(decimal)$). The length is 0x01 and there is data. The data (line 6 to 8) is 22 bytes, starting with 8 zeros. Comparing the 22 bytes of data to the 14 bytes of data shown in the caption of Figure 4.2.3 gives a difference of 8 bytes. The first 8 zeros are

introduced because $NumberIPv6Extension = 2$, using Formula 4.1 presented in section 4.2.3 with the number 2 results in $(2*8)+6 = 22$. However, the extension has a length of only 16 bytes meaning there should be 8 undefined bytes. When the first 8 zero bytes (line 6 of Listing 20) are considered undefined the result of Listing 20 matches the data as shown in Figure 5.1. The actual data starts with 0x01 (Figure 5.1 line 6), which is the same in Listing 19 line 7. The other 13 bytes are present and correct as well, however the length is shown in a decimal number in Figure 5.1.

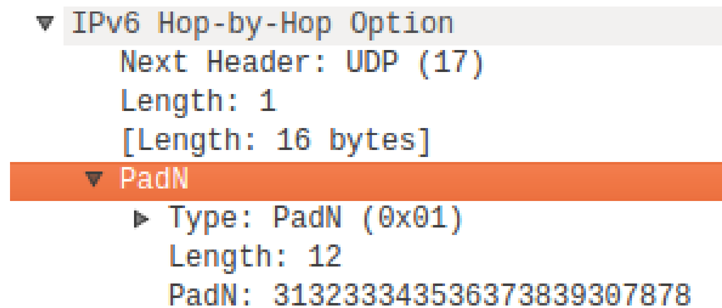

Figure 5.1: HopByHop extension displayed via a wireshark caption.

```
1 >> extensionHeadersIPv6Hex =
2     <HopByHopIPv6Hex
3      ExtensionHeaderIPv6Hex {
4       nextExtensionHeaderIPv6Hex = 0x11,
5       headerLengthIPv6Hex = 0x01,
6       headerPayloadIPv6Hex  = <0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
7        0x01,0x0c,0x31,0x32,0x33,0x34,0x35,0x36,
8        0x37,0x38,0x39,0x30,0x78,0x78>}>
```

Listing 19: Output when a known HopByHop extension header is parsed by Clash.

Incorrect HopByHop

In the previous section a correct extension was parsed by Clash. But to investigate what would happen when the extension is unknown, the HopByHop extension is disabled in Clash. In this test the same HopByHop message is fed to the system, however the system will not recognize the type. Listing 20 (line 2) shows the *WildcardIPv6Hex*, as expected because Clash can not find an entry in the encoding. All other fields in Listing 20 are equal to Listing 19, the data is represented by two dots. The Next header is identified correctly (Listing 20 line 3), so the parser can still use de rest of the frame. In both examples the next extension is $0x11$, a upper layer protocol, however this could be another extension as well. The system should recognize the so called chained extensions and use the information in the next extension field to encode the next extension.

```
>> ..
  <WildcardIPv6Hex {
    nextExtensionHeaderIPv6Hex = 0x11,
    headerLengthIPv6Hex = 0x01,
    headerPayloadIPv6Hex  = <0x00, ..

```

Listing 20: Output when an unknown HopByHop extension header is parsed by Clash.

5.2.1 Chained extensions

In a chained extension the next extension field will tell what the next extension will be, so the system needs this information from the previous extension to encode the current one. Figure 5.2 shows a caption from Wireshark with two chained extensions. The full caption is shown in Listing 35 in Appendix B.2. In Figure 5.2 the first is the HopByHop extension, the same as in the previous example. There is one main difference between 5.2 and the previous caption in Figure 5.1, the next header is changed to 44 (or $0x2C$). In simulation the first 8 lines in Figure 5.2 should give the same result as the previous example.
Line 9 shows the Fragmentation Header, the next header is 17 ($0x11$) and is an UDP frame. The length is reserved to $0x00$, followed by an offset (line 12 up to 14). The offset is zero except for the last bit. The last line of Listing 21 are five zeros followed by three nines (hexadecimal). In section 4.2.3 Listing 12 the length of a Fragmentation header was set to 6 (line 8), so Clash will not show any undefined bytes for the Fragmentation header.

```
▾ IPv6 Hop-by-Hop Option
    Next Header: Fragment Header for IPv6 (44)
    Length: 1
    [Length: 16 bytes]
  ▾ PadN
    ▸ Type: PadN (0x01)
      Length: 12
      PadN: 313233343536373839307878
▾ Fragment Header for IPv6
    Next header: UDP (17)
    Reserved octet: 0x00
    0000 0000 0000 0... = Offset: 0 (0 bytes)
    .... .... .... .00. = Reserved bits: 0
    .... .... .... ...1 = More Fragments: Yes
    Identification: 0x00000999
```

Figure 5.2: HopByHop followed by a Fragmentation extension displayed via a wireshark caption.

The chained extensions are interpreted by Clash, the result is shown in Listing 21. As expected first the HopByHop extension is shown on line 2 till 9, the result is equal to Listing 19 only line 4 has the value $0x2C$ where this was $0x11$ in Listing 19. Listing 21 line 9 is the constructor of the Fragmentation header, line 10 is the next extension header $0x11$, an UDP frame. The next line (line 11) is the length of $0x00$, the same as the caption. Line 12 has the payload, the first byte is $0x00$ and the second $0x01$, this is the offset in the caption of Figure 5.2. The other bytes in Listing 21 are zero, except the last two

which are $0x09$ and $0x99$ respectively, this in the Identification in the caption of Figure 5.2.
The test above shows that two chained extensions can be handled by the system, the test will work with a unknown extension as well. When the HopByHop was an unknown extension like the example in section 5.2 the first extension in Listing 21 would look like Listing 20, the output for the Fragmentation header would still be the same.

```
>> extensionHeadersIPv6Hex =
<HopByHopIPv6Hex
ExtensionHeaderIPv6Hex {
nextExtensionHeaderIPv6Hex = 0x2C,
headerLengthIPv6Hex = 0x01,
headerPayloadIPv6Hex  = <0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x01,0x0c,0x31,0x32,0x33,0x34,0x35,0x36,
0x37,0x38,0x39,0x30,0x78,0x78>}
,FragmentIPv6Hex ExtensionHeaderIPv6Hex {
nextExtensionHeaderIPv6Hex = 0x11,
headerLengthIPv6Hex = 0x0,
headerPayloadIPv6Hex = <0x0,0x1,0x0,0x0,0x9,0x99>}>
```

Listing 21: Output when a known HopByHop and Fragmentation extension headers are parsed by Clash.

