Mini Project Report

On

In Hardware Proxy and Rewrite Module (IPRM)

Submitted By

Rohit T P **Sunith VS**

Najid Navas **Alka T James**

In partial fulfilment of the requirements for the award of degree of Bachelor of Technology in Computer Science and Engineering.

DIVISION OF COMPUTER SCIENCE AND ENGINEERING
SCHOOL OF ENGINEERING COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY

ACKNOWLEDGEMENT

The project was made possible with the support and help of our guide **Mr.Vinod Kumar PP**. We thank him for this. Also, we thank our HOD **Dr. Sudeep Elayidom**, project coordinators **Ms. Ancy Zachariah** and **Ms. Aiswarya** for their guidance in helping us choose this topic.

Rohit T P
Sunith V S
Najid Navas
Alka T James

ABSTRACT

Network computing refers to performing computation within the network rather than forwarding data to a processing unit for computation. This mode of operation helps cut down latency by avoiding the many slow software and hardware pipelines to be used to process network data. Historically the approach for establishing In-Network computing is by using dedicated hardware between the network endpoint and the processing machine. This proves costly and hard to set up and maintain as there is the added work of maintaining separate hardware (cooling, security, stability).

Current state of art systems uses a more innovative approach. They integrate hardware accelerator cards directly into the computing machine to solve most drawbacks of separate hardware pass-through. These accelerator cards perform tasks common in the network infrastructure, like HMAC calculation, MD5 verification, SSL signing, Etc, without consuming valuable CPU resources. Even though costly, these cards are widely used in the industry to optimizes throughput and reduce latency.

In the project, a new hardware accelerator device is introduced, capable of direct integration into the NIC. A NIC add-on module has been created as part of the project, enabling the interception and modification of network packets after they are received by the NIC and before they are forwarded to the PCI bus. The hardware, known as the In-Hardware Proxy and Rewrite Module (IPRM), in a sense, performs a man-in-the-middle attack.

List of Figures

2.1 Arm based smart NICs [14] 3
2.2 Cloudflare Edge ASIC [16] 3
2.3 Xilinx Artix 7[17] 4

4.1 Data Flow Diagram 14
4.2 Block Diagram 15
4.3 Manchester Decoding, [15] 17
4.4 Data packet of IPv4 [18] 18

5.1 Timing Diagram 23
5.2 Schematic Diagram Decoder 24
5.3 Schematic Diagram Parser 25

6.1 Simple Packet Received 27
6.2 Simple Packet Output 28
6.3 Complex Packet Received 29
6.4 Complex Packet Output 30

C.1 Kernel 43
C.2 OS Main Function 44
C.3 Manchester Decoder 45
C.4 Packet Parser 46
C.5 Checksum Calculator 47
C.6 Custom Logic Block 47
C.7 Python Script 48

Contents

Abstract i

List of figures ii

1 Introduction **1**

2 System Study **2**

2.1 Existing System 2

2.1.1 SmartNICs 2

2.1.2 Programmable switch ASICs 3

2.1.3 FPGAs 4

2.2 Literature Review 4

2.2.1 Enabling Technologies 5

2.2.2 Application Classes 5

2.2.3 Programming Languages and Models 5

2.2.4 Challenges and Future Directions 6

2.3 Proposed System 6

3 Objectives **8**

4 System Design **11**

4.1 Algorithm 11

4.1.1 Manchester Decoder 11

4.1.2 Packet Parser 12

4.1.3 IP Checksum Generator 13

4.1.4 Ethernet Checksum Generator 13
4.2 Design Diagrams 14
4.3 Methodologies 15
4.3.1 Network Packet 15
4.3.2 Manchester Decoding 16
4.3.3 IP Header Parsing 17
4.3.4 Data Extraction 18
4.3.5 Destination Port Modification 19

5 System Implementation **20**
5.1 Modules 20
5.1.1 Manchester Decoder 20
5.1.2 Packet Parser 21
5.1.3 Checksum Calculator 21
5.1.4 Custom Logic Block 22
5.2 Tools and Languages 22
5.2.1 Python Script 22
5.2.2 Free Standing OS 23
5.3 Timing Diagram 23
5.4 Circuit Diagram 23

6 Result **26**
6.1 Sample Packets 26
6.1.1 Observation 30

7 Conclusion **32**

References **32**

Appendix A LLM (Large Language Model) 36

Appendix B ASIC (Application Specific Integrated Circuit) 39

Appendix C Source Code 42

Chapter 1

Introduction

The NIC add on module introduced in this project is designed to parse incoming UDP packets, extract their data, and modify the destination port based on the complexity of the received data. The complexity is measured based on the number of non ASCII characters. The system is easily expandable to support other protocols, and the switching logic can be modified based on the system's requirements.

The project demonstrates the potential of in network computing to significantly improve network performance by reducing the load on central processing units (CPUs) and facilitating efficient data processing. Integrating the IPRM directly into the NIC allows interception and modification of network packets in real time, reducing latency and improving throughput.

In addition to its performance benefits, the IPRM is also designed to be easily expandable and customizable. This allows users to adapt the system to support other protocols and modify the switching logic based on their specific requirements. The project represents a significant step forward in the field of in network computing, and it has the potential to pave the way for further innovation in this area.

Chapter 2

System Study

A comprehensive study of the existing systems presented in this section, their drawbacks and how our system improves on the current state of the art.

2.1 Existing System

Current state of art systems uses a more innovative approach. They integrate hardware accelerator cards directly into the computing machine to solve most drawbacks of separate hardware pass through. These accelerator cards perform tasks common in the network infrastructure, like HMAC calculation, MD5 verification, SSL signing, etc, without consuming valuable CPU resources. Even though costly, these cards are widely used in the industry to optimise throughput and reduce latency

2.1.1 SmartNICs

A SmartNIC is a network interface card with an onboard processor, allowing it to perform computation within the network. SmartNICs can be used to offload tasks from the central processing unit (CPU), freeing up processing power and improving network performance.A simple ARM based smart NIC is shown in the figure 2.1. They can be programmed to perform a wide range of tasks, including packet processing, encryption, and traffic management. Figure 2.1 shows a PCI Express smart NIC commercial produced by Broadcom.

Figure 2.1: Arm based smart NICs [14]

2.1.2 Programmable switch ASICs

A programmable switch ASIC is a type of switch that includes an application specific integrated circuit (ASIC) that can be programmed by the user. This allows the switch to perform computation within the network itself, reducing latency and improving throughput. Programmable switch ASICs can be used to implement a wide range of functionality, including routing, load balancing, and security. Figure 2.2 shows the internal view of Cloudflare Edge NIC.

