FPGA implementation of a network coding capable switch

by

Daniel Bernard Beaumont de Villiers

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Electronic) in the Faculty of Engineering at Stellenbosch University

Supervisors: Prof. H.A. Engelbrecht
Mr. A. Barnard

March 2020

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / Plagiarism *Declaration*

1 Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2 Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
I agree that plagiarism is a punishable offence because it constitutes theft.

3 Ek verstaan ook dat direkte vertalings plagiaat is.
I also understand that direct translations are plagiarism.

4 Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism.

5 Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Abstract

FPGA implementation of a network coding capable switch

D.B.B. de Villiers

Department of Electrical and Electronic Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MEng (E&E)

March 2020

The amount of internet connected devices is expected to increase dramatically in the near future. This is especially due to the widespread use of Internet of Things (IoT) devices. The Fifth-Generation (5G) of cellular network technologies aims to facilitate in the rapid expansion of IoT devices by providing an increased data rate, higher throughput, device capacity and connection reliability. In order for 5G to be fully integrated into existing telecommunication system, many new technologies are being developed.

Two technologies to help make 5G a reality are network coding and Software Defined Networking (SDN). Network coding is an alternative approach to traditional packet forwarding. Traditional packet-based networks use a "store-and-forward" approach, where intermediate nodes relay or replicate incoming information. Network coding provides an additional step and performs coding on the incoming data, known as "compute-and-forward".

SDN is another widely adopted network technology required for 5G. SDN segregates the traditional decentralized networking approach, into control and data processes. A software component is installed on each data forwarding device as the *dataplane*. The dataplane is the fast component of the network and is where all packet processing is conducted. The dataplane devices are controlled by centralised controller devices. The controllers have a "birds-eye-view" of the entire network and can therefore make more informed network processing decisions, compared to traditional networking.

Multiple software implementations of network coding have been developed and implemented. Network coding has been integrated into SDN in emulated and software environments. There exist many hardware devices that support SDN protocols such as OpenFlow.

However, there are no commercially available network hardware devices that support network coding. Researchers in the field of computer networking have to modify existing devices to include network coding functions. Network hardware devices are often proprietary and therefore it is difficult to modify existing devices to add custom features and functions, such as network coding.

This thesis solves these problems by implementing a network coding capable switch in both a software and hardware based environment. The software based network coding functions are created as Virtual Network Functions (VNFs) that are deployed in an SDN environment as required. The hardware based network coding functionality is implemented using a Field Programmable Gate Array (FPGA) device. Both software and hardware implementations are integrated together using the OpenFlow based SDN bridge, Open vSwitch (OvS). The overall platform is designed to run on a general purpose PC and allows network coding to be evaluated in both physical and virtual network environments, with physical and Virtual Machine (VM) hosts.

The network coding implementations are evaluated using a real packet based network. The VNF based network encoder and decoder achieve a coding throughput of 164.67 and 87.99 Mbps respectively. The FPGA based network encoder and decoder are able to achieve a coding throughput of 223.16 and 496.40 Mbps respectively, providing a speedup of 1.36 and 5.71 over the VNF based implementations. The FPGA logic is run using a single PCIe lane and 50Mhz clock frequency. Taking full advantage of the FPGA device resource utilization, all four possible PCIe lanes and the maximum clock frequencies, the encoder and decoder functions could be implemented to achieve a coding throughput of 1.5 and 2.96 Gbps respectively.

The thesis demonstrates that FPGA based network coding is feasible and provides a significant performance increase over software based implementations. The performance however is reduced dramatically when integrated with a real packet based network. Future work should focus on optimising the integration between OvS and the network coding functions. This would hopefully alleviate any potential integration bottlenecks.

Uittreksel

FPGA implementering van 'n netwerk kodering bekwame skakelaar

("FPGA implementation of a network coding capable switch")

D.B.B. de Villiers

Departement Elektriese en Elektroniese Ingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MIng (E&E)

Maart 2020

Die hoeveelheid internetverbindende toestelle sal na verwagting in die nabye toekoms dramaties toeneem. Dit is veral te wyte aan die wydverspreide gebruik van Internet van Items (IvI) toestelle. Die vyfde generasie (5G) van sellulêre netwerktegnologieë is daarop gemik om die vinnige uitbreiding van IvI toestelle te vergemaklik deur 'n verhoogde datatempo, hoër deurvoer, toestelkapasiteit en verbindingsbetroubaarheid te bied. Ten einde 5G volledig in die bestaande telekommunikasiestelsel te laat integreer, word baie nuwe tegnologieë ontwikkel.

Twee tegnologieë om 5G 'n werklikheid te maak, is netwerkkodering en Sagteware Gedefinieerde n etwerk (SGN). N etwerkkodering i s ' n alternatiewe benadering tot tradisionele pakketversending. Tradisionele pakketgebaseerde netwerke gebruik 'n "winkel-en-vorentoe" benadering, waar tussenknope die inkomende inligting weergee of herhaal. Netwerkkodering bied 'n ekstra stap en voer kodering uit op die inkomende data, bekend as "bereken-en-vorentoe".

SGN is nog 'n algemene gebruikte netwerktegnologie wat benodig word vir 5G. SGN skei die tradisionele gedesentraliseerde netwerkbenadering in beheer en dataprosesse. 'n Sagteware komponent word op die data deurstuurtoestel geïnstalleer as die "dataplane". Die dataplane is die vinnige komponent van die netwerk en dit is waar al die pakkieverwerkings gedoen word. Die dataplane toestelle word beheer deur gesentraliseerde beheertoestelle. Die beheerders het 'n "voël-oog"" van die hele netwerk en kan gevolglik meer ingeligte verwerkingsbesluite neem in vergelyking met tradisionele netwerking.

Verskeie sagtewareimplementering van netwerkkodering is ontwikkel en geïmplementeer. Netwerkkodering is in die geëmuleerde en sagtewareomgewings in SGN geïntegreer. Daar bestaan baie hardeware toestelle wat SGN protokolle ondersteun, soos OpenFlow.

Daar is egter geen kommersiële beskikbare hardeware toestelle wat netwerkkodering ondersteun nie. Navorsers op die gebied van rekenaarnetwerke moet bestaande toestelle verander om netwerkkoderingsfunksies in te sluit. Netwerkhardeware toestelle is dikwels ontoegangklik en daarom is dit moeilik om bestaande toestelle te verander om gespesialiseerde funksies, soos netwerkkodering, by te voeg.

Hierdie tesis los hierdie probleme op deur 'n skakelaar met 'n netwerkkodering in 'n sagteware en hardeware omgewing te implementeer. Die sagteware gebaseerde netwerkkoderingsfunksies word geskep as Virtuele Netwerk Funksies (VNF's) wat na behoefte in 'n SGN omgewing ontplooi word. Die hardeware gebaseerde netwerk kodering funksionaliteit word geïmplementeer met behulp van 'n FPGA toestel. Beide sagteware en hardeware implementasies word saam geïntegreer met behulp van die OpenFlow gebaseerde SGN brug, Open vSwitch (OvS). Die algehele platform is ontwerp om op 'n algemene rekenaar te funksioneer en kan die netwerkkodering in beide fisiese en virtuele netwerkomgewings evalueer, met fisiese en Virtuele Masjiene (VM).

Die implementering van die netwerkkodering word geëvalueer met behulp van 'n ware pakket gebaseerde netwerk. Die VNF gebaseerde netwerkkodeerder en dekodeerder het 'n kodering van onderskeidelik 164.67 en 87.99 Mbps. Die FPGA gebaseerde netwerkkodeerder en dekodeerder kan 'n kodering van 223.16 en 496.40 Mbps onderskeidelik behaal, met 'n versnelling van 1.36 en 5.71 oor die VNF gebaseerde implementasies. Die FPGA logika word uitgevoer met behulp van 'n enkele PCIe baan en 50 MHz klokfrekwensie. Met die volle benutting van die FPGA hulpbronbenutting, al vier moontlike PCIe bane en die maksimum klokfrekwensies, kan die kodeerder en dekodeerfunksies geïmplementeer word om 'n kodering van onderskeidelik 1.5 en 2.96 Gbps te verkry.

Die tesis demonstreer dat FPGA gebaseerde netwerkkodering uitvoerbaar is en dat dit 'n beduidende toename in prestasie lewer ten opsigte van sagteware gebaseerde implementerings. Die werkverrigting word egter dramaties verminder as dit geïntegreer is in 'n werklike pakketgebaseerde netwerk. Toekomstige werk moet fokus op die optimalisering van die integrasie tussen OvS en die netwerkkoderingsfunksies. Dit sal hopelik die moontlike knelpunte vir integrasie verlig.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations:

My supervisors, Prof. H.A Engelbrecht and Mr. A. Barnard, for their continuous support, encouragement, optimism and education throughout this thesis, as well as providing me with the necessary advice and feedback to complete successfully. I am privileged to have had the opportunity to work with and be mentored by both of you.

To my mother, Yolanda de Villiers for her continuous encouragement, and allowing me the opportunities to achieve my passions and goals. Without your support none of this would have been possible.

To my girlfriend, Christina Louw for her divine sense of humour and love. Thank you for standing by me though all the late nights and for always supporting me.

To my flatmates and friends in the MIH Media Lab, for all the good memories and encouragement throughout the last two years.

To the Department of Electrical and Electronic Engineering for providing me with a bursary for the past two years, as well as for funding the FPGA OpenVINO starter kits.

Dedications

This thesis is dedicated to Christina.

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

4G	Fourth Generation Cellular Network Technology
5G	Fifth Generation Cellular Network Technology
ACK	Acknowledgement
ALM	Adaptive Logic Module
ALU	Arithmetic Logic Units
API	Application Programming Interface
ARP	Address Resolution Protocol
BAR	Base Address Register
CPU	Central Processing Unit
CRC	Cyclic Redundancy Checksum
CUDA	Computer Unified Device Architecture
DLR	Dedicated Logic Registers
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
FEC	Forward-Error Correction
FIFO	First-in, first-out
FOSS	Free and Open Source Software
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit
HDL	Hardware Descriptive Language
IGMP	Internet Group Management Protocol
IP Core	Intellectual Property Core
IP	Internet Protocol
IaaS	Infrastructure as a Service
IoT	Internet of Things
KVM	Kernal-based Virtual Machine
LAN	Local Area Network

LCM	Linear Code Multicast
LFSR	Linear-Feedback Shift Register
MAC	Media Address Control
MPLS	Multiprotocol Label Switching
MTU	Maximum Transmission Unit
NFV	Network Function Virtualization
NIC	Network Interface Card
OSI	Open Systems Interconnection
OS	Operating System
OvS	Open Virtual Switch
P2P	Peer-to-Peer
PCIe	Peripheral Component Interconnect Express
PC	Personal Computer
PDM	Poll Mode Driver
PHY	Physical host
PIO	Parallel Input/Output
QEMU	Quick Emulator
RAM	Random Access Memory
RLNC	Random Linear Network Coding
RTT	Round Time Trip
SDK	Software Development Kit
SDN	Software Defined Networking
SR-IOV	Single-Root Input/Output Virtualization
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol
VHDL	Very-High-Speed Integrated Circuit Hardware Description Language
VM	Virtual Machine
VNF	Virtual Network Function
WMN	Wireless Mesh Networks
XOR	Exclusive-or

Variables

m	Galois field size
h	Generation size
G	Acyclic graph

V	Graph vertices
E	Graph edges
S	Senders
R	Receivers
N	Data packet length
$\boldsymbol{x_i}$	Source vector
x_i	Source symbol
$\boldsymbol{X}$	Source matrix
e	Single graph edge
$g_i(e)$	Coding coefficient symbol
$\boldsymbol{g(e)}$	Global coding vector
G_t	Coefficient matrix
$y(e)$	Encoded output symbol
$\boldsymbol{y(e_i)}$	Encoded output vector
$\boldsymbol{y(e)}$	Encoded output matrix
I_h	h x h Identity matrix
l	Number of 32-bit data segments
k	Coefficient data length for each packet
t_{coding}	Overall coding runtime
T_{coding}	Overall coding throughput
$T_{network}$	Network throughput
$t_{latency}$	Network latency
PDV	Network packet delay variation

Units

Mbps	Megabits per second
KB	Kilobytes
MB	Megabytes
GB	Gigabytes
Mhz	Megahertz
Ghz	Gigahertz
μs	Microseconds

Chapter 1

Introduction

1.1 Overview

The objective of this thesis is to design and implement a network switch with the capability of performing network coding, in a real packet-based network. The switch requires the ability to offload packet data to both software and hardware-based packet processors. We begin the discussion by outlining the various technologies explored before moving onto addressing the problems and need for the network switch. Once the problems are understood, we derive objectives to address the problems. We look at past work that is related to the thesis and list the contributions made by our work, in comparison.

1.2 Background

The amount of internet connected devices is expected to increase dramatically in the near future. This is especially due to the widespread use of Internet of Things (IoT) devices. The Fifth-Generation (5G) of cellular network technology aims to provide an increased data rate, higher throughput, device capacity and connection reliability. Specifically, 5G will enable the rapid expansion of IoT technologies. Unlike previous cellular network technology generations, 5G is consider to be a "revolution" rather than a generational upgrade as with Fourth-Generation (4G). There are a number of technological challenges that exist in order for 5G to be fully integrated into existing telecommunication systems. In response, many technologies are being engineered [1]. One of these technologies is network coding.

Network coding is an alternative approach to traditional packet forwarding. Traditional packet-based networks work on a "store-and-forward" basis, where intermediate nodes relay or replicate incoming information. Network coding provides an additional step and performs coding on the incoming data, as a "compute-and-forward" paradigm. The idea that information should not only be stored at intermediate nodes, but coding could also be performed originates

from Ashlswede et al. in [2]. Network coding has shown to improve throughput and redundancy in various applications, such as multicast streaming [3].

The principle example of network coding is explained using the *butterfly network*, taken from Fragouli and Soljanin in [4]. The butterfly network, in Fig. 1.1 consists of two sources S_1 and S_2, and two receivers R_1 and R_2. If R_1 requests data from both S_1 and S_2, then it will be able to receive from both. This is also the case when R_2 is receiving data from both sources. When both sources send data to both receivers simultaneously, then there is congestion at the edge CE. More specifically, consider if S_1 wants to send bit x_1 to R_2 simultaneously, while S_2 sends bit x_2 to R_1. In traditional information flow only one bit can be sent at a time. The other bit waits to be transmitted. In network coding however, the intermediate node C is able to combine both bits into a third packet $x_1 + x_2$. At the receiving end, each receiver can solve a simple system of linear equations and obtain x_1 and x_2.

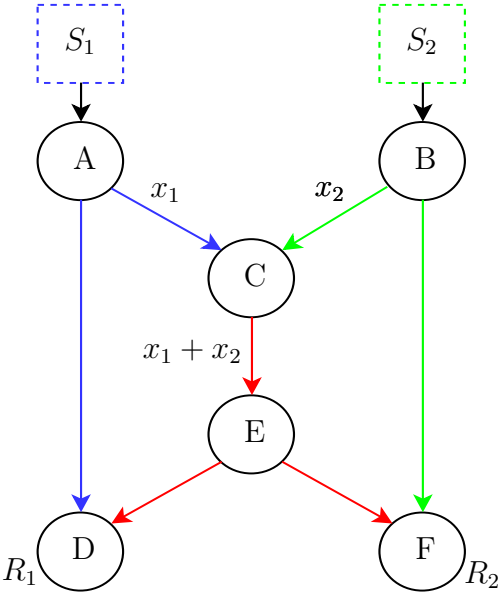

Figure 1.1: Butterfly network topology of two sources S_1 and S_2, and two receivers R_1 and R_2. Network coding advantage is obtained by combining bits x_1 and x_2 at the intermediate node C.