5.3 DHCP

For receiving DHCP-options a caption from Wireshark is used, only the part containing the options send in the offer by the DHCP-server is shown in Figure 5.3. There are seven options, all have a length followed by data, except for the last (option 255).

```
▾ Option: (53) DHCP Message Type (Offer)
    Length: 1
    DHCP: Offer (2)
▾ Option: (54) DHCP Server Identifier (192.168.102.1)
    Length: 4
    DHCP Server Identifier: 192.168.102.1
▾ Option: (51) IP Address Lease Time
    Length: 4
    IP Address Lease Time: (86400s) 1 day
▾ Option: (1) Subnet Mask (255.255.255.0)
    Length: 4
    Subnet Mask: 255.255.255.0
▾ Option: (3) Router
    Length: 4
    Router: 192.168.102.1
▾ Option: (6) Domain Name Server
    Length: 8
    Domain Name Server: 8.8.8.8
    Domain Name Server: 8.8.4.4
▾ Option: (255) End
    Option End: 255
```

Figure 5.3: DHCP options displayed in wireshark, only the options are shown.

The full caption is shown in Listing 36 in Appendix B.3. Feeding this as an input into the system and print it using decimals will result in the output shown in Listing 22. In the output the last four

options are the same as in Figure 5.3, starting with option 1 (Subnet Mask) (line 2). The length is 4 (as specified in RFC 2132 [39]). The output given by Clash is identical to the information from Wireshark. At line 5 the constructor for the router options (option 3) is shown. The length is 4 (line 6) and there is data. The 28 bytes on line 7 and 8 seem to be some random bytes, the last 4 bytes belong to the option. The four bytes hold the values: $192, 168, 102, 1$, identical to option 3 (line 16) of Figure 5.3. The rest of the data should be considered undefined because the option has a length of only four bytes. However, the random values are not completely random. Take for example the first three bytes of line 8, $192, 168, 102$, these are equal to the first 3 bytes of data from the *Server Identifier* (line 5 of Figure 5.3). The next three bytes on line 8 of Listing 22 are the first three bytes of data from option 51 (IP Address Lease Time) and so on. The data was still in the 28 byte buffer used for DHCP.
For option 6 (line 9 up to 12 of Listing 22) (Domain Name Server) the same holds as for the last option, all information of the last options are visible (line 11 and 12), meaning only the last 8 bytes are relevant. The data is again equal to the number shown in Figure 5.3.
For the final option, number 255 (End), the option value is empty (line 15), because it always has a length of 0 and the length field is not present as discussed in section 4.2.5, the length is set to zero (line 14).
The above shows Clash is able to receive a DHCP message and recognize the different options. The system can handle options with a variable length and will use the length given by the option to interpret the data.

```
1  >> ..
2    DHCPOptionSubnetMaskHex DHCPOptionHex
3      {optionLengthHex = 4,
4      optionValueHex = <255,255,255,0>},
5    DHCPOptionRouterHex DHCPOptionHex
6      {optionLengthHex = 4,
7      optionValueHex = <0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
8      192,168,102,0,1,81,255,255,255,192,168,102,1>},
9    DHCPOptionDomainNameServerHex DHCPOptionHex
10     {optionLengthHex = 8,
11     optionValueHex = <0,0,0,0,0,0,0,0,192,168,102,0,1,81,
12     255,255,255,192,168,102,8,8,8,8,8,8,4,4>},
13   DHCPOptionEndmarkHex DHCPOptionHex
14     {optionLengthHex = 0,
15     optionValueHex = <>}>})
```

Listing 22: Output when a DHCP option is parsed by Clash.

5.4 Sending VLAN

When a frame is send the VLAN is one of the special cases where the system makes a jump back in de code to send some parts again. The jump is presented in section 4.3. When the code from the section is tested in simulation the jump is visible in the output. For this test an input is made, as shown in Listing 23, line 1 shows the types of the input. The first type is a state called *SendingEthernetFrame*

```
(SendingEthernetFrame, EthernetFrame ,Maybe Byte, ..)
(SendingFrame (12,0), (EthernetFrame (EthernetHeader .. )
  (VlanDoV4 undefined undefined undefined undefined)), Nothing, ..)
```

Listing 23: Test input for Double VLAN test

```
(SendingFrame (13,0), (EthernetFrame (EthernetHeader .. ) (VlanDoV4 .. )),Just 0x88, ..)
(SendingFrame (14,0), (EthernetFrame (EthernetHeader .. ) (VlanDoV4 .. )),Just 0xA8, ..)
(SendingFrame (11,0), (EthernetFrame (EthernetHeader .. ) (VlanV4   .. )),Just 0x**, ..)
```

Listing 24: Test Output for Double VLAN test

(line 1) followed by an *EthernetFrame* (line 1). Line two shows the data fed into the system, the state will start at 12 and the EthernetFrame is a VlanDoV4 (line 3), meaning a double Vlan tag, all other fields are undefined.
Section 4.3 mentioned that the first 12 bytes are the MAC addresses, in this example being position 0 to 11 in the state. After sending the MAC-address the EtherType is send. The state in this example starts at 12, so the first byte of the EtherType should be outputted by the system.
Listing 24 shows the output, every line is one clock tick. Every line shows the next state of the system. Line 1 is state 13, and the data send is $0x88$, the first byte of the EtherType for a double VLAN tag. The second line of the listing is state 14, sending $0xA8$, the second byte of the EtherType.
Line 3 is the special case, the state variable will jump back to the value 11, the output (in this example) is undefined because the data in VLAN was undefined in Listing 23.
The process shown in Listing 24 will repeat itself for VLAN, because when double VLAN is used there are two tags. When there is no tag in the message the jump in the state will be omitted and the code will continue. When in the future there is a new EtherType with for example three VLAN tags, the jump can be repeated three times. The EtherType is not shown in the Clash type of Listing 23 or 24, only at the output, because the encoding used in Clash connects *VlanDoV4* to $0x88A8$.
The above is a powerful example where the encoding in Clash makes the code easier to read and will avoid making mistakes in what value is connected to a specific EtherType.