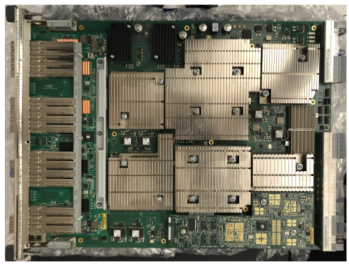

Figure 2.2: Cloudflare Edge ASIC [16]

2.1.3 FPGAs

A field programmable gate array (FPGA) is a type of integrated circuit that can be programmed by the user to perform a wide range of tasks. FPGAs can be used for in network computing by implementing custom logic within the network. This allows FPGAs to perform computation at high speeds, reducing latency and improving network performance. Figure 2.3 shows Artix 7 and entry level FPGEA by Xilinx.

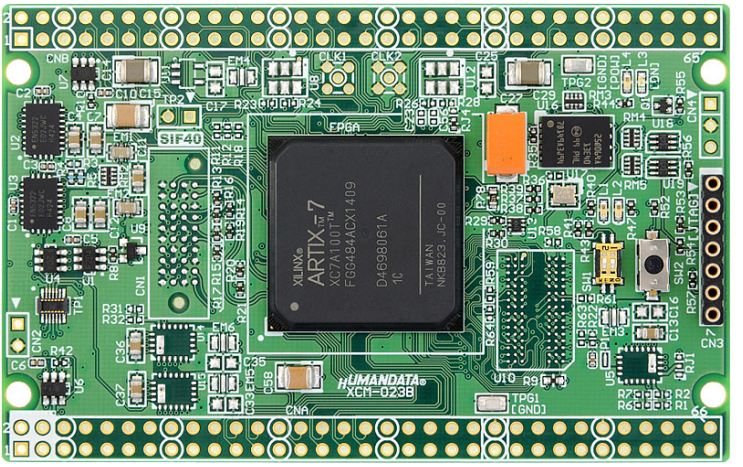

Figure 2.3: Xilinx Artix 7[17]

2.2 Literature Review

In recent years, network computing has emerged as a promising research area, revolutionizing the way network functions and data processing tasks are performed. This approach involves executing programs within network devices, such as programmable switch ASICs and SmartNICs, rather than relying solely on traditional end host computing. The literature in this field highlights the enabling technologies, application classes, programming languages, and models used in in network comput-

ing, while also addressing the associated challenges and future directions.

2.2.1 Enabling Technologies

The implementation of in network computing has been made possible by advancements in hardware technologies. Programmable switch ASICs play a significant role by allowing users to define custom packet processing logic, enabling in network computations at wire speed. Additionally, SmartNICs equipped with their processing capabilities provide the flexibility to offload tasks traditionally performed by the CPU, making in network computing more efficient and practical [16].

2.2.2 Application Classes

In network computing has been applied to various classes of network services and data processing tasks. One prominent application class involves the deployment of network functions directly within network devices, leading to more distributed and scalable network architectures [1]. Another important application is caching, where network devices store frequently requested data, reducing latency and improving application performance [9]. Data reduction and aggregation, which involves consolidating data from multiple sources within network devices, also benefit from in network computing's efficiency [10].

2.2.3 Programming Languages and Models

Several high level programming languages and models have been developed to facilitate in network computing. P4 (Programming Protocol Independent Packet Processors) is a widely used language that allows the specification of packet processing behavior in network devices [15]. P4 enables the customization of packet parsing, matching, and modifi-

cation, making it well suited for implementing custom network functionality. Other programming models like P4runtime and languages such as C/C++ have also been explored to program network devices and provide additional flexibility and abstraction [11, 12].

2.2.4 Challenges and Future Directions

Despite its potential, in network computing comes with challenges that need to be addressed. Security is a primary concern, given that allowing arbitrary code execution in network devices can introduce vulnerabilities. Proper access controls and isolation mechanisms are essential to ensure the integrity of the network [1]. Additionally, ensuring compatibility and interoperability among various programmable network devices remains a challenge, calling for standardization efforts and collaborative initiatives within the industry [5].

Looking ahead, in network computing is expected to undergo further advancements, equipping network devices with even more sophisticated computational capabilities. As the demand for low latency, high performance networking solutions continues to rise, in network computing is poised to play a vital role in shaping the future of network architectures and applications.

2.3 Proposed System

The project's proposed In Hardware Proxy and Rewrite Module (IPRM) is a new hardware accelerator device that can be integrated directly into the NIC. The project's IPRM intercepts and modifies network packets after they are received by the NIC and before they are forwarded to the PCI bus.

The project's NIC add on module is designed to parse incoming UDP packets, extract their data, and modify the destination port based on the complexity of the received data. The complexity is measured based on the number of non ASCII characters. The system is easily expandable to support other protocols, and the switching logic can be modified based on the requirements of the system.

The goal of the project is not to compete with commercial in network computing devices but rather to demonstrate the potential of in network computing to significantly improve network performance. By integrating the project's IPRM directly into the NIC, it is able to intercept and modify network packets in real time, reducing latency and improving throughput.

The proposed system will have:

- Manchester decoder module.
- Stream buffer.
- Constant size constant time packet parser.
- In stream parity calculator.
- Conditional stream in place modifier.
- Arbitrary packet routing integration.

Chapter 3

Objectives

The project aims to achieve the following objectives by implementing the In Hardware Proxy and Rewrite Module (IPRM) system:

1. **Reduce Network Latency**: Minimize network latency by performing computation within the network device itself, thereby eliminating the need to forward data to a central processing unit (CPU) for computation.

2. **Offload CPU Processing**: Offload CPU processing by implementing processing logic directly in the Network Interface Card (NIC), optimizing CPU utilization and enhancing overall system performance.

3. **Customizability and Flexibility**: Design the IPRM system to be easily expandable and customizable, allowing users to modify the switching logic and incorporate new functionalities to suit specific network environments and application scenarios.

4. **Demonstrate the Potential of In Network Computing**: Showcase the benefits of in network computing through the implementation of a custom hardware accelerator device in the NIC, leading to reduced latency and improved throughput.

5. **Benchmark Performance against Existing Solutions**: Compare the performance of the IPRM system against existing solutions, such as running a custom OS or using popular software based network proxies like Nginx, to evaluate its efficiency and effectiveness.

6. **Real World Application Demonstration**: Illustrate a real world application of the IPRM system by employing it as a reverse proxy and load balancer for a powerful server running multiple Language Model Models (LLMs).

7. **Investigate Cost Effectiveness**: Analyze the cost effectiveness of implementing the IPRM system compared to other solutions, considering the upfront investment for fabricating custom hardware and assessing long term benefits and savings in terms of network performance and resource utilization.

8. **Evaluate Scalability and Performance**: Assess the scalability of the IPRM system in handling increasing network traffic and its ability to maintain low latency under heavy workloads.

9. **Contribute to In Network Computing Research**: Make a meaningful contribution to the field of in network computing research by proposing a novel hardware accelerator device (IPRM) and presenting its practical applications and benefits for researchers and industry professionals.