The first practical implementation of network coding is shown by Chou et al. [5]. A packet format is presented that enables network coding to be used in practical packet-based networks. Chou et al. also introduces the idea of grouping packets into a *generation*. The coding operations are then performed on the generation itself.

The most adopted form of network coding used in practical implementations is Random Linear Network Coding (RLNC). In RLNC, intermediate

nodes select random coding coefficients independently. This allows the receiver to decode after obtaining sufficient independent linear combinations.

Examples of RLNC software libraries that can be used in practise are: *NCUtils* [6], *Random Linear Network Coding Library* [7] and *Kodo* [8]. The Kodo project by Steinwurf is the most mature and widely adopted in literature. Kodo provides a large variety of network coding algorithms and functions. Kodo's functions are written in C++ and provide wrappers for other programming languages such as C and Python. While network coding libraries such as Kodo exist, there is currently no commercial network switch that implements network coding.

According to [1], another widely adopted technology required for 5G is Software Defined Networking (SDN). Traditional networking is not flexible at a large scale. Network devices are difficult to configure and manage on an individual basis. This becomes problematic when network topologies need to be upgraded or modified. It is complicated to service these networks and as a result downtime can occur.

SDN aims to solve the difficulties of traditional networking by segregating networks into control and data processes [9]. A software component is installed on each data forwarding device as the *dataplane*. The dataplane is the fast component of the network and is where all the packet processing is conducted. Control of the network is centralised using one or more controller devices. The controllers have a "birds-eye-view" of the entire network. Network control applications are run on the controllers to characterise how the data is processed in the dataplane. The controller has direct control over the dataplane through an Application Programming Interface (API). OpenFlow is currently the most widely adopted API for SDN.

An OpenFlow device uses *flow tables* that dictate the actions to perform on each incoming packet [10]. Unknown packets received by a dataplane device are sent to the controller. The controller determines the appropriate actions and updates the flow table on the dataplane device. This creates a rule and future packets do not need to be sent to the controller. This provides a more dynamic approach to network management. Projects such as OpenFlow enable SDN to provide easier deployment and integration of networking technologies. SDN can be used to integrate network coding operations within a network by adding the appropriate OpenFlow flows to an SDN compatible network switch.

SDN integrates well with Network Function Virtualization (NFV). Following along the direction of computer servers, computer network functionality is also being virtualized. NFV or Virtual Network Functions (VNFs) are a recently introduced trend in computer network communication systems [1]. NFV tries to move away from using dedicated networking hardware. VNFs are usually created within Virtual Machines (VMs) that run in a hypervisor environment. Therefore a server with Network Interface Cards (NICs) can be used to deploy multiple network functions. Examples of such network functions are switches, routers, gateways, firewalls and forward-error-correction coders.

A SDN-based network can assign these network functions as required. Network coding can be implemented within a VNF to be deployed in a SDN-based network.

A fast packet processing library that can be used to create VNFs is Intel's Data Plane Development Kit (DPDK) [11]. The Linux operating system provides a great amount of flexibility for packet processing. The problem however, is that there is much overhead built into the default network stack. DPDK aims to solve this by making use of Poll Mode Drivers (PMDs) to allow for network function development in the userspace. PMDs are a type of "kernel bypass" that allow for packet processing without using the default Linux networking stack. DPDK has shown a large amount of performance increases in practical network use. An example of this is the widely adopted integration with Open vSwitch (OvS) [12].

OvS is a open source virtual switch that uses OpenFlow to provide SDN capability [13]. OvS can be installed on many different Operation Systems (OS), including Linux. Therefore, OvS can be run on a general purpose desktop or server computer. Both virtual and physical network space can be utilised by adding networking or processing hardware to the computer. Examples of such hardware are NICs and accelerators. The Peripheral Component Interconnect Express (PCIe) interface is one way of connecting additional hardware to the computer.

PCIe can be used to connect hardware such as an Field Programmable Gate Array (FPGA). FPGAs are Integrated Circuit (IC) devices that can be configured to implement digital logic-based circuits. Therefore, many different packet processing circuits can be developed to run on FPGAs. FPGAs provide the ability to run multiple parallel circuits at once and can provide increased performance over software-based algorithms. Therefore FPGA's can be interfaced through PCIe, with a computer running OvS in order to accelerate the RLNC algorithm.

1.3 Problem statement

Multiple software implementations of network coding have been developed and implemented. Network coding has been integrated into SDN in emulated and software environments. There exist many hardware devices that support SDN protocols such as OpenFlow.

However, there are no commercially available network hardware devices that support network coding. Researchers in the field of computer networking cannot go and buy an "off-the-shelf" network switch in order to evaluate network coding in a practical network. This leads researchers to have to modify existing devices to include network coding functions.

Hardware devices are often proprietary. While this is well justified in the commercial industry, it becomes problematic in research. It is difficult to

modify existing network devices to add custom features and functions, such as network coding.

This thesis solves these two problems by providing an easily modifiable hardware-accelerated network coding switch platform. The switch can be used to evaluate network coding in a real, packet-based network. This allows for further research to be conducted regarding network coding itself. Researchers do not have to focus on the implementation of network coding, but rather the use cases.

1.4 Objectives

The aim of this project is to create a flexible network switch platform, in order to evaluate network coding in a real packet-based network. Network coding functions are implemented in a hardware and software environment. This is achieved through the following objectives:

1. Design and implement a network coding capable switch in a software-based environment. The software implementation serves as a baseline for comparison with the hardware coders. The software implementation also allows for network coding to be used in a virtualization environment. Network coding can be deployed as a VNF and in an Infrastructure as a Service (IaaS) environment.

2. Design and implement a network coding capable switch with the capability of offloading network traffic to an FPGA to assist in packet processing acceleration.

3. Evaluate the runtime, throughput, latency and jitter performance of both the software and hardware network coding switch implementations, in a real packet-based network.

4. Maintain the philosophy of open networking by using techniques of SDN and NFV where possible, to provide a flexible and expandable solution. Following the current trend of modern networking enables this thesis to remain relevant for any future work.

1.5 Research methodology

The design and implementation of a network coding capable switch, should allow for easier configuration and study of network coding algorithms. In addition, the use of hardware-based acceleration should also increase the performance of network coding implementations in practical packet-based networks. These assumptions are validated through the following research methodology:

1. Designing and implementing virtual, software-based network coding functions to establish a baseline set of performance metrics (runtime, throughput, latency and jitter).

2. Designing and implementing FPGA-based, hardware network coding functions to evaluate the feasibility and performance of hardware-based network coding.

3. Designing and implementing suitable methods to interface both software and hardware coding functions together within a real packet-based network.

4. Creating a suitable set of testing procedures to accurately compare both software and hardware-based network coding implementations.

5. Performing the set of testing procedures using both software and hardware-based implementations and evaluating the results in comparison.

6. Outlining the benefits and weaknesses of both software and hardware-based implementations, and deriving suitable conclusions regarding both approaches.

1.6 Related work

Previous work to implement network coding in practical network systems using either SDN or FPGAs are done by [14], [1] and [15]. These works of literature are referenced and expanded upon throughout this thesis.

1.6.1 Kim, et al. Design and evaluation of random linear network coding accelerators on FPGAs

Kim, et al. implements FPGA-based RLNC decoder logic in [14]. Two different decoder designs are presented that take advantage of the parallel computation ability of FPGAs. The designs are synthesized using a Xilinx Virtex 5 FPGA. A maximum throughput of 64.98 Mbps is obtained by the decoder implementation, providing speedup of 13.84 compared to software implementations.

The designs make use of ModelSim to simulate the hardware implementation, and do not implement the decoders in a real packet-based network. The work also only implements a RLNC decoder and not an encoder. This thesis expands on the work by Kim, et al. by providing the design for both an FPGA-based RLNC encoder and decoder as well as integration into a real packet-based network. The decoder implementation is this thesis achieves a coding throughput of 378.86 Mbps, a speedup of 5.83 over the implementation in [14].

1.6.2 Hansen, et al. Network coded software defined networking: Enabling 5G transmission and storage networks

Hansen, et al. [1] emphasizes the importance of network coding with SDN for 5G technologies, by implementing an integration between OpenFlow and Kodo. The article discusses the problem of adding network coding capability to existing routers and switches. A solution is provided by implementing the coding functions on a VM as a VNF. The OvS bridge is used to direct traffic flow to the coder VM as required.

This thesis expands on the work of [1] by integrating hardware-based (FPGA) network coding functions into an SDN and OvS environment, alongside VNF-based network coding functions. The VNFs are implemented using the DPDK packet processing library.

1.6.3 Gabriel, et al. Practical deployment of network coding for real-time applications in 5G networks

The conference paper by the Gabriel, et al. [15] demonstrates a practical deployment of RLNC VNFs within an SDN environment to improve video streaming quality due to congestion based packet loss. The demonstrator presented makes use of off-the-shelf hardware and the OpenStack SDN environment. The RLNC functions are implemented using the Kodo library.

The work focuses on the setup and implementation of the integrated system, and does not provide network performance results. This thesis expands on the work of [15] by implementing the VNFs using the DPDK library and adding FPGA-based network coding functions to the SDN environment. Practical results are given based on the implementation, to provide indication of the strengths and limitations of implementing network coding in a SDN environment.

1.7 Contributions

The following contributions are made through this thesis:

- An FPGA network coding encoder and decoder is developed that can be used within a practical SDN network. This thesis presents FPGA logic for a RLNC encoder which has not yet been shown in literature. The thesis also showcases the design and implementation of integrating FPGA network coding encoder and decoder functions with the OvS OpenFlow SDN switch, and therefore a real packet-based network.
- A VNF-based network encoder and decoder that are implemented using the DPDK software library.

- A holistic research platform to evaluate network coding in a real packet-based network, using both physical and virtual networking spaces. The networking functions are integrated with OvS within an SDN environment.
- A network coding selection scheme to determine when to perform network coding in a real packet-based network.
- An SDN controller program design to implement multicast snooping within an SDN environment.

1.8 Thesis outline

- Chapter 2 provides a literature review to further expand on the concepts covered in this introduction. Previous related work is discussed. The chapter intends to inform the reader of current trends in computer networking, and provide the knowledge required for understanding the rest of the thesis.
- Chapter 3 provides a system overview of the network switch. Due to the various components and interfaces of the system, a system diagram is created. This provides a visual aid in understanding the overall system architecture. Each component is introduced and discussed.
- Chapter 4 discusses the software implementation of network coding functions. The specifics of how DPDK is used to create an encoder and decoder VM-based VNFs are described.
- Chapter 5 discusses the hardware implementation of network coding functions. The details of how the RLNC algorithm is implemented in FPGA hardware are given. The chapter provides information on the methodology, architecture, implementation and evaluation of the coders.
- Chapter 6 describes how the software and hardware implementations are interfaced with the OvS bridge. Details regarding the control plane operations are provided.
- Chapter 7 describes the experimental evaluation of the various components. The software and hardware implementations are tested and compared. The results are discussed and the overall system is evaluated.
- Chapter 8 provide conclusions and recommendations regarding the thesis.

Chapter 2

Literature Review

2.1 Introduction

The previous chapter introduces the concepts of network coding, SDN, NFV and FPGAs. In this chapter, we continue the discussion further by outlining the various sub-problems to address. Specifically, the requirements for a software and hardware platform, and network coding switch functionality are scoped. After outlining the sub-problems, we proceed into the various networking technologies to address the specified sub-problems. The literature review is used to study past work relevant to the design and development of a network coding capable switch.

2.2 Sub-problems to address

To design and develop a network coding switch to meet the thesis objectives from section 1.4, we begin the discussion with the various sub-problems that need to be addressed. This chapter reviews past literature to find a solution to the following:

2.2.1 Software platform for a network coding switch

A software-based network coding implementation is required to serve as a baseline for comparing and implementing hardware-based network coding. The software-based network coding implementation has the following requirements:

1. The software-based implementation must be able to use the RLNC algorithm.
2. The software-based network coding implementation must be SDN compatible and be used in a real packet based network.

3. The software-based network coding implementation must be deployable in a virtualization environment.

2.2.2 Hardware platform for a network coding switch

A suitable hardware platform is required to perform network coding. The hardware-based implementation has the following requirements:

1. The hardware implementation must be able to use the RLNC algorithm.
2. The hardware implementation must be able to integrate with an SDN environment to be used in a real packet-based network. Specifically, the switch must be able to run an SDN dataplane protocol (such as OpenFlow) and be able to communicate and interface with an SDN controller in the controlplane.

2.2.3 Network coding switch functionality

In addition to the software and hardware-based network coding implementations, the network switch also requires the following functionality:

1. The network switch must perform the basic switching function of forwarding networking packets.
2. The network switch must be SDN compatible.
3. The network switch must have the ability to offload network traffic to both the software and hardware-based network coding implementations.
4. The network switch must be able to function in both a physical and virtual network environment.
5. The network switch must be able to perform multicast snooping to be used in multicast streaming applications.

We begin the literature discussion with the traditional approach to network switching within a packet-based network. We then provide more details on network coding before moving onto discussing suitable platforms for the software and hardware-based network coding implementations. We conclude the literature review by evaluating the network coding switch functionality requirements.

2.3 Traditional network switching

The definition of *traditional* networking is not necessarily clear. In fact, neither is the definition of the adjective itself. In the context of this thesis however, traditional networking refers to computer networking concepts that are not part of an SDN ecosystem. More specifically, traditional networking refers to computer network systems where the data and control plane do not have a clear structural boundary. These networks are often, but not necessarily, decentralised.

The concepts of traditional networking form the basis of SDN techniques. Therefore this section is necessary to understand implementations regarding SDN and VNF. The explanations derive from [16], unless cited otherwise. The first concept is to understand how computer networks are structured.

Computer networks are divided into different layers. These layers resemble programming interfaces or libraries. There are many different models describing networking layers. The most popular is provided by the Open Systems Interconnection (OSI) project. The OSI attempts to establish networking standards by providing a seven layer model. A diagram depicting the various OSI layers is shown in Fig. 2.1. The layers of OSI are: physical, data link, network, transport, session, presentation and application.

Layers	Protocols
L7: Application	HTTP, FTP, SSH and DNS
L6: Presentation	ASCII, JPEG and GIF
L5: Session	API's and Sockets
L 4: Transport	TCP and UDP
L3: Network	IPv4, IPv6, ICMP and IGMP
L2: Data link	MAC, LLC and PPP
L1: Physical	WiFi, RJ45, USB and Bluetooth

Figure 2.1: OSI networking layer model with examples of protocols that belong to each layer.

The OSI layer of most importance within this thesis is the data link layer, or the L2 layer. The data link layer is responsible for node-to-node transfer and includes Local Area Network (LAN) communication. LANs are essentially

serial lines used to interface between network hardware. The most common LAN protocol: Ethernet, was originally developed in 1976 and became popular due to its low cost and ease of use.

2.3.1 Ethernet LAN protocol

An Ethernet network groups information bits into buffers of data called packets. Packets allow devices within a network to communicate. Packets often have a preceding sequence of bits called a header. The header provides the necessary information to transfer a packet through a network. The packet data that is required by the application layers is the payload.

In a computer, the Ethernet LAN protocol assigns each network interface a unique six byte hardware address. This address is known as the Media Access Control (MAC) address. Each network interface monitors for arriving packets containing a destination address equal to the device MAC address. If the packet matches, the interface forwards the packet to the CPU to be processed.

The Ethernet packet frame format is shown in Fig. 2.2. The packet traverses through the network from the source address *src_addr* to the destination address *dst_addr*. The Ethernet type *eth_type* field provides information regarding the protocol that is used by the payload. The payload is the actual data that needs to be transmitted by the packet. The payload is not part of the Ethernet header. The Cyclic Redundancy Checksum (CRC) *crc* field is an error-correcting code used to detect any data changes that occurred during transmission.