5.5 Summary

In the examples above the main functionality of the system is shown, for so more difficult parts of an IPv4 and IPv6 message. The basic structure can be saved in Clash and can be used to make decisions. The examples make clear that different choices in a message give a different output. The extensions show a robust solution even when the extension is unknown, the content can still be used making it a more robust system.

Chapter 6

Conclusion and future work

The main question of this research, and the advantages offered by the Clash type system for the implementation of a multi protocol network interface will be answered using the four sub-questions below.

6.1 Sub-questions

The first sub-question to be answered is:

How can similarities between protocols be handled in the Clash type system?

A large part of the answer follows from the implementation of the Data Link layer described in Section 4.2.2 and the Network layer described in Section 4.2.3. The EthernetType has an unique value for every protocol send in an Ethernet frame, however in most cases IPv4 and IPv6 will be used to sent the same Transport Layer protocol. The similarity between the payload in IPv4 and IPv6 is used in Clash by giving the constructor for IPv4 and IPv6 the same fields. For Clash there is no difference between a TransportType send via IPv4 or IPv6, this is exactly the philosophy given in the Network layer in Section 4.2.3, IPv6 has to solve limitations of IPv4 but serves the same purpose.
Because IPv4 and IPv6 are used for the same purpose, they have a shared type in Clash to store information. Fields in both protocols have a similar meaning and are used for the same record in the Clash type. Using the same record is a really powerful way to get information out of a type in Clash. When there is for example the need to know what Transport layer protocol is send, the *nextProtocol* record will give the result for both IPv4 and IPv6.
The implementation uses the similarities between the different protocols, sharing data types between similar protocols and layers. The data structure in Clash makes it possible to share functionality between data types. The Clash type system will make sharing easier and will give an error when a field is not used in the data type, which prevents making mistakes at compile-time instead of run time.

The second sub-question to be answered is:

Can IPv6 extension headers be handled reliable on an FPGA using Clash?

For answering this question three tests in section 5.2 play a central role, the tests show how the system will handle different extensions. The reliability is best visible in section 5.2 where an extension unknown to the system is fed as an input. Although the extension is unknown for the system the underlying protocol is still recognized. The behavior in this test can be crucial in for example a firewall where one is only interested in what payload is send and what ports are used to do so. The implementation is Clash can give a more reliable choice on blocking a specific protocol even when a client uses new extensions which are unknown to the system. The type system of Clash makes recognizing the extensions easier.

The third sub-question to be answered is:

In what way does the Clash type-checker help to handle Ethernet frames?

The part where the type-checker helps to handle Ethernet frames is the fixed length used in extensions and options. In Figure 4.28 presented in section 4.2.5 three DHCP options are handled all completely different in length. However, filling in the length as specified in the RFC to the corresponding constructor will give a data type in which all options can be stored. The reserved space in hardware will be sufficient for the options using a fixed length. The length of the input buffer will have the size of the longest option, whenever the user will modify the input buffer to a smaller value Clash will give a compile error. The forgoing is important and will give a system in which all supported options and extensions can be handled.

The fourth sub-question to be answered is:

Is the type system of Clash suitable for building a network stack?

Clash can be used to build a network stack, the simulation can be used to read and create an Ethernet message. The design follows the OSI-model, every layer has a separate data type in Clash. The EthernetFrame type presented in section 4.2.2 is used to store and compose an Ethernet message. The Clash type system is suitable for handling Ethernet messages and to generate a DHCP request.
The system can be used up to the Session layer of the OSI-model, higher layers are not implemented because it was expected to be a struggle to do so. In section 3.1 the highest layer of the OSI-model implemented on an FPGA is the Session layer.
All questions above give in part an answer to the main question of this research, showing some advantages of the Clash type system for handling an Ethernet message.

6.2 Main question

The forgoing questions give the advantages of using Clash in constructing a network interface, this is only part of the main question of the research done.

What (dis)advantages will the Clash type system offer when investigating the implementation of a flexible, expandable and generic multi protocol network interface on an FPGA?

There are also disadvantages when constructing an Ethernet stack in Clash, the most obvious being the custom encoding. Every code-block showing a part of the encoding in Chapter 4 is shortened to make it better readable. However there are still a lot of numbers shown, making the code hard to read. The fact all positions of the fields are fixed in the encoding makes it hard to update the encoding when a field is add. The benefit of encoding an Ethernet frame is that no new fields will be add to the standard, which would require all devices connected to Internet to be updated. However, there are fields reused, or split the get extra functionality, to add such changes will require a change in the encoding made in Clash.
The flexible implementation made comes with a cost, every protocol in the session layer will be interpreted as an IPv6 extension when send via the IPv6 protocol, this choice is explained in subsection 4.2.3. It is more likely there will be new extension headers add to the standard than new transport layer protocols. The system can handle new headers and can still be used to filter upper layer protocols out in a firewall application.
The implementation made is flexible; new protocols, options and extensions can be add to the system. This is an important aspect in an Ethernet application where new extensions of IPv6 are under development as explained in subsection 4.2.3. Even in the Network layer new protocols can be add, in the current implementation there is aN entry for EtherCAT, however there was not the time to make an implementation for a full EtherCAT system.
The system can handle two Network layer protocols, and can recognize three, this could be extended further. In every type made in Clash a fall back for protocols not implemented is add. Adding an extra constructor in the data type will represent a new protocol and will make the system more suitable to use on a multi protocol network.