10. **Improve Energy Efficiency**: Explore the potential energy efficiency gains by performing certain computations directly in the NIC, leading to reduced power consumption and greener network operations.

11. **Develop Comprehensive Documentation**: Create detailed documentation and guidelines for the IPRM system, facilitating its implementation, configuration, and maintenance by other researchers and network administrators.

12. **Open Source Implementation**: Provide an open source implementation of the IPRM system, encouraging collaboration, feedback, and further advancements in the field of in network computing.

13. **Promote Adoption of In Network Computing Technologies**: Promote the adoption of in network computing technologies in practical network architectures, by showcasing the benefits and applications of the IPRM system in various real world scenarios.

Chapter 4

System Design

Here the algorithm and logic for each module of the system are presented. The system design includes in depth details about the expected working of all modules and how they are to be interconnected. The methodology used in the design of the system and the components used in it is also explained.

4.1 Algorithm

Algorithms based on the Ethernet and IP spec is used to implement the system. The algorithm for each module is specified below.

4.1.1 Manchester Decoder

The Manchester decoder receives Manchester encoded bits from the NIC in port and decodes it to binary data. The decoded data is placed in a preallocated buffer. The algorithm 1 illustrates the algorithmic principle behind the working of this module.

Algorithm 1 Manchester Decoder Algorithm

Require: $trigger = 0$
  $N \leftarrow 1$
  while $True$ **do**
    if N is even **then**
      if $read = previous = 0$ **then**
        $trigger \leftarrow 1$
      else if $previous = 0$ **then**
        $buffer \leftarrow 0$
      else
        $buffer \leftarrow 1$
      end if
    else
      $previous \leftarrow read$
    end if
  end while

4.1.2 Packet Parser

The packet parser receives input from the buffer *decoded* and parses the data when *trigger* is exerted. The parsed data includes crucial information from the packet, such as source and destination ports, protocol, and data payload. Refer to algorithm 2.

Algorithm 2 Packet Parser Algorithm

Require: $trigger = 1$
  $ethertype \leftarrow decoded[392 - 1 : (392 - 16)]$
  $protocol \leftarrow decoded[304 - 1 : (304 - 8)]$
  $ethertype\_reg \leftarrow ethertype$
  $protocol\_reg \leftarrow protocol$
  if $ethertype = 2048$ and $protocol = 17$ **then**
    $source\_port \leftarrow decoded[216 - 1 : (216 - 16)]$
    $dest\_port\_original \leftarrow decoded[200 - 1 : (200 - 16)]$
    $dest\_port \leftarrow decoded[200 - 1 : (200 - 16)]$
    $data \leftarrow decoded[166 - 1 : 32]$
  end if

4.1.3 IP Checksum Generator

The IPRM needs to recalculate the IP checksum using the modified data from the buffer since it modified the packet. This is to ensure the services listening on the IP layer do not reject the packet due to corrupted checksum. The working is detailed in algorithm 3.

Algorithm 3 IP Checksum Algorithm

$sum \leftarrow 0$
$carry \leftarrow 0$
for $i \leftarrow 0$ to $length$ **do**
    $sum \leftarrow sum + data[i]$
    $carry \leftarrow (sum > 65535)$
    $sum \leftarrow sum + carry$
    $sum \leftarrow sum \& 65535$
end for
$checksum \leftarrow \sim sum$

4.1.4 Ethernet Checksum Generator

The IPRM needs to recalculate the Ethernet checksum using the modified data from the buffer since it modified the packet. This is to ensure the services listening on the physical and network layer do not reject the packet due to corrupted checksum. The logic to generate checksum is explained in algorithm 4.

Algorithm 4 Ethernet Frame Checksum Algorithm

```
crc ← 0
for i ← 0 to length do
    crc ← crc ⊕ data[i]
    for j ← 0 to 7 do
        if crc is odd then
            crc ← (crc >> 1) ⊕ poly
        else
            crc ← crc >> 1
        end if
    end for
end for
checksum ← crc
```

4.2 Design Diagrams

The data flow of the system is depicted in the data flow diagram 4.1. *decode_i* and *decode_i_0* shown in the figure are the entry points and they receive the partial bits from the Manchester buffer. The outputs are depicted as *data*[134 : 0], *decoded*[487 : 0], *dest_port*[15 : 0], *source_port*[15 : 0].

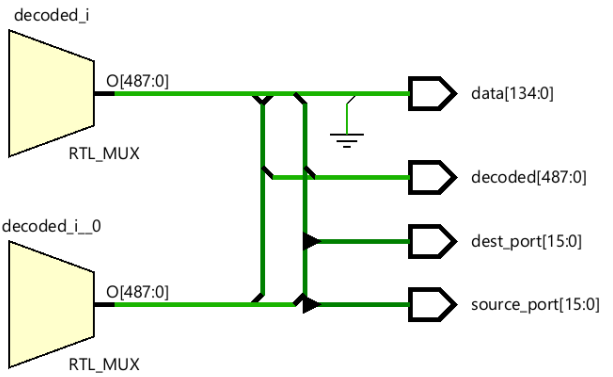

Figure 4.1: Data Flow Diagram

4.3 Methodologies

IPRM uses a multi block architecture where each block is modularised and fully hot swappable. The interconnection of each block in sequential order is shown in figure 4.2

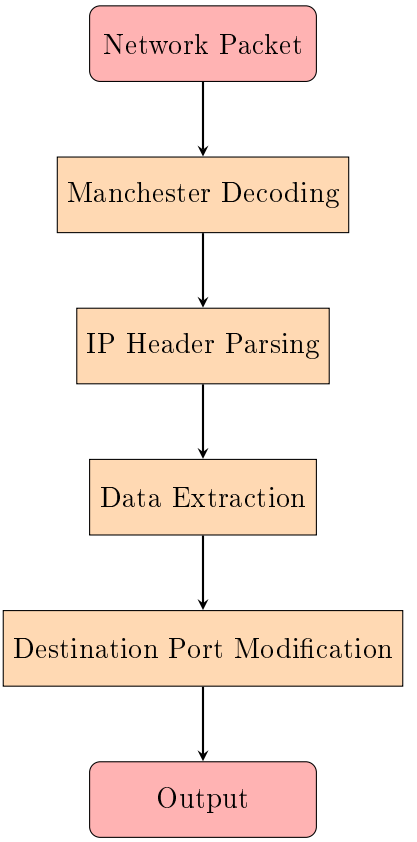

Figure 4.2: Block Diagram

4.3.1 Network Packet

Network packets are fundamental units of data used in computer networking, playing a pivotal role in the efficient and reliable transmission of information across networks like the internet. From a computer science perspective, network packets are akin to data structures that carry both

data and metadata. A typical packet consists of three main components: the header, payload, and trailer. The header contains control information, such as source and destination addresses, packet size, and protocol details. It assists routers and switches in determining the most efficient path for the packet to reach its destination. The payload is the actual data being transmitted, encompassing various types of information, such as files, images, videos, or any other data. Its size varies based on the content being sent. Finally, the trailer is responsible for error checking and data integrity during transmission. Including a checksum or cyclic redundancy check (CRC) value, the trailer allows the receiving device to verify the packet's integrity and detect any potential errors.