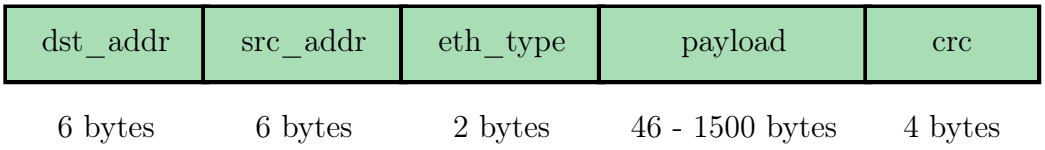

Figure 2.2: Ethernet packet format containing: destination address *dst_addr*, source address *src_addr*, Ethernet type *eth_type*, data payload and cyclic redundancy checksum *crc*.

2.3.2 Ethernet switch

Ethernet networks are connected together using switches. An Ethernet switch is a L2 networking device that functions on the MAC layer. These devices forward packets between multiple Ethernet interfaces within a network.

Ethernet switches forward packets using forwarding or MAC tables. A forwarding table is a lookup table with two primary fields: destination address and next hop output port. When an Ethernet packet is received by the

switch, a lookup is performed based on the destination address. The packet is forwarded to the correct device connected to the next hop port.

To begin forwarding packets, switches first need to establish the correct forwarding table. A learning process is used to populate the forwarding table from empty. The switch learns where each destination is through the use of *fallback-to-flooding*. If a packet is received and the destination is unknown, the packet is forwarded through each output port. The switch *floods* the packet to each port. At the same time, the switch stores the source address of the received packet in the forwarding table. The source address is stored along with the port where the packet arrived, therefore creating an entry in the forwarding table. Future incoming packet destination addresses are compared to the forwarding table. If the address matches, then the switch does not flood the network, but rather forwards it to the correct interface.

2.3.3 Multicast

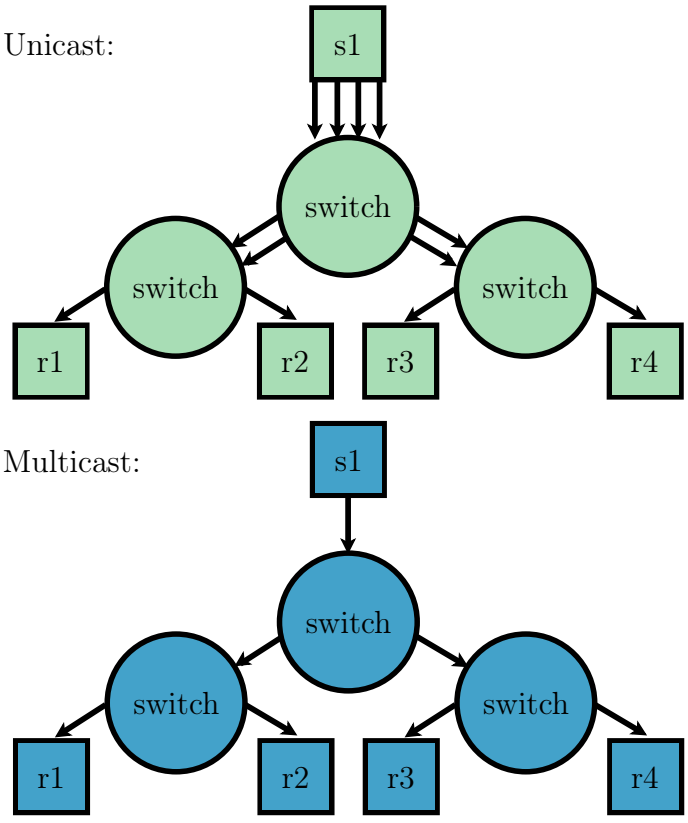

Figure 2.3: Unicast vs multicast streaming packet flows. The advantage of multicast is illustrated showing a reduced number of packet streams.

In an Ethernet network, packets are either transmitted via unicast, broadcast, or multicast. Unicast is traffic sent to a specific host, while broadcast and multicast send traffic to the whole network or a set of hosts respectively. Multicast is of particular interest in video streaming applications, where multiple users request the same data from the server simultaneously.

The advantage of multicast is demonstrated in Fig. 2.3. The example illustrates a source *s1* sending data to four separate receivers *r1*, *r2*, *r3* and *r4*. In a unicast network a separate packet is created with the destination address set to each receiver. Four packets with exactly the same payload, but different headers are transmitted. These packets are then forwarded through the network until they reach the receivers. In the multicast network however, the sender only creates one packet and therefore reduces the overall network traffic.

Multicast reduces network traffic by using group addresses. Multiple users or *subscribers* are able to receive data packets from a single source address within the multicast group. Multicast uses a unique MAC address format to distinguish from unicast Ethernet frames. A value of 1 for the least-significant bit in the first byte of the Ethernet MAC address indicates if the address is multicast.

While multicast can be done on a L2 layer, L3 or IP-based multicast is often preferred. By default, L2 switches cannot determine the location of multicast group members. As a result, switches will broadcast multicast packets instead. To avoid broadcast, switches can use what is known as *snooping*.

A popular protocol for IP-based multicast that enables L2 switch snooping is the Internet Group Management Protocol (IGMP). There are three versions of IGMP, namely IGMPv1, IGMPv2 and IGMPv3. The most widely used is IGMPv2, which is documented in the RFC-2236 standard [17]. The IGMPv2 protocol is used in this thesis and hereby referred to as IGMP.

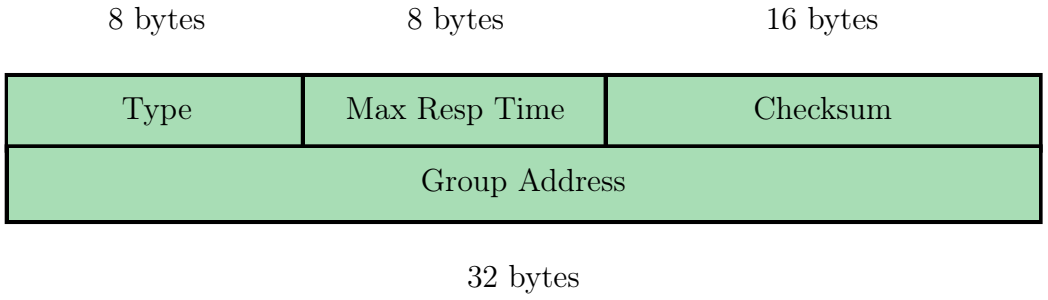

Figure 2.4: IGMPv2 packet format containing: message type, maximum allowed response time to respond to membership query, checksum and multicast group address.

Hosts in a network use IGMP messages to report their membership status to

neighbouring multicast routers. An IGMP packet has the format given in Fig. 2.4. The types of IGMP messages are membership query (0x11), membership report (0x16) and group leave (0x17). The maximum response time *max resp time* field is only used by the membership query and is the maximum allowed time to respond to a query. The multicast group being reported or left is held in the group address field.

An Ethernet switch with IGMP snooping monitors network traffic and inspects each IGMP packet. The switch creates a lookup table and stores a list of respective multicast group members. When a packet arrives, if the multicast group is known to the switch, then the switch forwards the packet to the correct members. The switch therefore, does not need to default to broadcasting and reduces the amount of network traffic.

While the multicast example in Fig. 2.3 only has a single transmitter, throughput is difficult to maintain when multiple multicast streams exist simultaneously. This is where network coding comes in. As introduced by the butterfly network in Fig. 1.1, network coding aims to improve throughput in multicast streaming applications.

2.4 Network coding

Network coding needs to be integrated into an intermediate device or network switch to benefit multicast traffic. To incorporate network coding into an Ethernet switch, an algorithm is required that can work in a real packet-based network.

This section aims to create a set of links from the original idea of network coding to practical coding implementations. A bias is taken towards finding practical parameters to utilise network coding for robust, continuous packet processing. Factors of packet construction, link stability, network topology and codecs are all investigated. The focus is on the practical and application side of network coding, and how it is derived from theory.

2.4.1 Codes to Kodo

The theoretical discussion begins with the *max-flow min-cut* theorem. The theorem states that the maximum information flow rate of a network is limited by the minimum cut. Taken from graph theory, the minimum cut is the sum of the removed edge capacities between the source and sink nodes, to stop flow.

Ahlswede's states a conjecture similar to max-flow min-cut where the source information rate must be less or equal to the max-flow of the sender to receiver [2]. Network configurations consisting of sources sending to multiple sink nodes (i.e. multicast) would require network coding in order to achieve maximum flow capacity. Ahlswede shows that even simple codes are able to achieve max-flow. Alpha codes, which are a type of block code, are used as the original bases

for network coding. Convolutional codes are also shown to be good alternatives to block codes for practical scenarios. While the idea was introduced here, Ahlswede left lots of room for improvement.

One of the first expansions to [2] was conducted by Li, et al. using linear codes for network coding [18]. In Li's approach data is represented as a row vector, and linear transformations are applied at coding nodes. The information leaving the coding nodes is a linear combination of the incoming data. This idea is presented as a linear code multicast or LCM. A physical realization of LCM in a acyclic network is provided. Each individual, intermediate node waits until data has been received on each incoming channel. The node performs network coding and sends the encoded data out on each outgoing channel. Each intermediate node performs coding based on a sequential order.

Use of an LCM is shown to achieve the same individual max-flow capacity presented by Ahlswede. This is an advancement because linear codes are easier to implement in practise. However, due to the approach being based upon synchronisation, it is not suitable for real-time applications. Li leaves room for improvement, and opens the field to many mathematical disciplines.

Koetter and Médard took advantage of [18]'s work in [19]. They provide an algebraic framework to solve network coding scenarios. The main contribution is suitable algebraic conditions to determine network feasibility, and study network capacity when using linear codes. The results of [2] are achieved using only scalar linear algebra over a finite field. This is significant in that it allows the use of powerful algebraic mathematics to solve network coding problems. A technique for finding linear coding coefficients is also provided. Another result presented by [19] is that network coding provides robustness in the context of link failure.

Previous work and that done by [19], uses an overview of the complete network for network coding. On the other hand, Ho, et al. shows that distributed randomised coding is beneficial over routing [20]. The idea is that each intermediate node selects coding coefficients from the finite field given by [19]. The receiving nodes only require the overall linear combination to decode the original information. Random coding maintains the network capacity of [2], while providing robustness. This robustness is achieved by distributing information over the network, allowing for any topology changes or link failures. The probability of successful information flow is shown to be proportional to the finite field size.

Two additional approaches to the work of [19] for determining network feasibility, and if a linear code is valid are given in [21]. This work is done in companion, to help formulate the results of [21]. A tighter bound on required coding field size over [19] is given.

All previous discussion up to this point has been focused on the feasibility of network coding from a mathematical side. The results presented by Chou, et al. provide a practical distributed network coding scheme in [5]. The theoretical work looks at synchronous network flow, where as practical systems

are subject to packet loss, delays and jitter which cause asynchronous flow. The capacities between nodes are not known, along with a less complete network overview. Chou addresses various additional practical problems such as cycle networks, link loss, topology changes and congestion. A packet format is proposed that allows for decentralised network coding operations. The results show that a field size of 2^8 is sufficient in practise, while 2^{16} is more than enough.

The scheme in [5] works as follows. Incoming symbols are packeted into vectors of specific length related to the field size, e.g. 1400 symbols for 2^8. On the outgoing side, each vector is expressed as a linear combination of the incoming vectors. Each packet includes the global coding vector as a header, allowing a receiver to decode the original information via Gaussian elimination. The cost of this approach is in transporting the coding coefficients. The overhead however, is relatively small compared to the overall packet size, especially considering the benefit of decentralisation. This approach allows receivers to decode the original information in a practical system, as well as when randomly chosen coefficients such as those in [20] are used.

In practical systems where packets transverse asynchronously through a network, the packet format by itself becomes problematic. This is because packets from the same source vectors can be received out of order. The decoder does not know which packets should be decoded together. To solve this, [5] introduces the concept of *generations*, in which packets from the same source are grouped by generation with a specific generation size.

Each generation has a specific number which is included in the packet header. Packets received at a node are buffered by generation number. When an outgoing transmission opportunity arises, a random linear combination of all packets in that generation are sent out as a packet. Older generations are removed based on a flushing policy, where packets that arrive belonging to older generations are discarded. A loss of throughput can be experienced from this and is proportional to the delay between the time it takes for the first packet to reach a node over the slowest and fastest paths. This delay is known as delay spread. Increasing the generation size is shown to decrease throughput loss, but increases net throughput due to increased packet header size. Another condition known as interweaving length is shown to have a large effect on decreasing throughput loss.

Interweaving length is the amount of buffers which the original multicast session is distributed. It is suggested that different flushing policies can be used to reduce throughput further. Packets that do not provide new information for generating random packets to be sent out are useless and therefore disregarded. Keeping track of non-innovative packets can also help to reducing bandwidth.

The simulation results performed by Chou show performance gained in terms of received rank, throughput and decoding delay as a function of sending rate, latency, field size, generation size, and interweaving length. The average receiver rank is shown to gain very minimal increase over field sizes of 2^8.

Throughput is described as the product of sending rate and average received rank divided by the generation size. The results show that throughput loss is inversely proportional to generation size and interweaving length. This shows that a larger generation size and longer interweaving length will reduce throughput losses.

After the significant practical contribution of [5], more theoretical study is conducted by some of the same authors from [19], [20] and [21]. This work presented by Ho, et al. in [3] presents a RLNC approach that achieves multicast capacity with almost certainty as the code length increases. The results contribute by expanding those of [20] and show how random codes can be created, as well as transmitted from source to receivers in a distributed network. The approach is able to maintain capacity as well as improve robustness. It is mentioned that network coding is not useful for all network configurations. As a result, simulations are conducted with two scenarios that benefit from network coding.

The scenarios that benefit from network coding are distributed networks, and networks with dynamic connections. Simulation results show that RLNC is able to outperform traditional routing approaches [20]. Overheads are observed while performing coding operations at the encoder and decoder, and with coefficients in the packet headers.

2.4.2 RLNC theory

In order to implement RLNC in a software and hardware-based platform, the RLNC theory needs to be well understood to determine the types of processing operations required. We therefore continue the discussion onto the details of the RLNC algorithm.

The encoder needs to encode the incoming packet data. Incoming, uncoded packets are grouped in *generations* of a *generation size*, h. All of the packets in a generation are from the same source. These incoming packets are multiplied by a set of randomly generated coding coefficients, as RLNC. The work previously done by Chou, et al. provides a foundation for the encoding process [5]. A packet format is proposed that does not require any consideration of graph topology or coding functions. The same notation is used to provide a comparative discussion.