6.3 Future work

In this research a network stack was made and tested using Clash. The code can be compiled to HDL, however the implementation was not tested on an FPGA. Almost all parts are in place to test the system, however there was no time left to do so. Before testing on an FPGA, the reply for DHCP has to be add and a retry should be implemented when there is no offer received in X seconds. The system should be able to get an IP address and will use it when a message needs to be send. For testing, a default message stored in blockram is a good start, later the blockram can be replaced by some system sending messages.
The system can at this moment only send messages via the UDP protocol, there are more upper layer protocols, for example TCP. There are more protocols that are commonly used, however TCP is implemented multiple times on an FPGA making it a good candidate for extending the Ethernet stack.

One protocol where the first parts are implemented in the system is EtherCAT, commonly used in industry. EtherCAT will bring new challenges regarding speed. EtherCAT is a real-time data bus meaning messages should be handled as quickly as possible. Besides the timing a extra physical port should be available to the FPGA. Implementing EtherCAT will be much more work, however the combination of standard Ethernet and EtherCAT will be more common in the future when factories get smarter and more machines will be connected to Ethernet in some way.
Of course there are many more protocols that can be added to the Ethernet stack, however the above protocols are recommended to start with: TCP for the large adoption around the world and EtherCAT as a prof of concept of the capabilities of Clash.

Bibliography

[1] *PROFINET*, Siemens, PI Support Center Haid-und-Neu-Strasse 7 76131 Karlsruhe Germany. [Online]. Available: https://www.profibus.com/technology/profinet/

[2] *EtherCAT Technology Group*, Beckhoff, Ostendstraße 196 90482 Nuremberg Germany. [Online]. Available: https://ethercat.org/default.htm

[3] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "Clash: Structural descriptions of synchronous hardware using haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, 2010, pp. 714–721.

[4] J. Chiang, *Designing with Multiple Industrial Ethernet Protocols on the same FPGA Platform*, Intel, Altera Corporation.

[5] H. Zimmermann, "Osi reference model–the iso model of architecture for open systems interconnection," *IEEE Transactions on communications*, vol. 28, no. 4, pp. 425–432, 1980.

[6] *Hardware Data Sheet Section I, Ethercat Slave Controller*, Version 2.3 ed., Beckhoff, Beckhoff Automation GmbH & Co. KG, Huelshorstweg 20, 33415 Verl, Germany, 02 2017.

[7] X. Wu, L. Xie, and F. Lim, "Network delay analysis of ethercat and profinet irt protocols," in *Industrial Electronics Society, IECON 2014-40th Annual Conference of the IEEE*. IEEE, 2014, pp. 2597–2603.

[8] A. Lofgren, L. Lodesten, S. Sjoholm, and H. Hansson, "An analysis of fpga-based udp/ip stack parallelism for embedded ethernet connectivity," in *NORCHIP Conference, 2005. 23rd*. IEEE, 2005, pp. 94–97.

[9] W. Lu, "Designing tcp/ip functions in fpgas," *from MSc THESIS, computer engineering, Mekelweg*, vol. 4, 2003.

[10] L. Hendriks, "Measuring ipv6 resilience and security," Ph.D. dissertation, University of Twente, Netherlands, 1 2019.

[11] H. Alvestrand, *rfc3935:A Mission Statement for the IETF*, October 2004.

[12] Beckhoff, *Hardware Data Sheet Section I Ethercat slave conntroller V2.3*, Beckhoff, Beckhoff Automation GmbH & Co. KG Huelshorstweg 20 33415 Verl Germany, 02 2017.

[13] D. Law, "Ieee standard for ethernet-amendment 1: Physical layer specification and management parameters for 2.5 gb/s and 5 gb/s operation over backplane," *IEEE Std 802.3 cb-2018 (Amendment to IEEE Std 802.3-2018)*, 2019.

[14] M. Awad, "Fpga supercomputing platforms: A survey," in *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009, pp. 564–568.

[15] V. Rosello, J. Portilla, and T. Riesgo, "Ultra low power fpga-based architecture for wake-up radio in wireless sensor networks," in *IECON 2011-37th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2011, pp. 3826–3831.

[16] R. Chéour, S. Khriji, D. El Houssaini, M. Baklouti, M. Abid, and O. Kanoun, "Recent trends of fpga used for low-power wireless sensor network," *IEEE Aerospace and Electronic Systems Magazine*, vol. 34, no. 10, pp. 28–38, 2019.

[17] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, "A cloud-scale acceleration architecture," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–13.

[18] Intel, *Triple-Speed Ethernet Intel® FPGAIP User Guide*, version 19.4 ed., Intel, Feb. 2020. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_ethernet.pdf

[19] *infineon xmc4300/4800 microcontroller with EterCAT core*, Infineon Technologies AG, Infineon Technologies AG 81726 Munich, Germany, Mar. 2019.

[20] G. S. Tobias Heilmeier, Frank Iwanitz, *Implementing Industrial Ethernet Field Device Functionality by Using FPGAs*, Softing, Richard-Reitzner-Allee 6, DE-85540 Haar, Germany, 2015.

[21] M. Knezic, B. Dokic, and Z. Ivanovic, "Increasing ethercat performance using frame size optimization algorithm," in *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. IEEE, 2011, pp. 1–4.

[22] N. M. Khalilzad, F. Yekeh, L. Asplund, and M. Pordel, "Fpga implementation of real-time ethernet communication using rmii interface," in *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*. IEEE, 2011, pp. 35–39.

[23] C. Haywood, *TCP IP Core*, opencores, Feb. 2016.

[24] M. R. Mahmoodi, S. M. Sayedi, and B. Mahmoodi, "Reconfigurable hardware implementation of gigabit udp/ip stack based on spartan-6 fpga," in *Information Technology and Electrical Engineering (ICITEE), 2014 6th International Conference on*. IEEE, 2014, pp. 1–6.

[25] T. Uchida, "Hardware-based tcp processor for gigabit ethernet," *IEEE transactions on nuclear science*, vol. 55, no. 3, pp. 1631–1637, 2008.

[26] R. Guerra Marín, K. van Berkel, and S. Stuijk, "Implementation of a low-latency ethercat slave controller with support for gigabit networks," Master's thesis, Eindhoven University of Technology, Prodrive Technologies, 2017.

[27] J. Cabal, P. Benáček, L. Kekely, M. Kekely, V. Puš, and J. Kořenek, "Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2018, pp. 249–258.