During data transmission, network packets are sent individually and can follow different routes through the network. Routers and switches efficiently direct packets along the most favorable path, ensuring optimal data delivery and load balancing across the network. Upon reaching the destination, the receiving device reassembles the packets based on their headers and reconstructs the original message or file. This process enables error recovery and guarantees that the transmitted data remains intact. Understanding network packets is essential for computer scientists, as it enables them to design and optimize network protocols and systems, ensuring seamless data transfer and communication across various applications and systems.

4.3.2 Manchester Decoding

Manchester encoding is a technique used to represent digital data as a series of transitions between high and low voltage levels within fixed time intervals. Each bit in the data is transmitted as two half bit periods, where the presence or absence of a transition in the middle of the bit

determines the binary value.

The process of Manchester decoding aims to recover the original binary data from the encoded signal. It involves analyzing the transitions and timing within each bit period to determine the corresponding binary value. If a transition occurs in the middle of a bit period, it signifies a '0', while its absence indicates a '1'. This reversal of logic is a distinguishing feature of Manchester encoding, making it self clocking and less susceptible to synchronization issues.

Manchester decoding is widely used in various networking protocols and technologies, including Ethernet, token rings, and certain wireless communication standards. Its robustness in clock recovery and ease of implementation make it an attractive choice for transmitting data reliably across networks. The accurate decoding of Manchester encoded signals ensures the precise retrieval of data, playing a crucial role in efficient and error free communication between devices and systems in computer networks. Figure 4.3 shows the timing diagram for a common Manchester encoded signal.

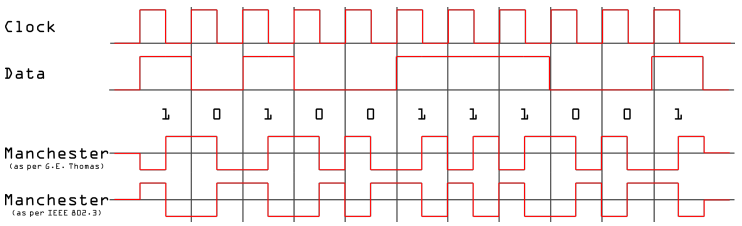

Figure 4.3: Manchester Decoding, [15]

4.3.3 IP Header Parsing

The IP header is a fixed size section located at the beginning of each IP packet and contains crucial fields that facilitate packet routing and de-

livery (refer figure 4.4). During parsing, the device extracts fields such as the source and destination IP addresses, Time to Live (TTL), and Protocol. The source and destination IP addresses enable routers to determine the packet's origin and intended destination, essential for forwarding decisions. The TTL field limits the packet's lifetime, preventing it from endlessly circulating in the network, and the Protocol field identifies the transport layer protocol (TCP, UDP, etc.) to which the packet's payload should be delivered.

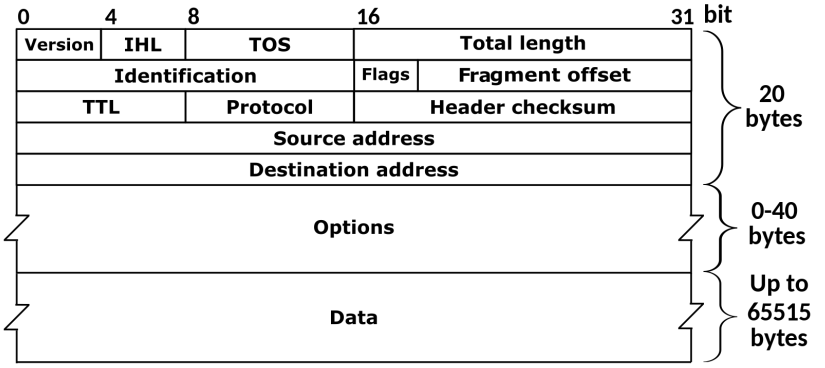

Figure 4.4: Data packet of IPv4 [18]

4.3.4 Data Extraction

This section involves capturing pieces of information from the decoded packet. The essential details that are extracted include the source port, which represents the originating port number; the destination port, indicating the intended delivery port; the source MAC address, a unique hardware identifier of the sender device; the destination MAC address, identifying the intended recipient device; the source IP address, revealing the packet's origin; and the destination IP address, specifying the intended delivery location. These extracted pieces of information are

vital for further analysis, routing decisions, and facilitating efficient communication and delivery of the packets across the network.

4.3.5 Destination Port Modification

The destination port modification is influenced by the complexity of the data contained within the packet. To measure the data complexity, the occurrence of Unicode characters in the data field is counted. The more non ASCII Unicode characters present, the higher the perceived complexity of the data. Consequently, the destination port is incremented accordingly based on this complexity measurement.

This approach finds utility in a large language model (LLM) operating within the network. By dynamically adjusting the destination port based on data complexity, the LLM can effectively cater to various languages and communication types. When encountering packets with simple data (few non ASCII characters), the LLM may prioritize the use of language models optimized for handling common languages efficiently. Conversely, when processing packets with complex data (containing numerous non ASCII characters), the LLM can switch to language models more adept at handling diverse languages and character sets. Such dynamic adaptation empowers the LLM to enhance language recognition, optimize processing mechanisms, and ultimately improve overall network performance while accommodating multilingual data processing needs.

Chapter 5

System Implementation

The system was implemented as four modules,

1. Manchester Decoder
2. Packet Parser
3. Checksum Calculator
4. Custom Logic Block

5.1 Modules

All modules are independently fabricated as Verilog blocks and combined by attaching wires between them. The system uses a mix of both synchronous and asynchronous modules.

5.1.1 Manchester Decoder

The Manchester decoder is a block consisting of an input stream and an output buffer and an enable line. Initially, the enable line is asserted so the block is operational at time $t = 0$. At every positive edge of the clock, the Manchester Decoder receives n^{th} semi bit. This semi bit is stored in a temporary register and an internal counter is incremented. Whenever the counter value becomes odd the current semi bit is combined with the previous one to decode the current but it is written to the output buffer. This process continues until two consecutive semi bits are zero (indicating silence). At this point, this block is disabled and an interrupt

is raised. Refer to source code C.3 for the Verilog implementation of this module.