The standard framework for network coding is taken from Ahlswede, et al. and describes a network as an acyclic graph [2],

$$G = (V, E)$$

The set of edges E carry information between the vertices V, from the source senders $S \subseteq V$ to the receivers $R \subseteq V$. The source vector for a single packet, of N symbols, is written as,

$$\boldsymbol{x_i} = [x_{i,1}, x_{i,2}, ..., x_{i,N}]$$

where each source symbol x_i represents one byte of data. Grouping the source packets along h incoming edges, as a generation, gives the source vectors,

$$\boldsymbol{x_1}, ..., \boldsymbol{x_h}$$

which is written in matrix form as,

$$\boldsymbol{X} = \begin{vmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_h} \end{vmatrix} = \begin{vmatrix} x_{1,1} & x_{1,2} & ... & x_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{h,1} & x_{h,2} & ... & x_{h,N} \end{vmatrix} \tag{2.1}$$

Following the same formation, the coding coefficient symbols $g_i(e)$, along edge e are written in vector form as,

$$\boldsymbol{g(e)} = [g_1(e), g_2(e), ..., g_h(e)]$$

where $\boldsymbol{g(e)}$ corresponds to the global coding vector for the edge e. Combining the global coding vectors for h incoming edges,

$$e_1, e_2, ..., e_h$$

is represented in matrix form, with rank h as,

$$G_t = \begin{vmatrix} \boldsymbol{g(e_1)} \\ \vdots \\ \boldsymbol{g(e_h)} \end{vmatrix} = \begin{vmatrix} g_1(e_1) & g_2(e_1) & ... & g_h(e_1) \\ \vdots & \vdots & \ddots & \vdots \\ g_1(e_h) & g_2(e_h) & ... & g_h(e_h) \end{vmatrix} \tag{2.2}$$

The output symbol for a given edge e is calculated as the linear combination of the source packets and the coding coefficients,

$$y(e) = \sum_{i=1}^{h} g_i(e)x_i = g_1(e)x_1 + g_2(e)x_2 + ... + g_h(e)x_h \tag{2.3}$$

Combining all the output symbols along e into a single output packet $\boldsymbol{y(e_i)}$ provides the output vector,

$$\begin{aligned} \boldsymbol{y(e_i)} &= \begin{vmatrix} y_1(e_i) \\ \vdots \\ y_N(e_i) \end{vmatrix}^T \\ &= \begin{vmatrix} g_1(e_i)x_{1,1} + g_2(e_i)x_{2,1} + ... + g_h(e_i)x_{h,1} \\ g_1(e_i)x_{1,2} + g_2(e_i)x_{2,2} + ... + g_h(e_i)x_{h,2} \\ \vdots \\ g_1(e_i)x_{1,N} + g_2(e_i)x_{2,N} + ... + g_h(e_i)x_{h,N} \end{vmatrix}^T \\ &= \boldsymbol{g(e_i)X} \end{aligned} \tag{2.4}$$

Further, combining all the output packets for a generation size h, provides the encoded output matrix,

$$\begin{aligned}
\boldsymbol{y(e)} &= \begin{vmatrix} \boldsymbol{y(e_1)} \\ \vdots \\ \boldsymbol{y(e_h)} \end{vmatrix} = \begin{vmatrix} y_1(e_1) & y_2(e_1) & ... & y_N(e_1) \\ \vdots & \vdots & \ddots & \vdots \\ y_1(e_h) & y_2(e_h) & ... & y_N(e_h) \end{vmatrix} \\
&= \begin{vmatrix} g_1(e_1) & g_2(e_1) & ... & g_h(e_1) \\ \vdots & \vdots & \ddots & \vdots \\ g_1(e_h) & g_2(e_h) & ... & g_h(e_h) \end{vmatrix} \begin{vmatrix} x_{1,1} & x_{1,2} & ... & x_{1,N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{h,1} & x_{h,2} & ... & x_{h,N} \end{vmatrix} \\
&= \begin{vmatrix} \boldsymbol{g(e_1)} \\ \vdots \\ \boldsymbol{g(e_h)} \end{vmatrix} \begin{vmatrix} \boldsymbol{x_1} \\ \vdots \\ \boldsymbol{x_h} \end{vmatrix} \\
&= G_t \boldsymbol{X}
\end{aligned} \tag{2.5}$$

Once the packets are encoded, they can transverse through the network to the decoder. The main aim of the decoder, is to decode the incoming encoded packets, in order to solve for the original source packets,

$$\boldsymbol{x_1}, ..., \boldsymbol{x_h}$$

The original source symbols are found by combining the received encoded packet and coefficient data into matrices as a system of linear equations and solving for the inverse of the input data matrix to obtain, from [5],

$$\begin{aligned}
\begin{vmatrix} G_t & \boldsymbol{y(e)} \end{vmatrix} &= \begin{vmatrix} g_1(e_1) & g_2(e_1) & ... & g_h(e_1) & y_1(e_1) & y_2(e_1) & ... & y_N(e_1) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1(e_h) & g_2(e_h) & ... & g_h(e_h) & y_1(e_h) & y_2(e_h) & ... & y_N(e_h) \end{vmatrix} \\
&= \begin{vmatrix} 1 & 0 & ... & 0 & x_{1,1} & x_{1,2} & ... & x_{1,N} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 1 & x_{h,1} & x_{h,2} & ... & x_{h,N} \end{vmatrix} \\
&= \begin{vmatrix} I_h & \boldsymbol{X} \end{vmatrix}
\end{aligned} \tag{2.6}$$

In RLNC, the random coefficients are taken from a Galois (also known as finite) field. Network coding operations are also performed over the Galois field. A Galois field is a mathematical field that contains a finite number of elements [22]. The field used throughout this thesis is a Galois field of degree eight ($GF(2^8)$), containing 256 elements.

This concludes the details regarding the necessary background on traditional networking concepts as well as on network coding. We continue the discussion onto the suitable platform for a software-based network coding switch.

2.5 Software-based network coding

A suitable platform for network coding in the software implementation is required. We review past software-based network coding implementations that have been implemented practically. Investigation into past implementations help address the requirement to be able to use the RLNC algorithm.

2.5.1 Software-based network coding implementations

Network coding is used in [23] to implement a co-operative peer-to-peer system for video streaming. The work done by Keller, et al. propose the MicroCast system to help aid in multicast streaming scenarios. A scenario where a group of smartphone users in the same proximity, want to watch the same video simultaneously is used.

The architecture of MicroCast includes the MicroNC-P2 algorithm that implements network coding. The network coding scheme uses generation-based coding over the field $GF(2^8)$. The coding coefficients are selected uniformly at random and therefore the scheme uses RLNC. The coding scheme is implemented in Java to be run on the Android mobile OS.

The paper mentions that network coding is a CPU intensive operation. To reduce CPU usage, the generation size is limited to 25 packets of 900 bytes each. Two implementations are created, the one with pure Java and the other through the Java Native Interface (C-based). Both implementations use lookup tables for Galois field multiplication and division, and exclusive-or (XOR) operations for addition and subtraction.

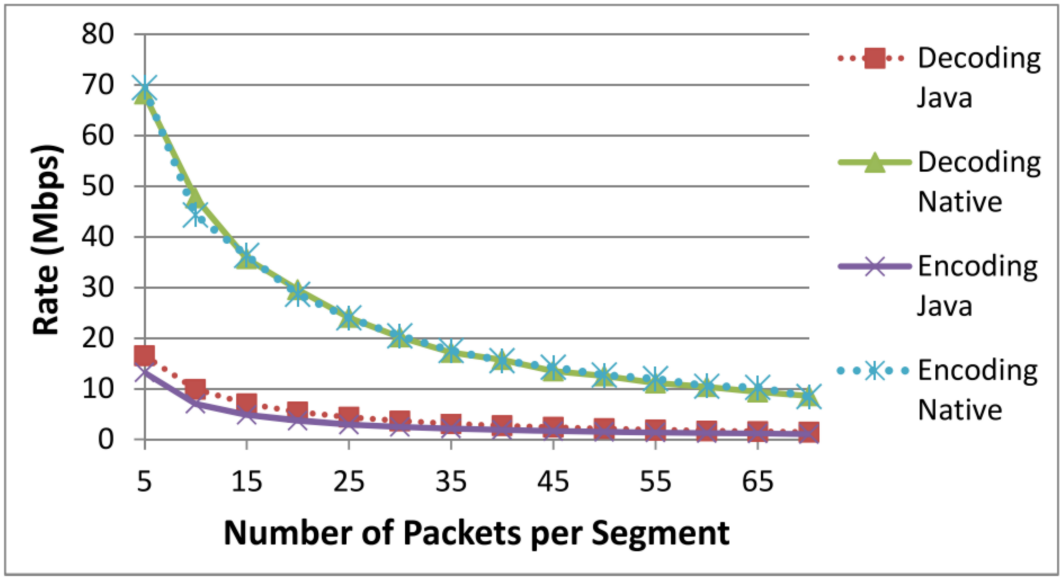

Figure 2.5: Throughput rates of the MicroCast network coding implementations for different segment (generation) sizes, taken from [23].

The pure Java implementation can encode and decode at a throughput of 2.9 Mbps and 4.3 Mbps respectively. The Java Native Interface implementation achieves a higher 24 Mbps for both encoding and decoding. The study shows the coding rates when the number of packets used in the generation is varied. The results are shown in Fig. 2.5, and indicate that higher coding rates are achieved with smaller segment (generation) sizes.

Another practical application of RLNC is shown with indoor sensor networks in [24]. RLNC is used as an erasure code and compared alongside Forward Error Correction (FEC). The results show that RLNC works best in environments of severe interference, while FEC works better in low interference environments. For these reasons, Angelopoulos, et al. propose a Joint Channel-Network Coding (JCNC) scheme [24].

Due to the delay and memory constraints of low-power sensors, the RLNC implementation in [24] uses a generation size of four packets. The RLNC operations are done over the field $GF(2^8)$ and make use of Linear Feedback Shift Registers (LFSR) to generate the random coefficients.

While the work done by [23] and [24] provides insight into network coding algorithms and their success in practise, the RLNC algorithm code is not made available. This makes it difficult for researchers to study network coding. To solve this problem Pedersen, et al. created the Kodo network coding library [25].

2.5.2 Kodo network coding library

The Kodo software library provides a research friendly environment to evaluate network coding. Kodo provides researchers with an open source tool for using network coding in practise. Coding schemes can be implemented with existing networking protocols. Kodo provides network coding algorithms including RLNC. The algorithms and functions are written in C++, C and Python.

Kodo works by segmenting data into chucks of h packets of length N. The span of hN is a generation as seen from [5]. Kodo provides a basic partitioning algorithm, which handles segmenting data into these generations. The user is able to select the generation size, packet size and the field size. The choice of field size is a trade-off between computational cost and coding efficiency. Registration is required to view the GitHub source and the software can be used freely in a research or education environment. Steinwurf provide an extensive tutorial and API for easy integration into user projects [8].

Due to Kodo's ease of use and implementation, it has been widely adopted. Kodo is considered the recommended network coding software library, while research shows that there are not many other software libraries that have similar support and scale. For this reason Kodo is investigated to implement network coding algorithms in the software implementation.

An example of Kodo in practise is the PlayNCool opportunistic protocol, introduced in [26]. The paper uses RLNC to increase wireless mesh network

(WMN) throughput. Simulations are done using the Kodo C++ library. Their reason for using Kodo is to be able to translate the simulated implementations to a practical network if required.

Hansen, et al. describes Kodo as a high-performance library that enables network coding to be deployed in a practical network with ease [1]. Specifically, Kodo is used to implement RLNC as a VNF in an SDN network. The coding scheme uses systematic code, where the original input symbols are first sent uncoded along with the encoded symbols. The coding operations are performed over the field $GF(2^8)$ using a generation of 10 packets, of 1356 bytes each.

The Kodo library will allow the software-based network coding platform to use the RLNC algorithm in practise. Due to the lack of open source RLNC software, the alternative would be to implement the network coding algorithm from the RLNC theory. Due to the maturity and widespread use of Kodo however, it is chosen to implement the RLNC functionality of the software-based platform.

2.6 Software-based network coding with SDN

Another requirement of the software implementation is to be compatible with SDN. To make the software implementation SDN compatible, we look at past network coding implementations that integrate with SDN.

Nemeth, et al. was one of the first to combine network coding and OpenFlow in [27]. A testbed is presented for inter-session network coding (combining packets from different network coding flows). The work demonstrates that smarter switches, such as those with OpenFlow are crucial for deploying network coding in practical network environments. The implementation extends the OpenFlow protocol by adding custom actions: *XOR-encode*, *XOR-decode* and *set-mpls-label-from-counter*. Network coding metadata is sent using Multicast Label Switching (MPLS) labels.

The first label is used by the OpenFlow switch to match packets to the flow table. The *set-mpls-label-from-counter* action is then used to prepend sequence numbers. The *XOR-encode* action makes a copy of the original packet, and places it on the encoding queue. Once packets are encoded the MPLS labels are updated and the encoded packets are sent through the pipeline. The decoding action *XOR-decode* is used to decode the encoded packets using their sequence numbers.

Another network coding and SDN integration, that also makes use of MPLS labels is done by Yang, et al. in [28]. The work done by [27] is only suitable for the butterfly network and therefore not real networks. Yang, et al. expands their work and proposes an OpenFlow Network Coding (OFNC) architecture with a standard communication protocol between the data and control planes. Mininet is used to simulate and evaluate the performance. The results show

that while link capacity is improved, coding calculation overhead causes transmission delays.

Network coding is integrated with SDN in a Wireless Mesh Networks (WMNs) application in [29]. This study implements their own protocol, *OpenCoding* which has similar SDN functionality to OpenFlow, but with network coding. OpenCoding makes use of SDNs entire network view to optimise network resources and make effective use of network coding. This approach showed improvements over standalone OpenFlow or network coding protocols.

The implementation from Hansen, et al. in [1] integrates Kodo with OpenFlow. A realistic network showcase of network coding is presented. A virtual network environment, using OvS is used to demonstrate easy integration of network coding into existing SDN systems. This approach enables network coding to be applied only when necessary, preventing limitations in scenarios where network coding does not provide improvement.

A more recent paper by Gabriel, et al. demonstrates a practical deployment of network coding for real-time applications [15]. The implementation deploys RLNC with SDN and NFV to improve video streaming quality over lossy and congested network channels. An OpenFlow-enabled controller is used to manage the network and the OpenStack cloud platform is used to manage the network coding functions. Instead of adding custom actions to OpenFlow as in [27], [28] and [29], flows are used to redirect traffic to the VNFs. OvS is used by OpenStack to implement switching between network interfaces.

The use of NFV to integrate software-based network coding and SDN enables the requirement of being able to deploy the software-based network coding platform within a virtualization environment. We therefore continue the discussion onto possible ways of implementing network coding as a VNF for the software-based implementation.

2.7 Software-based network coding within a virtualization environment

The first step in creating a VNF is to figure out how to perform packet processing on the Linux operating system. Specifically, network packets need to be modified in order to provide RLNC operations. We therefore continue the discussion to userspace orientated packet processing techniques.

2.7.1 Linux-based packet processing

The Linux network stack is shown in Fig. 2.6. The layers are abstracted into three primary sections, the userspace, Kernel space and physical space. The Kernel networking layers make use of POSIX sockets. The application begins by creating a socket. The socket's file descriptor is used to receive and transmit

packets over the network. The socket operations are performed as system calls using the system call interface.

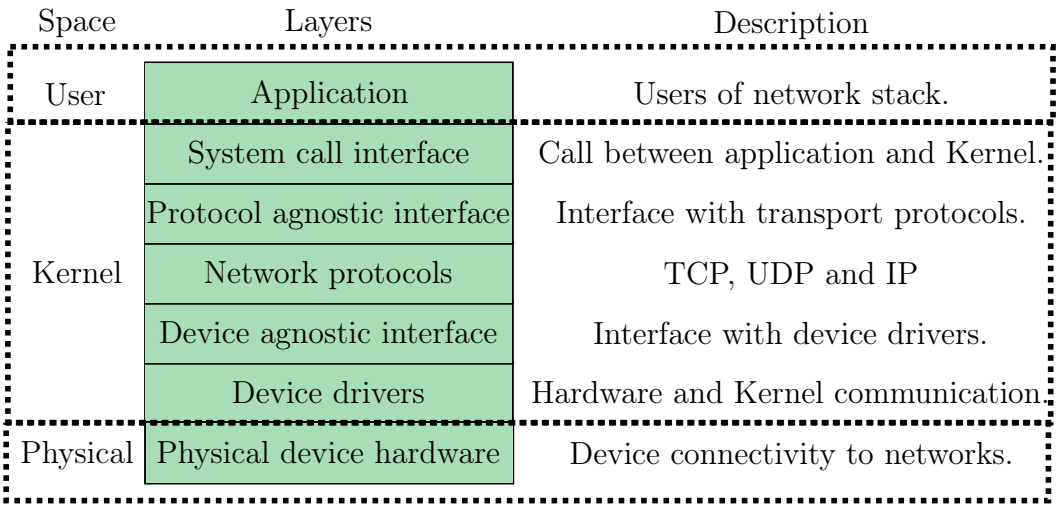

Figure 2.6: Linux networking layer model segmented into userspace, kernel and physical space regions.