[28] *IEEE 802 Numbers*, Internet Assigned Numbers Authority (IANA), 01 2020, website. [Online]. Available: https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml#ieee-802-numbers-1

[29] J. Postel *et al.*, *RFC 791: Internet protocol*, 1981.

[30] F. Gont, *RFC 7126: Recommendations on Filtering of IPv4 Packets Containing IPv4 Options*, February 2014.

[31] S. Deering and R. Hinden, "Rfc 2460: Internet protocol, version 6 (ipv6) specification," *Internet Engineering Task Force*, vol. 9, 1998.

[32] R. S. Deering and R. Hinden, "Rfc 8200: Internet protocol, version 6 (ipv6) specification," *Internet Engineering Task Force*, July 2017.

[33] S. Krishnan, J. Woodyatt, J. Kline, E amd Hoagland, and M. Bhatia, *rfc6564: A Uniform Format for IPv6 Extension Headers*, April 2012.

[34] S. Kent, *RFC 4303: IP Encapsulating Security Payload*, December 2005.

[35] M. Cotton, L. Eggert, J. Touch, M. Westerlund, and S. Cheshire, "Internet assigned numbers authority (iana) procedures for the management of the service name and transport protocol port number registry." *RFC*, vol. 6335, pp. 1–33, 2011.

[36] *Assigned Internet Protocol Numbers*, Internet Assigned Numbers Authority (IANA), 01 2020, website. [Online]. Available: https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xhtml

[37] J. Postel *et al.*, *RFC 768: User datagram protocol*, 1980.

[38] *Service Name and Transport Protocol Port Number Registry*, Internet Assigned Numbers Authority (IANA), 01 2020, website. [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

[39] S. Alexander and R. Droms, "Rfc 2132: Dhcp options and bootp vendor extensions," 1997. [Online]. Available: https://tools.ietf.org/html/rfc2132

Appendix A

Code

A.1 Ethernet type

```
data EthernetType
  = IPv4                      IpHeader  TransportType
  | IPv6                      IpHeader  TransportType
  | Vlan    VlanTag           IpHeader  TransportType
  | VlanDo  VlanTag  VlanTag  IpHeader  TransportType
  | Ecat        UnknownFrameHeader      TransportType
  | ProNet      UnknownFrameHeader      TransportType
  | UnknownType UnknownFrameHeader      TransportType
  | IncorrectFrame  IncorrectFrameHeader  SizeEthernetFrame
  deriving  (Lift,  Eq,Generic,NFDataX,  Show,  ShowX)
```

Listing 25: The compleat EthernetType including all data fields.

```
ANN module (
 let lengthIPHeader = ipLength+24
     lengthTransport = 82
     ipLength = 336+lengthExtension+lengthExtensionVec
     lengthExtension = ((snatToNum (SNat @(IPv6ExtensionLength NumberIPv6Extension)))*8 + 8)
     lengthExtensionVec = (snatToNum (SNat @((((SizeEthernetFrame-1)*8) +
   (8*(6+(8*NumberIPv6Extension))+16)*SizeEthernetFrame) +(CLog 2 (SizeEthernetFrame+1)) )))

     lengthVlan = 32
     lengthFrame = lengthIPHeader+lengthTransport+(lengthVlan*3)
     lengthUnFrame = 16
     lengthIncorect = 8*(snatToNum (SNat @(SizeEthernetFrame)))
     eTyp = 0xffff
     vTypeM = 0xffff0000
     vType = 0x81000000
     dvType = 0x88A8000081000000
     dvTypeM = 0xffff0000ffff0000
 in DataReprAnn
  $((liftQ [t|EthernetType|]))
  (lengthFrame)
  [ ConstrRepr 'IPv4
   (shiftL eTyp (lengthFrame-16))
   (shiftL 0x0800 (lengthFrame-16))
  [(shiftL((2^ lengthIPHeader)-1) (lengthFrame-lengthIPHeader)),
   (shiftL((2^ lengthTransport)-1) (lengthFrame-lengthIPHeader-lengthTransport)) ]
  , ConstrRepr 'IPv6
   (shiftL eTyp  (lengthFrame-16))
   (shiftL 0x86DD (lengthFrame-16))
  [(shiftL((2^ lengthIPHeader)-1) (lengthFrame-lengthIPHeader)),
   (shiftL((2^ lengthTransport)-1) (lengthFrame-lengthIPHeader-lengthTransport)) ]
  , ConstrRepr 'Vlan
   (shiftL vTypeM (lengthFrame-(32)))
   (shiftL (vType)  (lengthFrame-(32)))
  [(shiftL(0xffff) (lengthFrame-lengthVlan)),
   (shiftL((2^ lengthIPHeader)-1) (lengthFrame-lengthVlan-lengthIPHeader)),
   (shiftL((2^ lengthTransport)-1)
     (lengthFrame-lengthVlan-lengthIPHeader-lengthTransport)) ]
  , ConstrRepr 'VlanDo
   (shiftL dvTypeM  (lengthFrame-(64)))
   (shiftL (dvType)  (lengthFrame-(64)))
  [(shiftL(0xffff) (lengthFrame-lengthVlan)),
   (shiftL(0xffff) (lengthFrame-(2*lengthVlan))),
   (shiftL((2^ lengthIPHeader)-1) (lengthFrame-(2*lengthVlan)-lengthIPHeader)),
   (shiftL((2^ lengthTransport)-1)
     (lengthFrame-(2*lengthVlan)-lengthIPHeader-lengthTransport)) ]
```

```
46    , ConstrRepr 'Ecat
47     (shiftL eTyp (lengthFrame-16))
48     (shiftL 0x88A4 (lengthFrame-16))
49    [(shiftL(0xffff) (lengthFrame-16-lengthUnFrame)),
50     (shiftL((2^ lengthTransport)-1) (lengthFrame-16-lengthUnFrame-lengthTransport)) ]
51    , ConstrRepr 'ProNet
52     (shiftL eTyp (lengthFrame-16))
53     (shiftL 0x8892 (lengthFrame-16))
54    [(shiftL(0xffff) (lengthFrame-16-lengthUnFrame)),
55     (shiftL((2^ lengthTransport)-1) (lengthFrame-16-lengthUnFrame-lengthTransport)) ]
56    , ConstrRepr 'IncorrectFrame
57     (shiftL eTyp (lengthFrame-16))
58     (shiftL 0xffff (lengthFrame-16))
59    [(shiftL((2^ lengthIncorect)-1) (lengthFrame-lengthIncorect-16))]
60    , ConstrRepr 'UnknownType
61     (shiftL eTyp (lengthFrame-16))
62     (shiftL 0xffff (lengthFrame-16))
63    [(shiftL(0xffff) (lengthFrame-16-lengthUnFrame)),
64     (shiftL((2^ lengthTransport)-1) (lengthFrame-16-lengthUnFrame-lengthTransport)) ]
65    ])
66 deriveBitPack [t| EthernetType |]
```