5.1.2 Packet Parser

The packet parser is the module that reads from the output buffer of the Manchester Decoder and parses the network packet. This block consists of one input, one output and two bi directional wires. The input is connected to the output buffer of Manchester Decoder and the output is connected to the output buffer of the NIC. The bidirectional wires are used to integrate Ethernet Checksum and IP check Sum. The packet parser initially stays in the disabled state and is woken up by the interrupt produced by Manchester Decoder. Once the interrupt is received this module activates and in one clock cycle parses the entire frame as the offsets of all required data points are already known. Refer to source code C.4 for the Verilog implementation of this module.

5.1.3 Checksum Calculator

The checksum calculator is a two part block that attaches directly to the Packet Parser through the bidirectional wires. The Checksum Calculators are asynchronous blocks that continuously update the checksum values as the data in the Manchester Decoder output buffer changes. The asynchronous nature of the calculator can be maintained without introducing race conditions due to the fact that the checksum calculation is a constant time process for a packet of constant size. Refer to source code C.5 for the Verilog implementation of this module.

5.1.4 Custom Logic Block

The custom logic block is a virtual module that sits inside the Packet Parser. It holds custom logic that can operate on the packet that was parsed previously by the Packet Parser. This module is kept flexible to accommodate future scope for expansion. This module is capable of operating as a simple Turing Machine by considering the Packet Parser Output line as the tape. This block first examines the Ether type and Protocol to determine whether it should operate on the packet at hand. Once the packet passes this verification the complexity of data contained in it is assessed based on the number of non unicode characters present. Depending on the complexity the output port is decided by giving a higher port number to more complex data. Refer to source code C.6 for the Verilog implementation of this module.

5.2 Tools and Languages

The fabrication process was done in Verilog using Vivado for Behavioural Simulation and Synthesis. The circuit diagrams, timing diagrams and schematic diagrams were also created using Vivado itself. Due to the niche topic and nature of the project, there were not many freely available testing and debugging tools available so Python scripts, C, and Assembly programs were custom built to test and debug the project.

5.2.1 Python Script

Python Script to generate custom network packets and Manchester encode them were created. The script to parse output copied from the intermediate data registers and output buffers of the modules was also used to debug the project. Refer to source code C.7.

5.2.2 Free Standing OS

A freestanding OS was created to test the practical maximum throughput that can be produced using a best case software solution. Existing OS' proved inadequate for this purpose due to the large number of background processes running at any time and the variability caused by it in the microscopic measurements that are required for benchmarking IPRM. Refer to source code C.1 and C.2.

5.3 Timing Diagram

The timing diagram shown in figure 5.1 illustrates the data flow between each component at every time step.

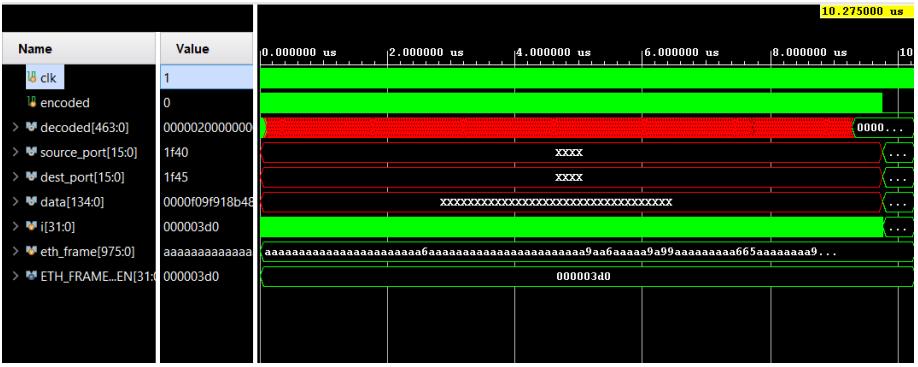

Figure 5.1: Timing Diagram

5.4 Circuit Diagram

The figure 5.2 shows the circuit of the decoder module and figure 5.3 shows the circuit of packet parser.

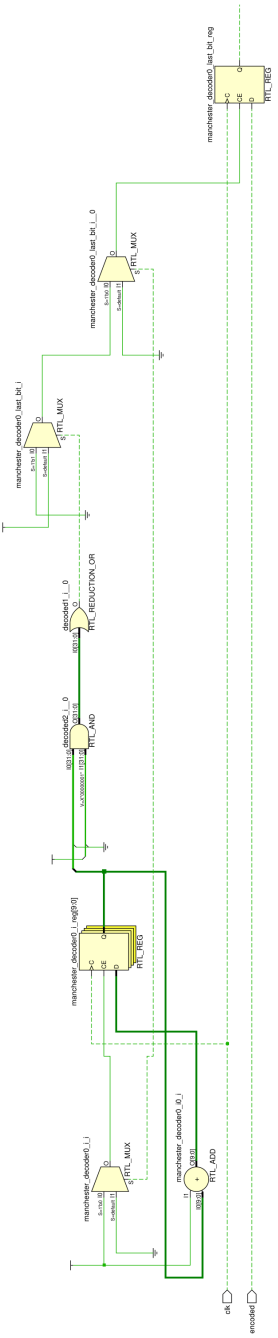

Figure 5.2: Schematic Diagram Decoder

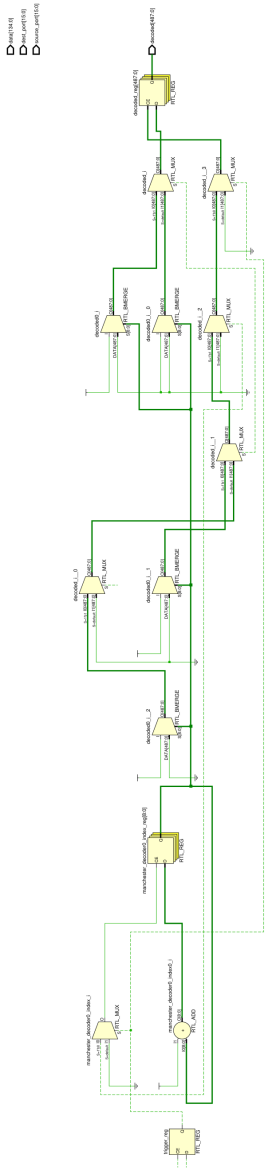

Figure 5.3: Schematic Diagram Parser

Chapter 6

Result

The IPRM was evaluated by parsing sample packets generated using the Python script. The module received input as a bit string from the test bench and produced output as decoded byte string. The byte String was then parsed by the Python script to verify its integrity.

6.1 Sample Packets

The working of IPRM is analysed using sample packets generated with the Python script. The script generates simple and complex packets. Here simple packets are generated by encoding packets with ASCII data only, were as complex packets can contain Unicode characters as well.