A generic method of processing packets within a Linux program is through the use of the universal *TUN/TAP* device driver [30]. TUN/TAP interfaces can either be used in TUN (TUNnel) or TAP (Terminal Access Point) mode to operate on the Ethernet or IP layers respectively. TUN/TAP devices are virtual network interfaces and unlike a physical interface, TUN/TAP allows for a userspace program to receive and transmit network packets. The userspace program can therefore perform packet manipulation on network traffic received by the TUN/TAP interface, using the C programming language. The network coding functions can be implemented in the same userspace program as the TUN/TAP interface. An alternative to using TAP/TUN is to create a custom program to interface with the OS kernel. This would be out of the scope of this thesis and therefore the TUN/TAP interface is used instead.

The TUN/TAP userspace program works by opening the "/dev/net/tun" file descriptor in Linux and using an input/output control system call to register a network device with the Kernel. The network device functions exactly like a physical device and can be assigned a MAC and IP address. Network traffic to be processed can therefore simply transmit to the TUN/TAP interface. The network device will disappear when the userspace program closes, unless a persistent device is created.

Packet processing using TAP/TUN within Linux provides great flexibility. With an increase in the use of 10GbE NICs however, the kernel networking stack begins to bottleneck processing performance [31]. While the network

implementations in this thesis do not explicitly make use of networking equipment equal or greater than 10Gbps, commercial networks and data centers do [32]. In fact, companies such as Mellanox already provide NICs of up to 200Gbps [33]. In order to keep this thesis relevant for future work, alternative packet processing techniques are investigated.

Bottlenecks can be alleviated through the use of *Kernel-bypass* networking. Kernel-bypass as the name implies, moves network protocol processing from the kernel to the userspace. In order to understand exactly why a Kernel-bypass is needed, we need to understand how packets are normally processed in Linux. The following explanations are taken from [34].

When a packet is received by a network interface in the physical space, it is first sent to a receive queue. The packet is then copied to the computers Random Access Memory (RAM) using Direct Memory Access (DMA). DMA allows for devices to access the RAM without the CPU being fully occupied during memory transfers. The CPU is used to initiate the transfer and receives an interrupt when the packet transfer is complete.

Once the DMA transfer is complete, the new packet is placed into a specifically allocated Kernel space buffer, namely the socket buffers "sk_buff struct". These buffers are allocated for each incoming packet and become free when the packet is transferred to the userspace.

The first problem with this approach is that many bus cycles are required to continually transfer packet data from the sk_buff to the userspace for each incoming packet. Another problem is that the Linux networking stack was designed to facilitate as many networking protocols as possible. As a result, the sk_buff struct contains metadata for all the protocols and is therefore overcomplicated. This reduces packet processing speed because each struct contains more data than necessary.

Kernel-bypass networking reduces the overheads caused by Kernel space packet processing. Packet handling can be moved to the hardware, OS or userspace. Hardware-based packet processing can be done through the use of SmartNICs. SmartNICs, however are expensive network devices and not necessary for the network speeds in this thesis. Instead, a userspace orientated Kernel-bypass is investigated.

2.7.2 DPDK

A userspace Kernel-bypass project that have been heavily adopted is DPDK. The work done by Kourtis, et al. showcase that VM-based VNFs are able to achieve significantly higher packet throughput performance when using DPDK over the Linux Kernel network stack [35]. Another example of DPDK performance gain is demonstrated in [36]. The white paper boasts a twelve times increase in performance of OvS when using the DPDK datapath over the Linux networking stack.

Further investigation into the usage of DPDK with OvS, shows that DPDK is widely adopted for use with OvS. Cloud computing platforms such as OpenStack and CloudStack both have tutorials on using DPDK with OvS in [37] and [38] respectively. For these reasons, DPDK is investigated as the method of Kernel-bypass for the implementations in this thesis.

The DPDK software consists of many drivers and libraries that can be used to perform packet processing. Not only does DPDK provide packet manipulation, but it also minimises the number of CPU cycles required to do so. DPDK makes it more easy and intuitive to develop packet processing applications by providing an extensive documentation. An overview of the DPDK architecture is given in Fig. 2.7 from [11].

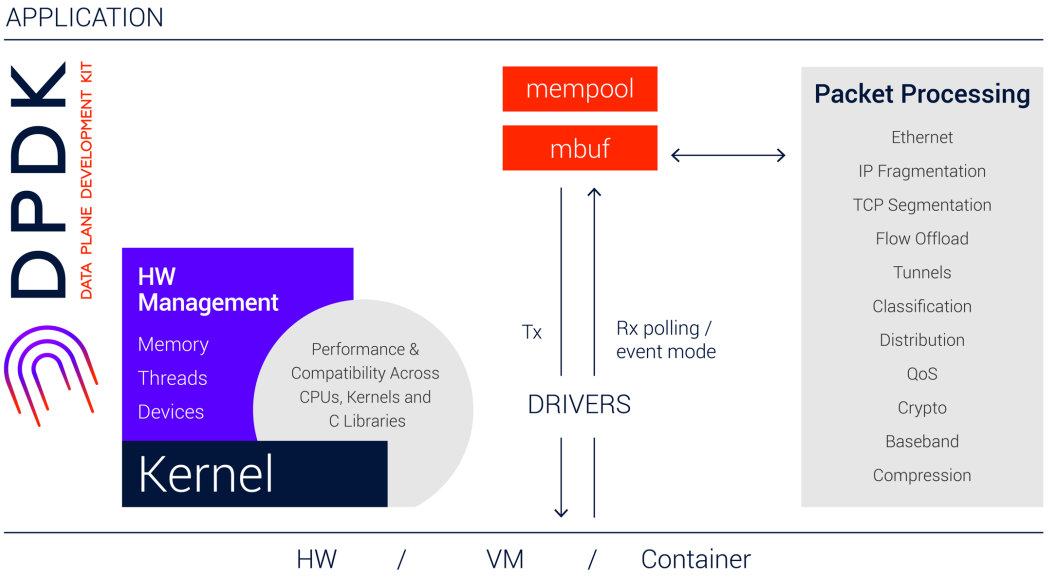

Figure 2.7: DPDK overview diagram showcasing available packet processing functionality, from [11]

DPDK operations begin by *unbinding* network devices from the Linux network stack. The ports are then managed by one of the DPDK provided drivers: *vfio_pci*, *igb_uio* or *uio_pci_generic*. These drivers enable the userspace to interact with the network devices. While these drivers still contain Kernel modules, they are only used to initialise devices and do not make use of the Linux networking stack.

Once the network devices are bound to DPDK, further communication is done using the DPDK Poll Mode Drivers (PMDs). PMDs are the main reason for DPDK's performance increases. A PMD is a collection of APIs that run in the userspace. PMDs are used to quickly access the receive and transmit descriptors of network devices in order to accelerate packet processing.

Along with PMDs, DPDK makes use of *huge pages* [39]. Huge pages in Linux are page sizes bigger than 4KB, and specifically in an x86-64 architecture are 2MB or 1GB. A memory page is a contiguous block of virtual memory that enables applications from having to manage shared memory space. Virtual memory also provide memory isolation as well as let applications use more memory through the use of paging. A Translation Lookaside Buffer (TLB) is used to store virtual memory to physical memory translations. The TLB cache is limited in size. Huge pages increase performance by reducing the number of address translations and TLB usage, therefore increasing the performance of DPDK.

This concludes the literature review regarding the software implementation of network coding. We therefore continue the discussion onto finding a suitable platform for hardware-based network coding. The hardware platform requirements are to implement the RLNC algorithm and to be integrated with SDN.

2.8 Hardware accelerated network coding

One of the main bottlenecks of network coding is the computational complexity of the encoding and decoding operations. The above mentioned network coding implementations are mostly done using CPUs. Other hardware accelerators such as Graphics Processing Units (GPUs) and FPGAs are able to provide performance increase of software algorithms. This is mainly due to the ability of using parallel processing. For this reason we investigate alternative processing implementations of the RLNC algorithm.

2.8.1 Network coding on GPUs

A GPU is a dedicated hardware device that is able to perform multiple parallel mathematical calculations. The primary use case for GPUs are in image and video graphics processing. While many of these tasks can and were previously done on the CPU, they can be offloaded to the GPU for increased performance. Due to the increased performance gain in parallel processing applications, GPUs have been used to implement many other algorithms apart from visual rendering. This makes it possible to implement network coding on a GPU.

One of the first GPU implementations of network coding is done by Chu, et al. in [40]. Algorithms are proposed that aim to maximize the parallelism of encoding and decoding operations. The implementations provided make use of the Computer Unified Device Architecture (CUDA) programming model to run network coding algorithms on the GPU.

The network coding operations are performed in the field $GF(2^8)$. The encoding process makes use of logarithmic and exponential tables to perform

Galois field multiplication. It takes $O(hN)$ time to encode an entire generation in their encoding operation. The decoding operation solves a system of linear equations after receiving all h packets in the generation. However Chu, et al. do not use the GPU to perform decoding due to parallel implementation issues using CUDA. Instead, the CPU is used to compute the inverse matrix.

Shojania, et al. present another GPU-based network coding implementation, *Nuclei* in [41]. The coding functions are created using CUDA, written in C. One of the performance bottlenecks in [40]'s implementation is that the Galois multiplication is done using lookup tables, and memory access delays are imposed. To curb this, [41] uses a loop-based multiplication and shows that the approach is better suited for parallel implementation.

Both above mentioned implementations show that GPUs are able to provide performance increases in network coding operations, using parallel processing. The one problem with GPUs are that they cannot easily be implemented in embedded devices, such as those used in IoT applications. This makes it less feasible to use a GPU in the network coding test platform. FPGAs however, do not need to be implemented alongside a PC and therefore can be deployed in an embedded environment more easily. For this reason we consider FPGAs for hardware acceleration.

2.8.2 Network coding on FPGAs

A design for an FPGA-based network coding decoder is proposed in [14]. The article by Kim, et al. provides and compares two different decoding methods. The approach improves decoding speeds by dividing and paralleling the decoding process. The optimization is focused on Galois field operations and performing matrix multiplication.

RLNC is used as the network coding algorithm, and all computation is done in the field $GF(2^8)$. Two types Galois field Arithmetic Logic Units (ALUs) are compared. The one method is a table lookup method and the other is a computation-orientated method. The table lookup method was also seen in [23] and [40]. A precalculated table is used to store the results of Galois field operations. Using logarithm and antilogarithm, the lookup tables allow for Galois field multiplication and division to simply become addition and subtraction.

Two decoding architectures are proposed that each make use of the different Galois field ALUs: *Inversion first (INVF)* and *Full Gauss-Jordan (FGJ)*. The first method uses Gauss-Jordan elimination to find the inverse of the coefficient matrix and then multiplies the result with the encoded matrix. The FGJ method performs Gauss-Jordan elimination on both the coefficient and input data to obtain the source data.

The resultant implementation is done using a Xilinx FPGA device and synthesized with the Xilinx ISE package. The modules are then evaluated using ModelSim SE. The maximum possible clock frequency is 83.3 MHz. The

highest throughput produced is 65.98 Mbps using the INVF method with the computational-orientated Galois field ALU with a generation size of 16 packets. The results are compared to AMD Phenom-X4 and Intel Core 2 Quad desktop CPUs implementations, with speed increases of a multiple of 13.84 and 6.73 respectively.

The work done by Kim, et al. in [14] shows that RLNC can feasibly be implemented on an FPGA. Their implementation however, is only evaluated through simulation and not a real network. While a Virtex 5-based FPGA is used during synthesis, their implementation does not actually use a real FPGA either. The question remains as to the performance when RLNC is implemented on an actual FPGA in a real network.

An FPGA is required to implement the RLNC functions in hardware, within a real network. The first requirement is to be able to interface the FPGA with the computer. PCIe is a high speed expansion interface used to connect add-on cards by providing multiple fast lanes to interface with the computer. For this reason we investigate two PCIe-based FPGA platforms for this thesis: *NetFPGA* and *OpenVINO Starter Kit*.

2.8.3 NetFPGA

The NetFPGA is a Xilinx-based FPGA network development platform with four network interfaces [42]. There exist multiple different variants such as the SUME, CML, 10G and 1G models. Due to the flexibility of the FPGA, more than 3500 different network systems have been implemented on the NetFPGA platform [43].

The original NetFPGA OpenFlow switch developed in [44] can be extended to include additional network functionality. This is shown in [45] and [46] to implement computationally intensive network inspection and attack prevention algorithms. Other integrations of OpenFlow on the NetFPGA have been done in [47], [48] and [49]. These integrations discuss the advantage of using an FPGA for increased portability, flexibility and performance in an SDN environment.

The NetFPGA CML model is a 1Gbps device with a Xilinx Kintex-7 and is one of the current actively supported NetFPGA models [50]. The CML is ideal for implementing complex network devices while still remaining cost effective compared to the 10Gbps SUME model. The PCIe 2.0 interface is used to connect to the CML, which enables the host computer to act as the OpenFlow controller, while additional nodes are connected to each of the Ethernet ports. This configuration along with the FPGA enables flexible implementation of different networking devices. The CML model is compatible with existing Xilinx design kits, allowing for the use of Xilinx design software and Ethernet Intellectual Property (IP) cores [51].

While the NetFPGA provides great flexibly, the most basic model costs 1400 USD. DPDK and OvS shows that it is feasible to implement the Open-

Flow switch on the Linux OS and offload traffic if needed. It is therefore not necessary to have the NICs directly attached to the FPGA, but instead as a seprate device. For this reason, we continue the search for a more cost effective FPGA development platform.

2.8.4 OpenVINO Starter Kit

The OpenVINO Starter Kit is an PCIe-based Intel Cyclone V-based FPGA development board intended for developing OpenCL and other mainstream applications [52]. The OpenVINO starter kit is priced at 525 USD, and does not include on board NICs. A diagram of the developemnt kit is shown in Fig. 2.8.

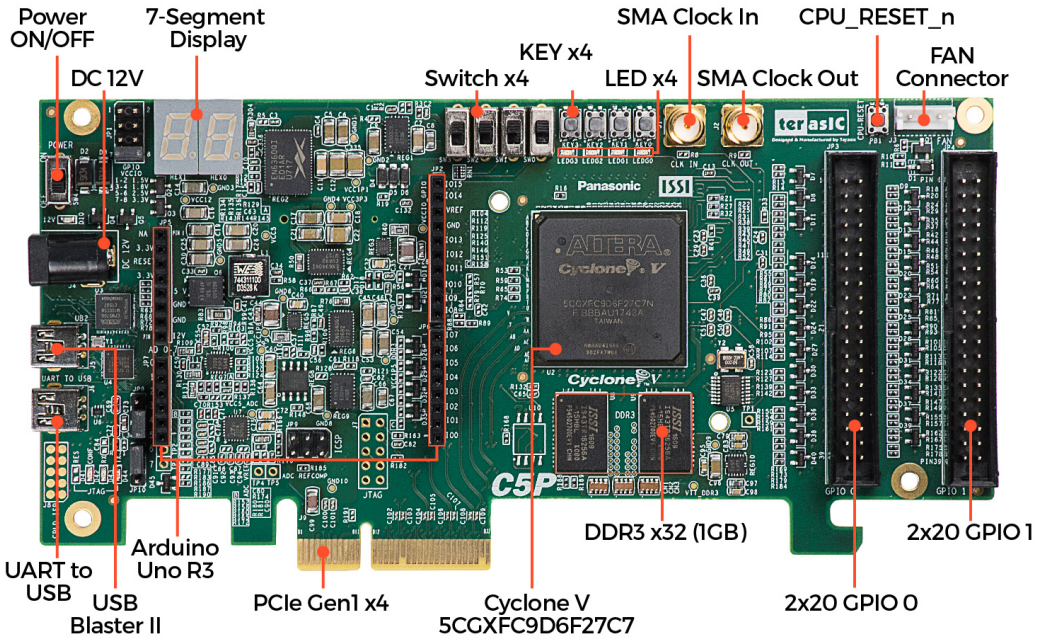

Figure 2.8: OpenVINO Starter Kit hardware diagram, from [52].