Listing 26: The compleat Annotation for the EthernetType including all data fields and calculations.

A.2 Annotation IpHeader

```
# ANN module (
  let
    ipLength = 336+lengthExtension+lengthExtensionVec
    lengthExtension = ((snatToNum (SNat
      @(IPv6ExtensionLength NumberIPv6Extension)))*8) +8
    lengthExtensionVec = (snatToNum (SNat @((8*(6+(8*NumberIPv6Extension))+24)*
      (SizeEthernetFrame)+(CLog 2 (SizeEthernetFrame+1)) )))
  in DataReprAnn
    $(liftQ [t|IpHeader|])
    (ipLength+16)
    [ ConstrRepr 'IPv4Header (shiftL 0xffff ipLength) (shiftL 0x0800 ipLength)
    [(shiftL(0xf) (ipLength-4)),
     (shiftL(0xf) (ipLength-8)),
     (shiftL(0x3f) (ipLength-14)),
     (shiftL(0x3) (ipLength-16)),
     (shiftL(0xffff) (ipLength-32)),
     (shiftL(0xffff) (ipLength-48)),
     (shiftL(0x7) (ipLength-51)),
     (shiftL(0x1fff) (ipLength-64)),
     (shiftL(0xff) (ipLength-72)),
     (shiftL(0xff) (ipLength-80)),
     (shiftL(0xffff) (ipLength-96)),
     (shiftL(0xffffffff) (ipLength-128)),
     (shiftL(0xffffffff) (ipLength-160))]
    , ConstrRepr 'IPv6Header (shiftL 0xffff ipLength) (shiftL 0x86DD ipLength)
    [(shiftL(0xf) (ipLength-4)),
     (shiftL(0xff) (ipLength-12)),
     (shiftL(0xfffff) (ipLength-32)),
     (shiftL(0xffff) (ipLength-48)),
     (shiftL(0xffff) (ipLength-64)),
     (shiftL(0xff) (ipLength-72)),
     (shiftL(0xff) (ipLength-80)),
     (shiftL(0xffffffffffffffffffffffffffffffff) (ipLength-208)),
     (shiftL (0xffffffffffffffffffffffffffffffff) (ipLength-336)),
     (shiftL((2^lengthExtension)-1) (ipLength-(336+lengthExtension))),
     (shiftL((2^lengthExtensionVec)-1) (ipLength-(336+lengthExtension+lengthExtensionVec)))]
]) #
deriveBitPack [t| IpHeader |]
```

Listing 27: The compleat Annotation for the IpHeader including all data fields and calculations.

A.3 Ipv6Extention

```
data IPv6Extention
  = FragmentIPv6 (ExtensionHeaderIPv6 6)
  --44 (0x2C)
  | RoutingIPv6  (ExtensionHeaderIPv6 6)
  --optional more bytes
  -- 43 (0x2B)
  | HopByHopIPv6  (ExtensionHeaderIPv6 (IPv6ExtensionLength NumberIPv6Extension))
  -- 0 (0x00)
  | WildcardIPv6 (ExtensionHeaderIPv6 (IPv6ExtensionLength NumberIPv6Extension ))
  deriving (Lift, Eq,Generic,NFDataX, Show, ShowX)
```

Listing 28: The compleat IPv6Extention data type including all data fields.

```
# ANN module (let
    maxOptionLength = (snatToNum (SNat
      @((IPv6ExtensionLength NumberIPv6Extension))))
    maxLen = maxOptionLength*8
    posLen = (shiftL 0xff (maxLen))
    nexLen = (shiftL 0xff (maxLen+8))
  in DataReprAnn
  $((liftQ [t|IPv6Extention|]))
    (maxLen+24)
    [ConstrRepr 'FragmentIPv6 (shiftL 0xff (maxLen+16)) (shiftL 44 (maxLen+16))
      [nexLen + posLen + (2^(8*6)-1) ],
     ConstrRepr 'RoutingIPv6  (shiftL 0xff (maxLen+16)) (shiftL 43 (maxLen+16))
       [nexLen + posLen + (2^(8*6)-1) ],
     ConstrRepr 'HopByHopIPv6 (shiftL 0xff (maxLen+16)) (shiftL 0 (maxLen+16))
       [nexLen + posLen + (2^(8*maxOptionLength)-1) ],
     ConstrRepr 'WildcardIPv6 (shiftL 0x00 (maxLen+16)) (shiftL 0 (maxLen+16))
       [nexLen + posLen + (2^(8*maxOptionLength)-1) ]
    ]) #
deriveBitPack [t| IPv6Extention |]
```

Listing 29: The compleat Annotation for the IpHeader including all data fields and calculations.

A.4 TransportType

```
data TransportType
  = UDP UDPHeader
  | NewProtocol NewProtocolHeader
  deriving (Lift, Eq,Generic,NFDataX, Show, ShowX,BitPack)
```

Listing 30: The compleat TransportType data type including all data fields.