A packet with simple text content sent to port 8000 figure 6.1

```
Manchester 10101010101010101010 ... Truncated
Manchester Length 976
Ethernet Header
00.00.00.00.00.02 00.00.00.00.00.01
08.00 ff.04.0e.30

IP Header
45.00.00.2b 00.00.40.00 40.11.b7 6c.c0.a8.01.02
6f.20 57.6f 72.6c 64.20
+---------------+----------+----------+----------+
| Byte Index    | 0        | 1        | 2        |
+===============+==========+==========+==========+
| Bits          | 00000000 | 00000000 | 00000000 |
+---------------+----------+----------+----------+
| Hex           |          |          |          | .... Truncated
+---------------+----------+----------+----------+
Length of data: 15

Unpacked packet:
{'src_mac': '00:00:00:00:00:02',
 'dst_mac': '00:00:00:00:00:01',
 'src_ip': '192.168.1.2',
 'dst_ip': '192.168.1.3',
 'src_port': 8000,
 'dst_port': 8001,
 'data': b'Hello World !!!'}
```

Figure 6.1: Simple Packet Received

```
Manchester 10101010101010101010 ... Truncated
Manchester Length 976
Ethernet Header
00.00.00.00.00.02 00.00.00.00.00.01
08.00 ff.04.0e.30

IP Header
45.00.00.2b 00.00.40.00 40.11.b7 6c.c0.a8.01.02
6f.20 57.6f 72.6c 64.20
+---------------+----------+----------+----------+
| Byte Index    | 0        | 1        | 2        |
+===============+==========+==========+==========+
| Bits          | 00000000 | 00000000 | 00000000 |
+---------------+----------+----------+----------+
| Hex           |          |          |          |
+---------------+----------+----------+----------+ .... Truncated
Length of data: 15

Unpacked packet:
{'src_mac': '00:00:00:00:00:02',
 'dst_mac': '00:00:00:00:00:01',
 'src_ip': '192.168.1.2',
 'dst_ip': '192.168.1.3',
 'src_port': 8000,
 'dst_port': 8001,
 'data': b'Hello World !!!'}
```

Figure 6.2: Simple Packet Output

The output produced figure 6.2

```
Manchester 10101010101010101010 ... Truncated
Manchester Length 976
Ethernet Header
00.00.00.00.00.02 00.00.00.00.00.01
08.00 89.db.11.c7

IP Header
45.00.00.2b 00.00.40.00 40.11.b7 6c.c0.a8.01.02
48.65 6c.6c 6f.20 57.6f
+--------------+----------+----------+----------+
| Byte Index   | 0        | 1        | 2        |
+==============+==========+==========+==========+
| Bits         | 00000000 | 00000000 | 00000000 |
+--------------+----------+----------+----------+
| Hex          |          |          |          |
+--------------+----------+----------+----------+ .... Truncated
Length of data: 15

Unpacked packet:
{'src_mac': '00:00:00:00:00:02',
 'dst_mac': '00:00:00:00:00:01',
 'src_ip': '192.168.1.2',
 'dst_ip': '192.168.1.3',
 'src_port': 8000,
 'dst_port': 8001,
 'data': b'\xf0\x9f\x91\x8bHello World'}
```

Figure 6.3: Complex Packet Received

A packet with complex text content (text containing ASCII and Unicode characters) sent to port 8000 figure 6.3

The output produced figure 6.4

```
Manchester 10101010101010101010 ... Truncated
Manchester Length 976
Ethernet Header
00.00.00.00.00.02 00.00.00.00.00.01
08.00 89.db.11.c7

IP Header
45.00.00.2b 00.00.40.00 40.11.b7 6c.c0.a8.01.02
48.65 6c.6c 6f.20 57.6f
+-------------+----------+----------+----------+
| Byte Index  | 0        | 1        | 2        |
+=============+==========+==========+==========+
| Bits        | 00000000 | 00000000 | 00000000 |
+-------------+----------+----------+----------+
| Hex         |          |          |          |
+-------------+----------+----------+----------+ .... Truncated
Length of data: 15

Unpacked packet:
{'src_mac': '00:00:00:00:00:02',
 'dst_mac': '00:00:00:00:00:01',
 'src_ip': '192.168.1.2',
 'dst_ip': '192.168.1.3',
 'src_port': 8000,
 'dst_port': 8005,
 'data': b'\xf0\x9f\x91\x8bHello World'}
```

Figure 6.4: Complex Packet Output

6.1.1 Observation

From the output produced by IPRM, the working is clear. When a packet contains simple text content (only ASCII) the module silently forwards the packet whereas when a packet with complex content (Unicode characters) is encountered the destination port is adjusted. Here the destination port is calculated by the formula,

$$offset = number\_of\_unicode\_characters \tag{6.1}$$

$$destination\_port = incomming\_port + offset \tag{6.2}$$

This formula can be adjusted as seen fit to produce differing behaviour.

Chapter 7

Conclusion

In conclusion, the project demonstrates the potential of in network computing to significantly improve network performance by reducing the load on central processing units (CPUs) and facilitating efficient data processing. The project's In Hardware Proxy and Rewrite Module (IPRM) is a new hardware accelerator device that can be integrated directly into the NIC, allowing it to intercept and modify network packets in real time.

The test results of this project show the practical real world implications of the technology. Even though the project has achieved results surpassing state of the art in packet proxying, a study has to be conducted comparing the project's solution to proprietary hardware modules manufactured by companies like CISCO and NVIDIA. The major drawback identified in the project's method is the huge upfront investment required to fabricate the hardware. This can be reduced by using modern wafer fabrication techniques currently used in the manufacture of high performance CPUs and GPUs.

References

[1] Theophilus A. Benson. 2019. In-Network Compute: Considered Armed and Dangerous. In Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3317550.3321436

[2] J. Postel . ISI (28 August 1980) "User Datagram Protocol" https://www.ietf.org/rfc/rfc768.txt

[3] Postel, J., "Transmission Control Protocol," RFC 761, USC/Information Sciences Institute, January 1980. https://datatracker.ietf.org/doc/html/rfc761

[4] Postel, J., "Internet Protocol," RFC 760, USC/Information Sciences Institute, January 1980.

[5] Dalal, Y. and C. Sunshine, "Connection Management in Transport Protocols", Computer Networks, Vol. 2, No. 6, pp. 454-473, December 1978.