While the OpenVINO starter kit uses an older PCIe generation (version 1.0) than that of the NetFPGA (version 2.0), Intel reports PCIe version 1.0 with four lane performance throughput benchmarks of 6584 Mbps and 7056 Mbps for DMA read and write respectively [53]. These results are just shy of the theoretical PCIe four lane maximum of at 7416 Mbps for both DMA reads and writes. The OpenVINO starter kit is therefore more than capable of providing the required throughput for hardware offloading of network packets. The expected throughput results for hardware offloading are based on the FPGA decoder implementation of [14].

An important consideration for an FPGA is the number of resources. This is difficult to do because each hardware vendor, and each FPGA generation use different resource implementations. Direct comparisons between devices is therefore not always accurate. The resource utilization results from [14] cannot be used to determine which Intel FPGA will work, specifically the Cyclone V on the OpenVINO kit.

Luckily, with Intel Quartus Prime, resource utilization is determined after place and route and can be done without the physical FPGA. Therefore the design can be completed and the resultant resource usage can be checked to see if it will fit on the device. This approach is used in this thesis. Chapter 7 shows that the implementation fits on the OpenVINO starter kit.

The OpenVINO starter kit will allow the hardware requirements for a network coding switch to be met. The RLNC algorithm can be implemented in FPGA logic on the Cyclone V device. The development kit can be integrated with a computer to be used in an SDN environment and therefore a real packet-based network.

We have established how to meet the requirements for the software and hardware-based network coding platforms. We continue the literature review onto meeting the additional network coding switch functionality.

2.9 Network coding switch functionality

The network coding switch needs to be SDN compatible. From the above mentioned work it is clear that SDN, specifically OpenFlow is a crucial requirement for integrating network coding into existing networks. It is also shown that OvS is a feasible and popular virtual network switch that uses the OpenFlow protocol. For this reason OvS is investigated further.

2.9.1 Open vSwitch

The emergence of network virtualization requires a virtual network switch with much functionality. OvS is a virtual switching platform that uses the OpenFlow protocol [13]. OvS was designed for virtual environment networking as a alternative to traditional software switching architectures.

OvS works with most hypervisors and operating systems. VMs can be interconnected with minimal effort using OvS as the virtual switch. OvS is designed to be flexible and open and for this reason has been heavily adopted in practical networking applications [13].

The components of OvS is shown in Fig. 2.9. The main component is the userspace switch daemon *ovs-vswitchd*. The userspace switch daemon instructs the *Kernel datapath module* on how to handle incoming packets through OpenFlow actions. The kernel datapath module is used to access the packet buffers on the physical NIC and virtual interfaces. The *ovsdb-server* is used by the

daemon to store the OpenFlow flow tables. The flow table entries are determined by the controller. The first packet received on the port needs to be sent to the ovs-vswitchd module to determine the appropriate forwarding actions. Subsequent packets however, take full advantage of the kernel and do not get sent back to the userspace unless the flow rules change.

OvS allows for many of the network coding switch functionality requirements to be met. The functionality of performing basic packet forwarding is met as OvS is able to function as a L2 learning switch by implementing the necessary OpenFlow rules. OvS also meets the requirements of SDN compatibility by being OpenFlow compatible. To meet the requirements of being able to perform multicast snooping, the OvS bridge can implement the necessary flow rules by creating a controller application that monitors multicast network traffic.

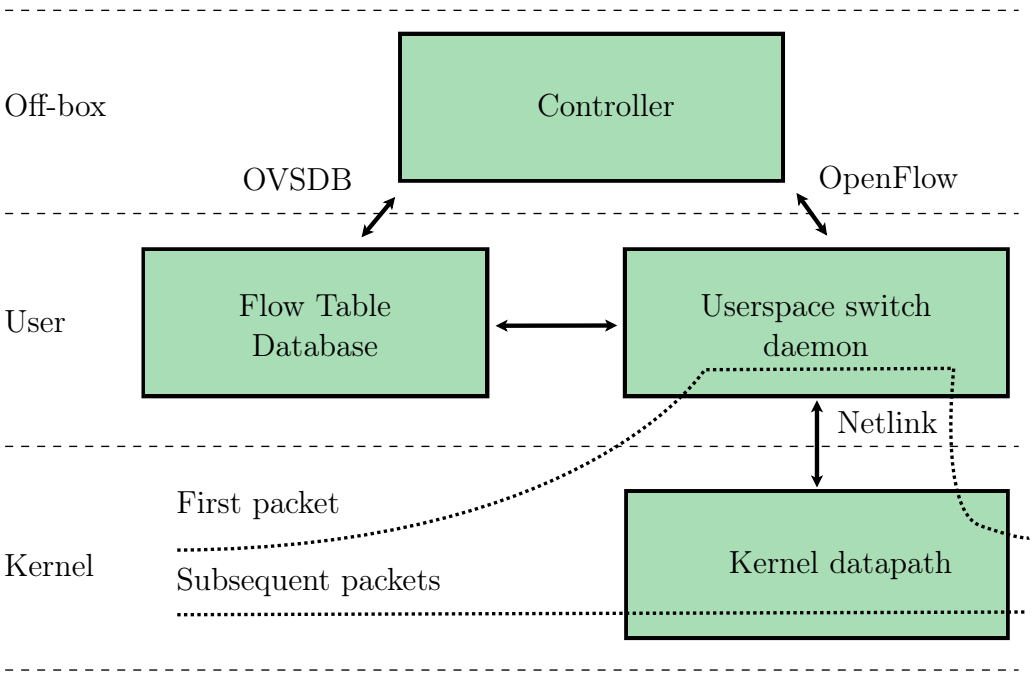

Figure 2.9: Open vSwitch diagram showing integration between userspace and kernel components. The first packet is sent to the userspace switch daemon and subsequent packets are forwarded via the faster kernel component.

OvS provides great flexibility by allowing for network functions to be deployed as required. Software-based network coding can be deployed as a VNF connected to OvS. In order to deploy a VNF, a hypervisor is required. A hypervisor is a process layer that allows for VMs to be created, managed and run on a host machine. There exist two hypervisor types, namely type 1 and type 2. The difference is that type 1 hypervisors run directly on the hardware,

while type 2 hypervisors run on the host OS. The difference is shown in Fig. 2.10. Examples of type 1 hypervisors are Hyper-V, Xen and ESXi. These are often used in datacenters and provide the best performance. Examples of type 2 hypervisors are VMWare Workstation and Virtualbox.

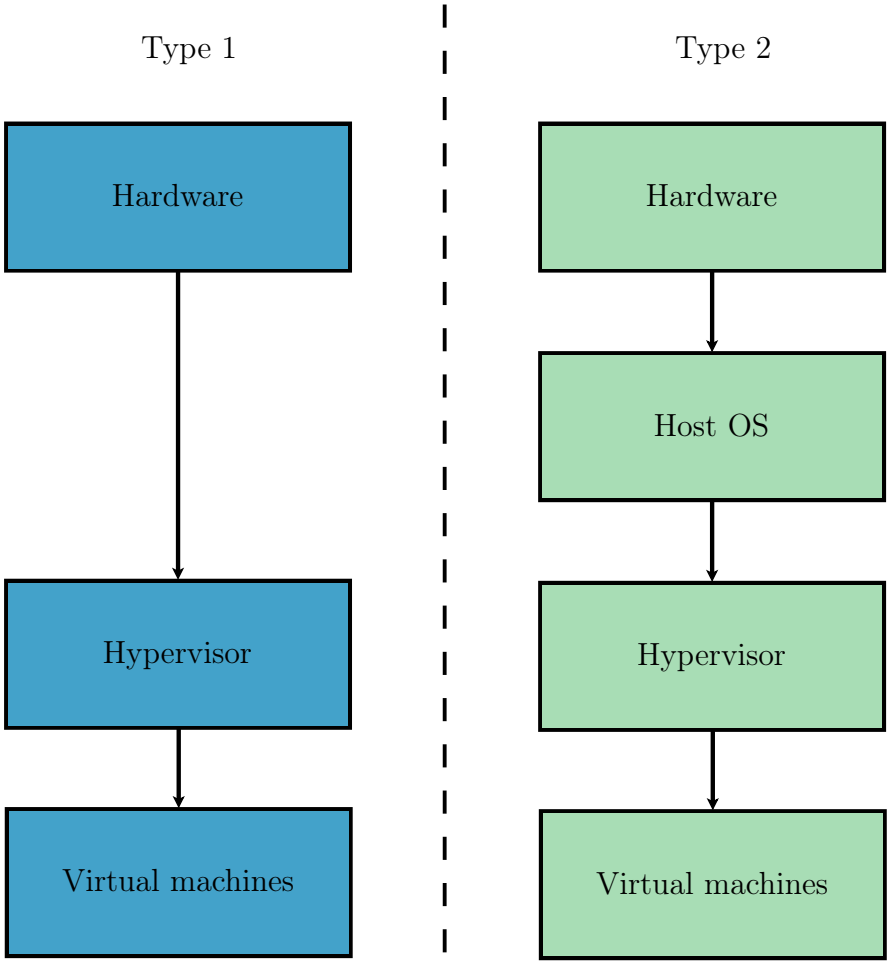

Figure 2.10: Type 1 vs Type 2 hypervisor. Type 2 runs on the host OS and includes an additional component over type 1.

The Kernel-based Virtual Machine (KVM) is a virtualization platform that is built into Linux. KVM is known as a hybrid hypervisor and enables Linux to be turned into a type 1 hypervisor, while still maintaining OS level flexibility that comes with a type 2 hypervisor. KVM can be used alongside Quick Emulator (QEMU) to create, run and manage VMs directly on the Linux OS. The KVM hypervisor is used in this thesis as the work is not dependant on any specific hypervisor and the authors preference of using the Linux OS over Windows. The software-based implementation can be moved to the Windows OS (using the hyper-v hypervisor) with minimal effort if required. The VNF

VMs however, are designed to run the Linux OS and therefore the VNF design is focused on Linux only compatibility.

The hypervisor provides virtual ports to connect the VMs to the OvS bridge. Physical network ports such as those provided by a NIC, can be connected to the OvS bridge to interface the VM and physical machine networking. This allows OvS to meet the requirement of functioning in both the physical and virtual network space. This also meets the requirements of being able to offload traffic to both the software and hardware network coding implementations.

2.10 Summary

This chapter provided a literature review to determine the objectives and requirements of the network coding capable switch. The review begins by discussing the traditional approach to packet-based network switching. Concepts such as the Ethernet protocol and switch are discussed as well as multicast networking.

The theory behind network coding is provided. The discussion begins by investigating links between the original idea of network coding, to practical implementations, with focus on practical parameters to utilize network coding in the software and hardware implementations. The most important network coding algorithm for practical network coding, RLNC is discussed in detail. Packet representation of the encoding and decoding process are discussed to provide the necessary background for implementing network coding in practise.

Next, the software-based network coding platform is investigated. The requirements of implementing the RLNC algorithm in software is investigated by reviewing past software-based network coding implementations. The Kodo software library is shown to be the most viable option to add RLNC functionality to the software coding platform. Further requirements of the software platform are investigated by looking at past work of software-based network coding within SDN and VNF environments. Methods of implementing network coding within a VNF are discussed by looking at Linux-based packet processing and the DPDK packet processing library.

The use of GPUs and FPGAs are both investigated as possible hardware-based network coding platforms. FPGA implementations are favoured over GPUs due to being more flexible in IoT and embedded system applications. With regards to using FPGA, two possible options are investigated: the NetFPGA and the OpenVINO starter kit. The cost of the OpenVINO starter kit makes it the more desirable device, while still being able to meet the requirements of implementing the RLNC algorithm and being SDN compatible.

The literature review concludes by investigating OvS as a solution to meet the necessary requirements of the additional network coding switch functionality.

Chapter 3

System architecture overview

3.1 Introduction

The previous chapter provided a literature review in order to meet the requirements of the network coding switch. In this chapter, these requirements are used to determine an overall system architecture design.

3.2 System architecture

The system is split into three sections: bridge interface, physical layer and virtual layer. Both the physical and virtual layer contain respective coder and host components. The overall system architecture is shown in Fig. 3.1.

The bridging interface enables network traffic to traverse seamlessly between the physical and virtual components. This is important because it allows for a researcher to evaluate network coding in their layer of choice. The objective of open networking is maintained by allowing the user the freedom to choose their networking environment. A system user can choose to deploy a combination of components for different test case scenarios. An example would be to use a VNF-based encoder to encode network traffic from one physical host to another, while using an FPGA entity as the decoder.

The physical layer provides the ability to evaluate network coding in a network with physical hosts. This provides a similar plug-and-play experience as with commercial network switches. Hardware-based network coders are interfaced to offload packets for processing.

Modern networking systems make extensive use of virtualization [13]. Providing integration with the virtual space allows for network coding to be evaluated alongside emerging virtual network technologies. Another benefit to incorporating the virtual layer is that multiple VM hosts can be deployed. Network coding scenarios can be evaluated with as many hosts as the server can handle. Use of the virtual layer therefore reduces the requirements of additional hardware and saves on equipment costs.

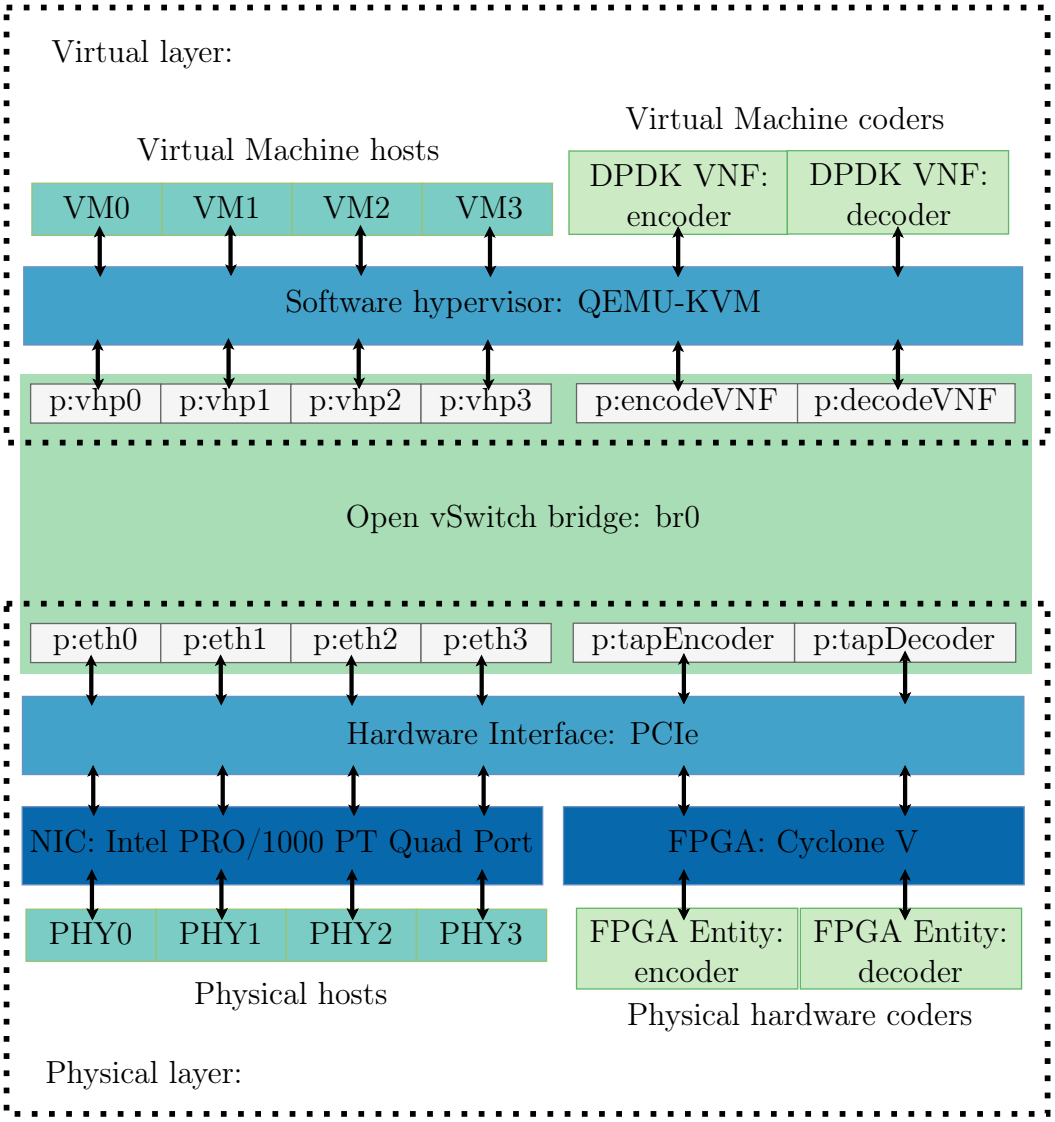

Figure 3.1: Network coding switch system overview: The switch consists of a physical and virtual layer, interfaced together using the Open vSwitch bridge. The physical layer includes an Intel quad port NIC to connect to four physical hosts. An FPGA hardware-based encoder and decoder are connected via PCIe for hardware packet processing. The virtual layer uses QEMU-KVM as a hypervisor to interface the Open vSwitch bridge to virtual machine hosts, and virtual machine-based encoder and decoder functions.