A.5 DHCPBody

```
data DHCPBody n=
    DHCPBody{    --a body
    sourceDHCP            :: EthernetWord,
    lengthDHCP            :: EthernetWord,
    versionDHCP           :: EthernetWord}
  | DHCPBodyV4 {  --a IpV4 body
    messageTypeDHCP       :: Byte,
    sourceDHCP            :: EthernetWord,
    typeDHCP              :: Byte,
    hardwareTypeDHCP      :: Byte,
    hardwareAddressDHCP   :: Byte,
    lengthDHCP            :: EthernetWord,
    versionDHCP           :: EthernetWord,
    hopsDHCP              :: Byte,
    transactionIDDHCP     :: Vec 4 Byte,
    elapsTimeDHCP         :: Vec 2 Byte,
    flagsDHCP             :: Vec 2 Byte,
    cookieDHCP            :: Vec 4 Byte,
    gatewayAddress        :: Vec 6 Byte,
    serverAddress         :: (DHCPServer 4),
    optionsDHCP           :: Vec n DHCPOptions}
  | DHCPBodyV6 {  --a IpV6 body
    messageTypeDHCP       :: Byte,
    sourceDHCP            :: EthernetWord,
    typeDHCP              :: Byte,
    versionDHCP           :: EthernetWord,
    transactionIDDHCP     :: Vec 4 Byte,
    lengthDHCP            :: EthernetWord,
   -- serverAddressV6     :: (DHCPServer 8 ),
    optionsDHCP           :: Vec n DHCPOptions}
  deriving (Lift, Eq,Generic,NFDataX,Show, ShowX, BitPack)
```

Listing 31: The compleat DHCPBody data type including all data fields.

A.6 DHCPOptions

```
1  data DHCPOptions
2    = DHCPOptionRouter  (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
3    | DHCPOptionIdentifier (DHCPOption 4)
4    | DHCPOptionIPForwarding (DHCPOption 1)
5    | DHCPOptionNonLocalSource (DHCPOption 1)
6    | DHCPOptionLimeToLive (DHCPOption 1)
7    | DHCPOptionLeasTime (DHCPOption 4)
8    | DHCPOptionClientIdentifier (DHCPOption 7)
9    | DHCPOptionTimeOffset (DHCPOption 4)
10   | DHCPOptionSubnetMask (DHCPOption 4)
11   | DHCPOptionRequestedIP  (DHCPOption 4)
12   | DHCPOptionRenewalTime  (DHCPOption 4)
13   | DHCPOptionRebindingTime (DHCPOption 4)
14   | DHCPOptionParameterRequestList (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
15   | DHCPOptionDomainName  (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
16   | DHCPOptionDomainNameServer  (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
17   | DHCPOptionMessageType  (DHCPOption 1)
18   | DHCPOptionEndmark  (DHCPOption 0)
19   | DHCPOptionWildcard  (DHCPOption ((DHCPMaxOptionLength DHCPOptionLength)))
20   deriving (Lift, Eq,Generic,NFDataX,Show, ShowX)
```

Listing 32: The compleat DHCPOptions data type including all data fields.

```
# ANN module (let
    maxOptionLength = (snatToNum (SNat @(DHCPMaxOptionLength DHCPOptionLength)))
    maxLen = maxOptionLength*8
    posLen = (shiftL 0xff (maxLen))
  in DataReprAnn
    $((liftQ [t|DHCPOptions|]))
    (maxLen+16)
    [ConstrRepr 'DHCPOptionIdentifier (shiftL 0xff (maxLen+8)) (shiftL 54 (maxLen+8))
      [posLen + (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionIPForwarding (shiftL 0xff (maxLen+8)) (shiftL 19 (maxLen+8))
      [posLen + (2^(8*1)-1) ],
    ConstrRepr 'DHCPOptionNonLocalSource (shiftL 0xff (maxLen+8)) (shiftL 20 (maxLen+8))
      [posLen + (2^(8*1)-1) ],
    ConstrRepr 'DHCPOptionLimeToLive (shiftL 0xff (maxLen+8)) (shiftL 23 (maxLen+8))
      [posLen + (2^(8*1)-1) ],
    ConstrRepr 'DHCPOptionLeasTime (shiftL 0xff (maxLen+8)) (shiftL 51 (maxLen+8))
      [posLen +  (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionClientIdentifier (shiftL 0xff (maxLen+8)) (shiftL 61 (maxLen+8))
      [posLen +  (2^(8*7)-1) ],
    ConstrRepr 'DHCPOptionTimeOffset (shiftL 0xff (maxLen+8)) (shiftL 2 (maxLen+8))
      [posLen + (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionSubnetMask (shiftL 0xff (maxLen+8)) (shiftL 1 (maxLen+8))
      [posLen + (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionRequestedIP (shiftL 0xff (maxLen+8)) (shiftL 50 (maxLen+8))
      [posLen + (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionRenewalTime (shiftL 0xff (maxLen+8)) (shiftL 58 (maxLen+8))
      [posLen + (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionRebindingTime (shiftL 0xff (maxLen+8)) (shiftL 59 (maxLen+8))
      [posLen + (2^(8*4)-1) ],
    ConstrRepr 'DHCPOptionRouter (shiftL 0xff (maxLen+8)) (shiftL 3 (maxLen+8))
      [(2^(8*maxOptionLength+8)-1) ],
    ConstrRepr 'DHCPOptionParameterRequestList (shiftL 0xff (maxLen+8))
      (shiftL 55 (maxLen+8))
      [(2^(8*maxOptionLength+8)-1) ],
    ConstrRepr 'DHCPOptionDomainName (shiftL 0xff (maxLen+8)) (shiftL 15 (maxLen+8))
      [(2^(8*maxOptionLength+8)-1) ],
    ConstrRepr 'DHCPOptionDomainNameServer (shiftL 0xff (maxLen+8)) (shiftL 6 (maxLen+8))
      [(2^(8*maxOptionLength+8)-1) ],
    ConstrRepr 'DHCPOptionMessageType (shiftL 0xff (maxLen+8)) (shiftL 53 (maxLen+8))
      [posLen + (2^(8*1)-1) ],
    ConstrRepr 'DHCPOptionEndmark (shiftL 0xff (maxLen+8)) (shiftL 255 (maxLen+8))
      [posLen + (2^(8*0)-1) ],
    ConstrRepr 'DHCPOptionWildcard (shiftL 0x00 (maxLen+8)) (0)
      [2^(maxLen+8)-1]
    ]) #
deriveBitPack [t| DHCPOptions |]
```

Listing 33: The compleat Annotation for the DHCPOptions including all data fields and calculations.