[6] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. 2008. NOX: towards an operating system for networks. SIGCOMM Comput. Commun. Rev. 38, 3 (July 2008), 105–110. https://doi.org/10.1145/1384609.1384625

[7] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. 2005. Design and implementation of a routing control platform. In Proceedings of the 2nd

conference on Symposium on Networked Systems Design and Implementation - Volume 2 (NSDI'05). USENIX Association, USA, 15–28

[8] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. 2010. Internet inter-domain traffic. In Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM '10). Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/1851182.1851194

[9] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. In Proceedings of the ACM SIGCOMM 2008 conference on Data communication (SIGCOMM '08). Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1402958.1402967

[10] Muhammad Bilal Anwer, Murtaza Motiwala, Mukarram bin Tariq, and Nick Feamster. 2010. SwitchBlade: a platform for rapid deployment of network protocols on programmable hardware. In Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM '10). Association for Computing Machinery, New York, NY, USA, 183–194. https://doi.org/10.1145/1851182.1851206

[11] Open Networking Foundation. "Software-Defined Networking (SDN) - The New Norm for Networks." Available at: https://www.opennetworking.org/sdn-definition/

[12] ONOS Project. "ONOS - Open Network Operating System." Available at: https://onosproject.org/

[13] ONOS Project. "ONOS - Open Network Operating System." Available at: https://onosproject.org/

[14] Arm Community blogs Arm based smartNIC,https://community.arm.com/cfs-filesystemfile/__key/communityserver-components-secureimagefileviewer/communityserver-blogs-components-weblogfiles-00-00-00-21-42/2626.SmartNIC.png_2D00_1600x900.png_2D00_900x506x2.png?_=637093221745268078/

[15] Stefan Schmidt, Manchester Encoding Both Conventions, Wikipedia, Wikimedia Commons, https://en.wikipedia.org/wiki/File:Manchester_encoding_both_conventions.svg

[16] ACN Newswire, Programmable switch-ASICs, https://blog.cloudflare.com/content/images/2020/11/MPC4E-3D-32XGE.jpg

[17] Xilinx Artix-7, HuMANDATA LTD. - Intel (Altera) and Xilinx FPGA Boards manufacturer in Japan,https://www.hdl.co.jp/XCM-023W/top.800.jpg

[18] Michel Bakni, IPv4 Packet, Wikimedia Commons, https://commons.wikimedia.org/wiki/File:IPv4_Packet-en.svg

Appendix A

LLM (Large Language Model)

Large language models are a groundbreaking development in the field of artificial intelligence that has revolutionized natural language processing tasks. These models are deep learning based algorithms capable of processing and generating human like text. They have attracted considerable attention and excitement due to their ability to handle complex language tasks and their potential applications in various domains. This appendix provides a brief overview of large language models, their characteristics, applications, benefits, challenges, and ethical considerations.

1. Characteristics of Large Language Models

- **Architecture:** Large language models are typically based on transformer architectures, which allow them to handle long range dependencies and process vast amounts of data efficiently.

- **Parameters:** These models contain millions (or even billions) of trainable parameters, which contribute to their impressive performance on various language tasks.

- **Pre training and Fine tuning:** Large language models are usually trained in two steps: pre training on a large corpus of text to learn language representations, followed by fine tuning on specific downstream tasks to adapt to particular applications.

2. **Applications of Large Language Models**

   - **Natural Language Understanding (NLU):** Large language models excel at tasks like sentiment analysis, question answering, named entity recognition, and language translation.
   - **Natural Language Generation (NLG):** They can generate coherent and contextually relevant text, making them useful for chatbots, text summarization, and creative writing.
   - **Language Representation Learning:** Large language models learn rich and contextually aware embeddings that benefit various other machine learning tasks.

3. **Benefits of Large Language Models**

   - **State of the Art Performance:** Large language models have achieved state of the art results on a wide range of natural language processing benchmarks.
   - **Generalization:** These models can be fine tuned on various tasks with relatively small amounts of data, enabling better generalization to different applications.
   - **Accessibility:** The availability of pretrained models has lowered the barrier to entry for NLP tasks, allowing developers and researchers to build applications without extensive language expertise.

4. **Challenges and Limitations**

   - **Computation and Resource Intensive:** Training and running large language models require substantial computational resources,

making them inaccessible to some researchers and organizations.

- **Data Bias and Fairness:** Large language models can inherit biases present in the training data, leading to biased or unfair outputs in certain contexts.
- **Ethical Concerns:** The generation of highly realistic text raises concerns about potential misuse, such as generating fake news or malicious content.

5. Ethical Considerations

- **Bias Mitigation:** Researchers and developers must proactively work to identify and mitigate biases in large language models to ensure equitable and fair outputs.
- **Transparency and Explainability:** Efforts should be made to make the decision making process of large language models more transparent and interpretable to foster trust with users.
- **Responsible Use:** Guidelines and policies should be established to govern the ethical use of large language models and prevent malicious applications.

Large language models have become a fundamental technology in natural language processing, enabling significant advancements across various language related tasks. Their potential benefits are vast, but they also come with challenges related to ethics, bias, and resource requirements. By addressing these challenges responsibly, large language models can continue to contribute positively to the advancement of AI driven natural language understanding and generation applications.

Appendix B

ASIC (Application Specific Integrated Circuit)

An Application Specific Integrated Circuit (ASIC) is a specialized integrated circuit designed to perform specific tasks or functions tailored to a particular application. Unlike general purpose microprocessors, ASICs are customized to optimize performance, power efficiency, and area utilization for specific applications. This appendix provides an overview of ASICs, their characteristics, applications, benefits, challenges, and considerations.

1. Characteristics of ASICs

- **Custom Design:** ASICs are custom designed to meet the precise requirements of a specific application or task, leading to optimized performance and efficiency.

- **Functionality:** They are optimized to execute a particular function or set of functions, reducing unnecessary overhead and complexity.

- **Manufacturing Process:** ASICs are fabricated using advanced semiconductor manufacturing processes to achieve high performance and reliability.

- **Non Programmable:** Unlike programmable devices like FPGAs, ASICs are non programmable and cannot be reconfigured once manufactured.

2. Applications of ASICs

ASICs find applications in various industries and domains, including:

- **Telecommunications:** ASICs are used in network processors, modem chips, and wireless communication devices for efficient data processing.
- **Consumer Electronics:** ASICs power specialized features in smartphones, digital cameras, smart TVs, and gaming consoles.
- **Automotive:** ASICs are employed in automotive electronics for functions like engine control, safety systems, and infotainment.
- **Industrial Automation:** ASICs are used in robotics, factory automation, and control systems to enhance performance and reduce costs.
- **Cryptocurrency Mining:** ASICs are specifically designed for mining cryptocurrencies, such as Bitcoin and Ethereum, to perform complex hashing operations efficiently.

3. Benefits of ASICs

- **High Performance:** ASICs are optimized for specific tasks, providing significantly higher performance compared to general purpose processors.
- **Power Efficiency:** Due to their custom design, ASICs consume less power per operation, making them suitable for power constrained devices.

- **Area Efficiency:** ASICs can be designed to have a smaller footprint, making them space efficient for integration into compact devices.
- **Cost Effective at Scale:** For high volume applications, ASICs can be cost effective, as the manufacturing cost per unit decreases with larger production runs.

4. Challenges and Considerations

- **Development Cost:** ASIC design requires significant upfront investment in terms of time, engineering expertise, and tooling.
- **Nonreconfigurable:** Once manufactured, ASICs cannot be changed or updated, making them unsuitable for applications that require frequent updates or changes.
- **Time to Market:** ASIC development can take months to years, which may affect the time to market for certain products.
- **Niche Applications:** ASICs are most suitable for high volume, specialized applications, and may not be practical for low volume or rapidly evolving applications.