A bridging platform is required that will enable the use of both physical and virtual layers. Off-the-shelf network switches do not have the general processing power, resource availability, or flexibility to run a hypervisor with virtual machines. Hardware components cannot be added to existing network switches to perform network coding. For these reasons, the overall system is

designed to be implemented on a general purpose PC. The use of a PC enables interfacing with the virtual and physical layers using a hypervisor and PCIe respectively.

The PC needs an operating system to run the hypervisor and control software, as well as necessary drivers for PCIe. While there are many possible operating systems available, Linux is chosen. The Linux Foundation provides extensive documentation on the Linux kernel [54]. This enables great flexibility in terms of creating network interfaces. Using Linux maintains the objective of open networking. Many different networking platforms have been written for Linux, including OvS.

3.3 Bridge component: Open vSwitch

The bridging interface needs to be implemented in software in order to run on a Linux PC. The simplest way to implement software switching in Linux is to use the default Ethernet bridge administration utility, *brctl*. The brctl utility is used to set up, maintain and monitor a software-based Ethernet bridge running in the kernel. The default bridge implementation, however does not use SDN and therefore does not meet the objective of open networking.

In Chapter 2 we argue that for this thesis, the most appropriate OpenFlow-based virtual switch is OvS. The reasons for using OvS as the central switching component are as follows.

- OvS makes use of OpenFlow and is therefore SDN compatible. This ensures that the bridging component meets the objective of open networking.
- OvS is a software-based switch which enables use of both physical and virtual networking layers.
- OvS is open source and can therefore be modified to add packet processing.
- OvS runs on any Linux, BSD or Windows operating system. Therefore any computer with the capability of running an OS can be used as the base for the networking switch.
- OvS can be interfaced with DPDK to increase the performance of data plane processing. OvS-DPDK enables Linux-based packet processing above 10Gbps which is not obtainable using the default network stack. OvS-DPDK prevents the bridging component from bottlenecking the system in future iterations.

The host and coder components are interfaced with the OvS bridge *br0* through respective ports. Table 3.1 shows which ports are used by the bridge.

The first column refers to the interface and port name used to identify the port. The second column provides the port type associated with each port. The type of network, virtual or physical, is given in the third column. The last column lists the component or device connected to the port.

OvS provides various port types for different use case scenarios [55]. The port types used in this implementation are *dpdk* and *dpdkvhostuser*. The dpdk port allows for attaching physical and TAP interfaces with a DPDK-backend. This port is therefore used to connect the physical hosts and TAP interfaces that connect to the FPGA, to OvS. The virtual machines are all bridged using the dpdkvhostuserclient port. This port type makes use of DPDK-based vHost user ports. VHost User is the networking interface used by the QEMU emulator [56]. This is also the recommended use case by OvS [55].

Port\Interface	**Port Type**	**Network**	**Connected to**
eth0 (enp7s0f0)	dpdk	physical	PHY0
eth1 (enp7s0f1)	dpdk	physical	PHY1
eth2 (enp8s0f0)	dpdk	physical	PHY2
eth3 (enp8s0f1)	dpdk	physical	PHY3
tapEncoder	dpdk	physical	FPGA encoder
tapDecoder	dpdk	physical	FPGA decoder
vhp0	dpdkvhostuserclient	virtual	VM0
vhp1	dpdkvhostuserclient	virtual	VM1
vhp2	dpdkvhostuserclient	virtual	VM2
vhp3	dpdkvhostuserclient	virtual	VM3
encodeVNF	dpdkvhostuserclient	virtual	VNF encoder
decodeVNF	dpdkvhostuserclient	virtual	VNF decoder

Table 3.1: OvS Bridge port configuration: br0

3.3.1 Setup configuration

A specific sequence of Linux terminal commands are executed to configure the system from boot time to the fully interfaced layout in Fig. 3.1. These commands are written as bash shell scripts and grouped under the "netconfig" folder in the GitHub repository, *OvS-DPDK-Coding-Switch* [57]. A tabulated summary of the scripts used are given in table 3.2.

Each script is responsible for configuring a specific component of the system. The "setup_all.sh" script is used to run the other scripts in the correct order. This script can be modified by the user to enable or disable components for different use case scenarios. Screenshots depicting the outputs of the various scripts are given in appendix 8.2.

Bash script	**Action(s)**
setup_ovsdpdk.sh	Set-up and start OvS bridge with DPDK.
setup_nichosts.sh	Bind NIC ports to DPDK and add to bridge.
setup_fpga.sh	Load FPGA PCIe driver and add ports to bridge.
setup_vmhosts.sh	Start VMs and add ports to bridge.
setup_vmvnfs.sh	Start coding VNFs and add ports to bridge.
setup_all.sh	Run all of the setup scripts in order.

Table 3.2: Network configuration scripts summary

The "setup_ovsdpdk.sh" script form table 3.2 is used to configure the OvS bridge with DPDK on the Linux operating system. A flow diagram of the process is given in Fig. 3.2.

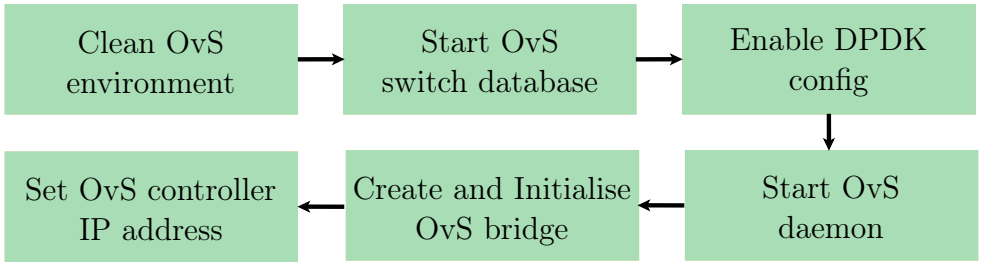

Figure 3.2: Open vSwitch and DPDK setup script flow diagram

The setup begins by cleaning the OvS environment. Any previous running OvS database server and switch daemon processes are killed. This provides a clean environment to setup OvS. The OvS database server *ovsdb* is then started. The next step is to set the runtime configuration to initiate DPDK. Once the configuration has been set, then the OvS switch daemon *ovs-vswitchd* is started. OvS is then running on the system, but no ports or flow tables exist.

Next the bridge *br0* is created. If a previous bridge is running, it is first deleted. Once the bridge is running then the controller IP details are assigned to the bridge so that the bridge can be controlled by the controller. If no controller is specified or times out, by default the bridge runs as a normal learning switch. However, if a controller is specified then full control is taken by the controller application.

Once OvS and DPDK are successfully configured, the output details are displayed to the user. The details are regarding the status of the bridge and DPDK Hugepage memory usage. The host and coder components can now begin to interface with the switch.

3.4 Physical layer

On the physical layer, two component devices are interfaced to the PC via PCIe. The first device, an Intel PRO/1000 PT quad port NIC is used to provide four physical Ethernet ports. These ports are connected to physical machines: *PHY0*, *PHY1*, *PHY2* and *PHY3*. The second device is an Intel Cyclone V-based FPGA. The FPGA runs the accelerated network coding encoder and decoder entities. The physical layer lab setup is shown in appendix 8.2.

3.4.1 Intel NIC

The Intel PRO/1000 PT quad port NIC is chosen because it is relatively inexpensive and provides compatibility with DPDK. Therefore multiple NICs could be used to interface many physical hosts together if the PC supports multiple PCIe interfaces. While other NICs provide additional features, the implementation does not use them and therefore the Intel PRO/1000 is more than sufficient. The physical machines connected to the NIC are all Linux-based and can run Linux network testing tools such as iperf, nmap and tcpdump.

3.4.2 FPGA

The hardware encoder and decoder entities are run on the FPGA. Network traffic is sent to TUN/TAP interfaces in TAP mode. TAP interfaces offload the packets onto the FPGA. Further detail regarding the implementation of the coding entities is provided in chapter 5.

The FPGA device chosen is the OpenVINO starter kit from Terasic. The development board uses an Intel Cyclone V 5CGXFC9 FPGA and supports four lane PCIe. The OpenVINO starter kit is chosen because it was one of the least expensive PCIe-based development boards, with an Intel FPGA commercially available at the time. The resources provided by the Cyclone V are greater than required to run the encoders or decoders from Chapter 5.

The "setup_fpga.sh" script is used to load the PCIe FPGA driver and add the TAP interfaces to OvS. TAP kernel interfaces are used to connect the FPGA to OvS through a userspace application. While the literature review in Chapter 2 indicates that a kernel-bypass packet processing technique provides increased performance, the PCIe application provided by Terasic makes use of DMA and therefore using DPDK would not leverage any benefits and a more simple interface of TAP is used instead. However, in future implementations without using the provided library from Terasic, DPDK could be used to implement the userspace networking for the FPGA and OvS interface. Details explaining the interface software is provided in Chapter 6.

3.5 Virtual layer

On the virtualization side, QEMU is used along with KVM as the hypervisor to run VMs. Two instances of VMs are used in the system. The one instance is to run standard Linux-based machines: *VM0*, *VM1*, *VM2*, *VM3*. The second instance are the encoder and decoder functions as dedicated VNFs. DPDK is used along with the Kodo library to implement the network coding functions in the C programming language.

3.5.1 VNFs

The VNFs are created using Ubuntu 18.04 server Linux VMs. DPDK is run on the VM to add network coding packet processing. Further details on the implementation of the RLNC VNFs are given in Chapter 4.

A single DPDK application "dpdk_coder" with both the encoder and decoder functions is created. The user can choose to run the the coder in *encode*, *decoder* or *nocode* modes. Encode and decode mode call the RLNC encoder and decoder functions respectively. The nocode node simply forwards the packets through the VNF without and coding operations. Nocode mode is used as a baseline to evaluate the time taken for packets to traverse through the VM. Source code for the DPDK coding application can be found on GitHub [58].

3.6 Summary

This chapter provides the system overview, and how the various system components are integrated to perform the task of a network coding-capable switch. The system is constructed using three main components: a networking bridge, a physical layer and a virtual layer. OvS is chosen as the networking bridge and provides all the network switching functionality to integrate the virtual and physical layers. Details regarding the port configuration are provided.

The physical layer consists of the Intel PRO/1000 PT quad port NIC used to connect to physical machines, and the OpenVINO FPGA device to perform hardware-based network coding. The virtual layer consists of the QEMU/KVM hypervisor to run VMs and VNFs. The VNFs are implemented using the DPDK packet processing library.

Chapter 4

Network coding in the virtual layer: Virtual Network Functions

4.1 Introduction

In the previous chapter we explained the setup and configuration of the system to support both virtual and physical network interfaces. In this chapter we discuss how network coding is implemented as virtual network functions. We implement the software component of the network coding functions as discussed in section 2.5. The software-based coding functions are created using the DPDK packet processing library and implemented as VNF VMs. We discuss the components required to create the software functions, as well as the steps taken to implement the components.

4.2 Methodology

We begin the software component design by describing the required components. Once the components are established, we begin the process of constructing the components and detail the process used to do so. The software construction is implemented in three steps. First a Python-based implementation is created to understand and verify the functionality of the Kodo library. Next, DPDK is used to construct the networking layers. The networking functionality is done to a point where packets can be received, processed and transmitted using DPDK-based interfaces. Then finally, we combine the Python Kodo functionality with the DPDK networking layers to create the C-based Kodo RLNC functions. The functions are implemented in VMs as VNFs.

4.3 Component overview

In order to implement the software based network coding functions, the following components are required:

- A component performing network coding functionality to process packet data using the RLNC algorithm. The network coding functionality is implemented using the Kodo software library.
- Networking functionality to interface the network coding functions with the rest of the (real packet-based) network. DPDK is used to implement this functionality and run the software based coding functions as VNFs.

The discussion continues onto the process followed to construct the software coding components. We begin with the Python-based coding implementation, to investigate and design a RLNC algorithm using Kodo. The RLNC algorithm obtained from the Python-based implementation is rewritten in C in the final DPDK based VNF.

4.4 Kodo Python baseline

The first step in creating a network coding function is to establish a functionality *baseline*. The functionality baseline is a working implementation of network coding that showcases the correct encoding and decoding results. The baseline is used as a reference to determine that the VNF-based coders are producing the correct outputs.

The functionality baseline is created using the Kodo-Python RLNC functions [59]. Python is chosen to create the baseline application over C due to its ease of use and script-ability. Furthermore, the Python and C Kodo libraries are simply wrappers of Kodo's C++ functions. The functionality of the Python and C Kodo libraries are therefore the same, allowing for the baseline application to be rewritten in C with DPDK, at a later stage.

The flow diagram of the Python implementation is shown in Fig. 4.1. First the coding parameters: field size m, number of packets h and packet size N are declared. The coding parameters are input to the encoder and decoder factory functions to create the coder objects. Next the input and coefficient data lists are declared and populated. The encoder and decoder objects are assigned the declared data lists. The coding process begins by encoding each packet and immediately decoding the encoded result. The decoded result is then compared to the input data list. If the input data is equal to the decoded data, then the coding result is successful. The software code used in the Python implementation is provided in Appendix 8.2.