Appendix B

Wireshark

B.1 HopByHop extension

```
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  Destination: Broadcast (ff:ff:ff:ff:ff:ff)
  Source: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Type: IPv6 (0x86dd)
Internet Protocol Version 6, Src: ::0.1.18.52, Dst: ::0.2.152.118
  0110 .... = Version: 6
  .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
  .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
  Payload Length: 44
  Next Header: IPv6 Hop-by-Hop Option (0)
  Hop Limit: 64
  Source: ::0.1.18.52
  Destination: ::0.2.152.118
  IPv6 Hop-by-Hop Option
    Next Header: UDP (17)
    Length: 1
    [Length: 16 bytes]
    PadN
      Type: PadN (0x01)
      Length: 12
      PadN: 313233343536373839307878
User Datagram Protocol, Src Port: 1234, Dst Port: 8888
  Source Port: 1234
  Destination Port: 8888
  Length: 28
  Checksum: 0x36f5 [unverified]
  [Checksum Status: Unverified]
  [Stream index: 0]
  [Timestamps]
Data (20 bytes)
  Data: 736f6d652072616e646f6d207261772064617461
  [Length: 20]
```

Listing 34: The compleat HopByHop extension, caption from Wireshark.

B.2 HopByHop and Fragment extension

```
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
    Destination: Broadcast (ff:ff:ff:ff:ff:ff)
    Source: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Type: IPv6 (0x86dd)
Internet Protocol Version 6, Src: ::0.1.18.52, Dst: ::0.2.152.118
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 52
    Next Header: IPv6 Hop-by-Hop Option (0)
    Hop Limit: 64
    Source: ::0.1.18.52
    Destination: ::0.2.152.118
    IPv6 Hop-by-Hop Option
        Next Header: Fragment Header for IPv6 (44)
        Length: 1
        [Length: 16 bytes]
        PadN
            Type: PadN (0x01)
                00.. .... = Action: Skip and continue (0)
                ..0. .... = May Change: No
                ...0 0001 = Low-Order Bits: 0x01
            Length: 12
            PadN: 313233343536373839307878
    Fragment Header for IPv6
        Next header: UDP (17)
        Reserved octet: 0x00
        0000 0000 0000 0... = Offset: 0 (0 bytes)
        .... .... .... .00. = Reserved bits: 0
        .... .... .... ...1 = More Fragments: Yes
        Identification: 0x00000999
    Reassembled IPv6 in frame: 2
Data (28 bytes)
    Data: 04d222b8001c36f5736f6d652072616e646f6d2072617720...
    [Length: 28]
```

Listing 35: The compleat HopByHop and Fragment extension, caption from Wireshark.

B.3 DHCP caption

```
1  Ethernet II, Src: Cisco_ee:0e:d3 (00:6c:bc:ee:0e:d3), Dst: IntelCor_6a:b8:00 (b0:35:9f:6a:b8:00)
2      Destination: IntelCor_6a:b8:00 (b0:35:9f:6a:b8:00)
3      Source: Cisco_ee:0e:d3 (00:6c:bc:ee:0e:d3)
4      Type: IPv4 (0x0800)
5  Internet Protocol Version 4, Src: 192.168.102.1, Dst: 192.168.102.139
6      0100 .... = Version: 4
7      .... 0101 = Header Length: 20 bytes (5)
8      Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
9      Total Length: 576
10     Identification: 0x0000 (0)
11     Flags: 0x0000
12     Fragment offset: 0
13     Time to live: 64
14     Protocol: UDP (17)
15     Header checksum: 0x2ad0 [validation disabled]
16     [Header checksum status: Unverified]
17     Source: 192.168.102.1
18     Destination: 192.168.102.139
19 User Datagram Protocol, Src Port: 67, Dst Port: 68
20     Source Port: 67
21     Destination Port: 68
22     Length: 556
23     Checksum: 0xd1ba [unverified]
24     [Checksum Status: Unverified]
25     [Stream index: 1]
26     [Timestamps]
27         [Time since first frame: 0.000000000 seconds]
28         [Time since previous frame: 0.000000000 seconds]
29 Dynamic Host Configuration Protocol (Offer)
30     Message type: Boot Reply (2)
31     Hardware type: Ethernet (0x01)
32     Hardware address length: 6
33     Hops: 0
34     Transaction ID: 0x19936a50
35     Seconds elapsed: 0
36     Bootp flags: 0x0000 (Unicast)
37         0... .... .... .... = Broadcast flag: Unicast
38         .000 0000 0000 0000 = Reserved flags: 0x0000
39     Client IP address: 0.0.0.0
40     Your (client) IP address: 192.168.102.139
41     Next server IP address: 192.168.102.1
42     Relay agent IP address: 0.0.0.0
43     Client MAC address: IntelCor_6a:b8:00 (b0:35:9f:6a:b8:00)
44     Client hardware address padding: 00000000000000000000
45     Server host name not given
46     Boot file name not given
47     Magic cookie: DHCP
48     Option: (53) DHCP Message Type (Offer)
49         Length: 1
50         DHCP: Offer (2)
```

```
51     Option: (54) DHCP Server Identifier (192.168.102.1)
52         Length: 4
53         DHCP Server Identifier: 192.168.102.1
54     Option: (51) IP Address Lease Time
55         Length: 4
56         IP Address Lease Time: (86400s) 1 day
57     Option: (1) Subnet Mask (255.255.255.0)
58         Length: 4
59         Subnet Mask: 255.255.255.0
60     Option: (3) Router
61         Length: 4
62         Router: 192.168.102.1
63     Option: (6) Domain Name Server
64         Length: 8
65         Domain Name Server: 8.8.8.8
66         Domain Name Server: 8.8.4.4
67     Option: (255) End
68         Option End: 255
69     Padding: 000000000000000000000000000000000000000000000000...
```

Listing 36: The compleat DHCP offer, caption from Wireshark.