ASICs offer tailored solutions for specific applications, providing high performance, power efficiency, and area optimization. They find applications in a wide range of industries, but their development requires careful consideration of cost, time, and market requirements. When appropriately designed and deployed, ASICs can significantly enhance the efficiency and functionality of various electronic devices and systems.

Appendix C

Source Code

This project utilises source programs written in Verilog, Python, C, and Assembly. Some of the relevant code snippets are given below. The code snippets may not contain the required import and include statemets. For any one trying to reproduce the project, the complete source code used is published under GPL3 Licence at `https://github.com/rohittp0/packet-switch`

Free Standing OS

The OS for this project is built using C and Assembly. The kernel is majorly built using Assembly and the network parser is written entirely in C.

```
extern void _init();

void start(BootParams* bootParams)
{
    // Call global constructors
    _init();

    HAL_Initialize();

    log_debug("Main", "Boot device: %x", bootParams->BootDevice);
    log_debug("Main", "Memory region count: %d", bootParams->Memory
        .RegionCount);
    for (int i = 0; i < bootParams->Memory.RegionCount; i++)
    {
        log_debug("Main", "MEM: start=0x%llx length=0x%llx type=%x"
            ,
            bootParams->Memory.Regions[i].Begin,
            bootParams->Memory.Regions[i].Length,
            bootParams->Memory.Regions[i].Type);
    }

    puts("\r\n\r\n\r\n");
    puts("        +=========================+\r\n");
    puts("        ||                       ||\r\n");
    puts("        ||     Not Windows       ||\r\n");
    puts("        ||                       ||\r\n");
    puts("        +=========================+\r\n");
    puts("\r\n\r\n");

    log_debug("Main", "Kernel exit @ Tick %d", getMillis());

    main();
    log_debug("Main", "OS exit @ Tick %d", getMillis());

end:
```

Figure C.1: Kernel

```
int main() {
    uint64_t start = getMicros();
    printf("Current time: %d\n", start);

    bool isUDP = false;

    EthernetFrame ethFrame = parseEthernetFrame(rawPacket);
    IPFrame ipFrame;
    UDPFrame udpFrame;

    if (ethFrame.type == 0x0800) {
        ipFrame = parseIPFrame(rawPacket + 14 * 8);

        if (ipFrame.protocol == 17) {
            udpFrame = parseUDPFrame(rawPacket + 14 * 8 + ipFrame.
                headerLength * 8);
            isUDP = true;
        } else
            printf("Not a UDP packet\n");
    } else
        printf("Not an IP packet\n");

    printf("Time to parse: %d Micro Seconds\n", getMicros() - start
        );

    if (isUDP) {
        printEthernetFrame(&ethFrame);
        printIPFrame(&ipFrame);
        printUDPFrame(&udpFrame);
        printPayload(rawPacket + 42 * 8, (ipFrame.totalLength -
            ipFrame.headerLength * 4 - 8) * 8);
    }

    return 0;
}
```

Figure C.2: OS Main Function

IPRM

IPRM is completely written in Verilog. The code snippets depicted below are of different hardware blocks.

```
always @(posedge clk) begin:
    PASS_THROUGH_MANCHESTER_DECODER0_MANCHESTER_DECODE
    if (trigger) begin
        disable PASS_THROUGH_MANCHESTER_DECODER0_MANCHESTER_DECODE;
    end
    if ((manchester_decoder0_i % 2)) begin
        if ((manchester_decoder0_last_bit && (!encoded))) begin
            decoded[manchester_decoder0_index] <= 1;
        end
        else if (((!manchester_decoder0_last_bit) && encoded))
            begin
            decoded[manchester_decoder0_index] <= 0;
        end
        else if (((!manchester_decoder0_last_bit) && (!encoded)))
            begin
            trigger <= 1;
        end
        else if ((manchester_decoder0_last_bit && encoded)) begin
            $finish;
        end
        manchester_decoder0_index <= (manchester_decoder0_index +
            1);
    end
    else begin
        manchester_decoder0_last_bit <= encoded;
    end
    manchester_decoder0_i <= (manchester_decoder0_i + 1);
end
```

Figure C.3: Manchester Decoder

```
always @(decoded, trigger) begin: PASS_THROUGH_IP_PARSER0_LOGIC
    integer ether_type;
    integer protocol;
    integer i;

    if ((!trigger)) begin
        disable PASS_THROUGH_IP_PARSER0_LOGIC;
    end

    ether_type = decoded[392-1:(392 - 16)];
    protocol = decoded[304-1:(304 - 8)];

    ethertype_reg <= ethertype;
    protocol_reg <= protocol;


    if (((ether_type == 2048) && (protocol == 17))) begin
        // Custom Logic Block
    end
end
```

Figure C.4: Packet Parser

```
module generate_ip_checksum(
    input [159:0] header,
    output reg [15:0] checksum
);
    integer i;
    reg [15:0] word;
    always @* begin
        checksum = 16'h0000;
        for (i = 0; i < 20; i = i + 2) begin
            word = (header[i] << 8) + header[i + 1];
            checksum = checksum + word;
            checksum = (checksum & 16'hffff) + (checksum >> 16);
        end
        checksum = ~checksum & 16'hffff;
    end
endmodule
```

Figure C.5: Checksum Calculator

```
source_port = decoded[216-1:(216 - 16)];
dest_port_original = decoded[200-1:(200 - 16)];
dest_port = decoded[200-1:(200 - 16)];

data = decoded[166-1:32];

for (i = 0; i < 125; i = i + 8) begin
    if (data[i +: 8] > 127) begin
        dest_port = dest_port + 1;
    end
end
```

Figure C.6: Custom Logic Block

Testing and Debugging Scripts

Python scripts were used in testing and debugging the project.

```
def main():
    SIMPLE = "Hello World !!!".encode("utf-8")
    COMPLEX = "  Hello World".encode("utf-8")

    packet = pack(
        src_mac="00:00:00:00:00:01",
        dst_mac="00:00:00:00:00:02",
        src_ip="192.168.1.2",
        dst_ip="192.168.1.3",
        src_port=8000,
        dst_port=8001,
        data=COMPLEX
    )

    manchester = [str(x) for x in manchester_encode(packet, False)]
    print("Manchester", "".join(manchester))
    print("Manchester Length", len(manchester))

    print_packet(packet)
    print(int_to_bits(bytes_to_string(packet)))

    unpacked = unpack(packet)
    print("Length of data: {}".format(len(unpacked["data"])))

    print("\nUnpacked packet:")
    pprint(unpacked, sort_dicts=False)
```

Figure C.7: Python Script