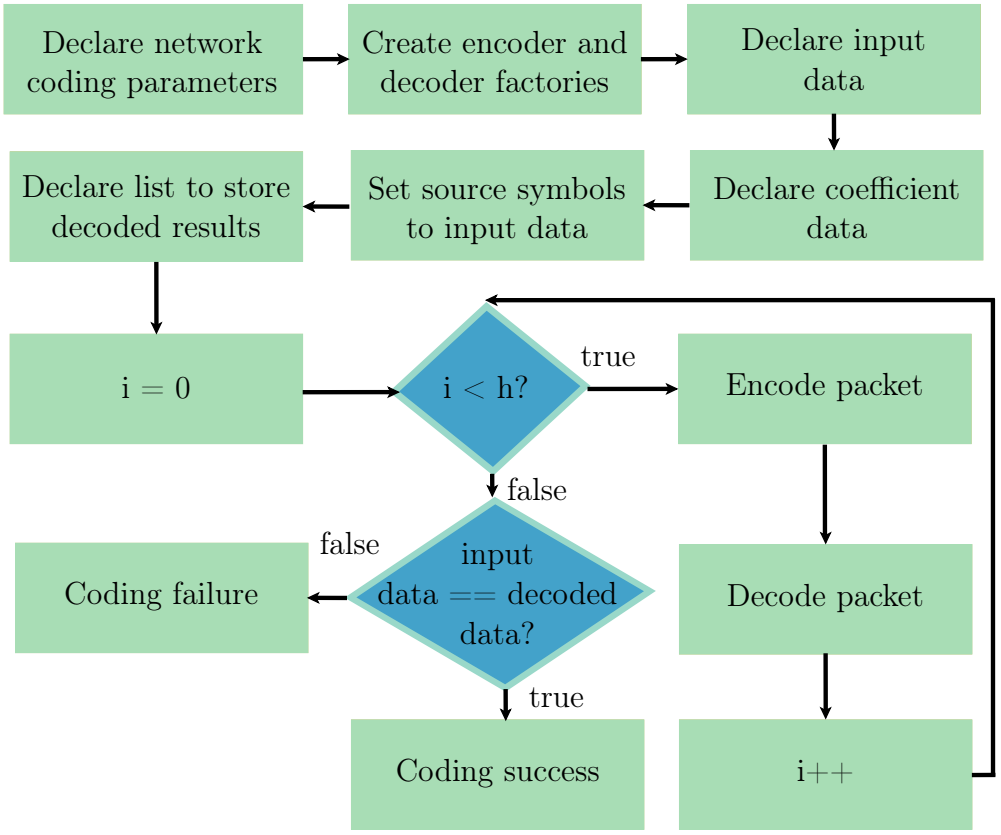

Figure 4.1: Python implementation flow diagram. The coding loop consists of encoding each packet and immediately decoding the encoded result. The loop continues until each packet in the generation of packet size h has been decoded successfully. If the decoded results equal the input data then the coding process is successful.

The Python baseline performs RLNC without any networking functionality. Instead of using data received from a network interface, coding operations are performed on fixed input and coefficient data. The input data refers to the un-coded source data. The input data and coefficient data formats are shown in Figs. 4.2 and 4.3 respectively.

The source data from Fig. 4.2 consists of h data packets of N bytes each. These packets are grouped into a generation of size hN bytes. A coefficient data vector of length h bytes is required for each of the h source data packets. The coefficient data vectors are grouped together in Fig. 4.3 to form a coefficient data matrix of size h^2 bytes. The coding coefficients and input data are represented as hard-coded lists in the Python implementation.

The Python implementation outputs the encoded data during the encoding process. The encoded data along with the un-coded and coefficient data is used as a reference to design the VNF-based coders. Specifically, the encoded data is

used to check if the VNF encoders are producing the correct output. The VNF decoder output is also checked by using the Python implementation encoded data as an input.

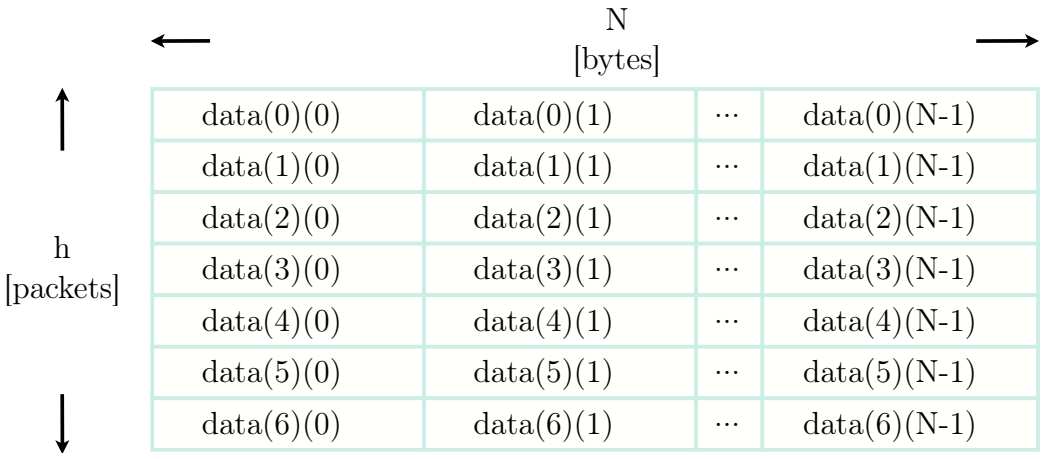

Figure 4.2: Kodo input data format. Source input data is grouped into a generation of size hN bytes.

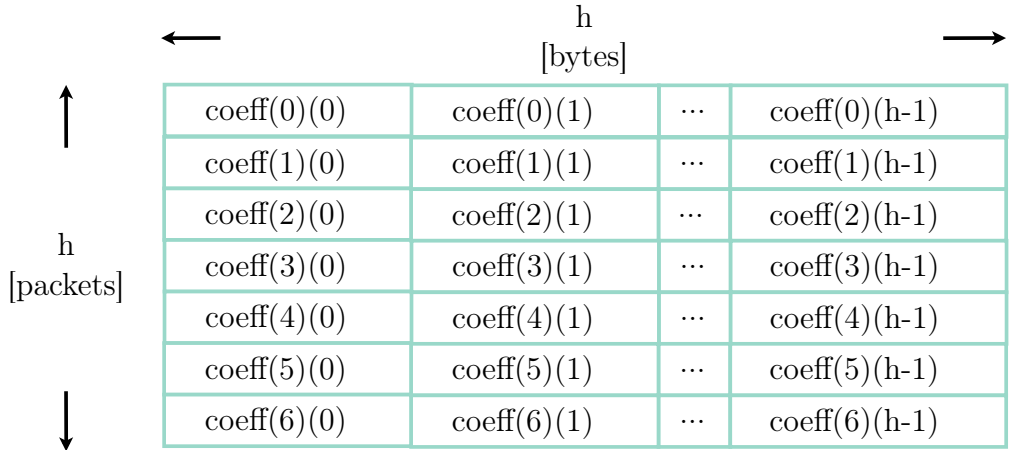

Figure 4.3: Kodo coding coefficient data format. All the coefficients are grouped into a matrix of h^2 bytes.

The source, coefficient and encoded data are kept constant throughout this thesis. The data values are randomly generated once and then kept for all implementations, including the VNF and FPGA coders. This is crucial as it ensures a fair testing environment across all coding implementations. The values used are listed in appendix 8.2.

4.5 DPDK networking layer

The networking functionality is constructed using the DPDK 18.11 software library. DPDK is installed on a VM running the Ubuntu 18.04 Linux OS. Hugepages are allocated at boot for packet buffering, as required by DPDK. The amount of hugepage memory to allocate is taken from the blog post of Trayner in [60].

The blog article calculates 4096 MB of required hugepages for using OvS-DPDK, with both physical and virtual maximum transmission unit (MTU) sizes of 1500 and 9000 bytes respectively. While the packet sizes used in this thesis are limited to 64 bytes, to accommodate for future work, the same size of hugepages is used.

A single network interface port is used to receive and transmit packet data. The network interface of the VM is assigned to be controlled by the DPDK PMD using a configuration script on start-up. The network link is first unbound from the Linux networking stack and then bound to the DPDK with the *uio_pci_generic* kernel driver. The setup script is given in appendix 8.2.

The main goal of the networking layer is to perform packet processing, i.e. specifically to receive data, perform network coding and then transmit the result. The networking layer is created as a C application. The implementation of the overall DPDK networking layer is done systematically.

4.5.1 Receive and transmit loopback

The first step in developing the DPDK networking layer is to create a loopback application. In a communication system, loopback is considered to be a primary way of testing. Loopback involves transmitting a packet received by the network interface without performing any processing. This confirms that the receiving and transmitting functions are operating as expected.

The loopback application requires a means of receiving and transmitting packets in the userspace DPDK application. DPDK provides many APIs that allow for the network interface to be controlled from the userspace. The Ethernet device API enables access to the receive and transmit file descriptors of the networking interface port [11].

4.5.2 DPDK interface configuration

While the network interface has been bound to DPDK, it needs to be configured in the userspace application, in order to receive and transmit packets. The DPDK network interface requires receive and transmit queues to buffer incoming and outgoing packet data. The setup procedure is shown in Fig. 4.4. The portID is set to be “0” because there is only one DPDK interface on the VNF. The portID is used by all proceeding functions to modify the network descriptors.

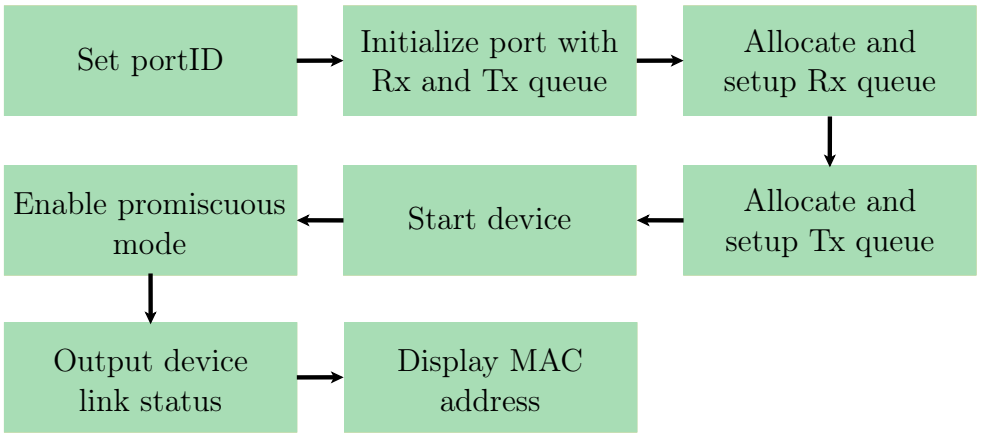

Figure 4.4: DPDK Ethernet interface configuration procedure.

The Ethernet device is initialised with a single receive and transmit queue because only a single logical core is assigned to the DPDK port. Both queues are setup and allocated memory. The device is then started and promiscuous mode is enabled. Promiscuous mode allows all packets received on the port to be sent to the DPDK application, and not just packets sent to the MAC address specifically. It is important to enable promiscuous mode because packets scheduled for coding have MAC addresses relative to their intended destination and not the coding VNF. Therefore, the packets would normally be ignored without promiscuous mode.

One thing to note, is that DPDK takes control of the Ethernet interface. The interface is no longer listed under Linux utilities such as *ip* and the older *ifconfig*. The DPDK configuration procedure therefore concludes by showing device link status and MAC address. The configuration and networking details can be confirmed as a result.

4.5.3 Receiving packets

Once the Ethernet interface is configured the application can begin network communication. The *rte_eth_rx_burst()* function from the RTE Ethernet Device API is used to receive packets:

```
static uint16_t rte_eth_rx_burst(
          uint16_t port_id,
          uint16_t queue_id,
          struct rte_mbuf **rx_pkts,
          const uint16_t  nb_pkts )
```

The function takes four parameters and outputs the number of packets received. The input parameters, are the port identifier, receive queue index,

address to store received data and number of packets to receive. The port and queue IDs are set to "0" because there is only one network interface, with a single receive queue. The received packets are stored in packet buffers.

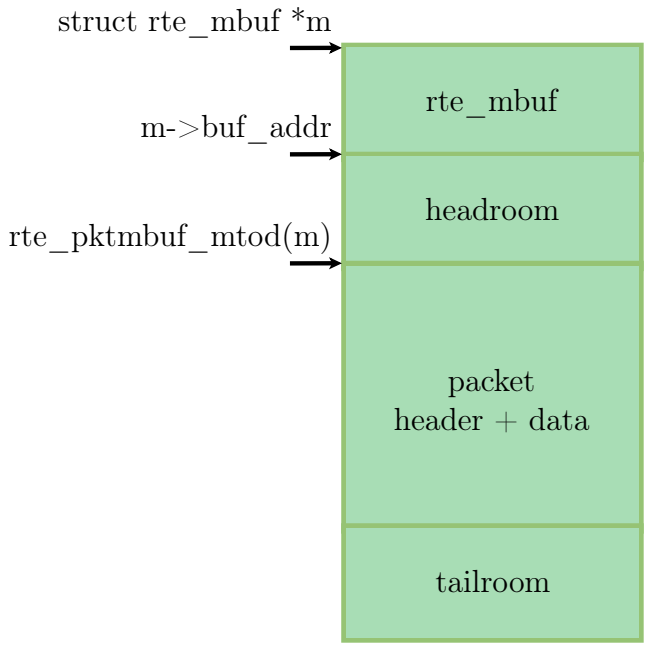

Figure 4.5: DPDK memory buffer layout.

An *rte_mbuf* struct is used by the DPDK application to store network packet data. The Mbuf library is used to manage and configure the memory buffers. Fig. 4.5 shows the layout of the mbuf buffer. The actual packet header and data received by the network interface are encapsulated within DPDK metadata. The *rte_pktmbuf_mtod()* function is called on the mbuf to return the location of the actual packet header and data. Packet processing is done at this location, and without the DPDK metadata.

The buffers are stored in a single *mempool*, managed by the Mempool library. The *rte_pktmbuf_pool_create()* function is called to create the mempool struct. Each mempool uses a unique name and makes use of ring data types to store the mbufs. A new mbuf is allocated using the *rte_pktmbuf_alloc()* function, which returns a pointer to the new mbuf.

4.5.4 Transmitting packets

To complete the loopback function, the received packets are transmitted out of the same port. Packets are transmitted using the *rte_eth_tx_burst()* function:

```
static uint16_t rte_eth_tx_burst(
          uint16_t  port_id,
          uint16_t  queue_id,
          struct rte_mbuf ** tx_pkts,
          uint16_t  nb_pkts )
```

The port and queue ID are set to be the same as the receive function, "0". A mbuf pointer to the packet data to be transmitted is given. In the loopback case, the received packet data is used.

4.5.5 Loopback function verification

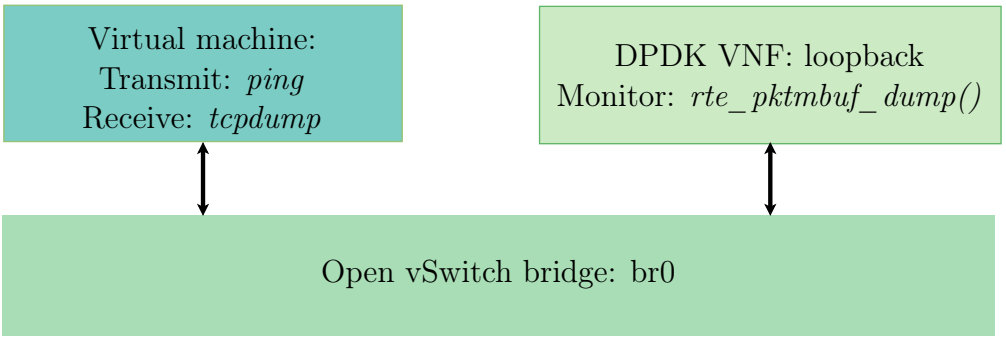

Figure 4.6: Setup used to verify DPDK networking loopback function.

To verify that the loopback is functioning correctly, packets are sent from one of the non-coding VMs, and the loopback results are sniffed. The *ping* and *tcpdump* Linux network utilities are used to send packets and inspect packets on the non-coding VM respectively. The utilities cannot be run on the VNF because the network interface is bound to DPDK. Instead, the *rte_pktmbuf_dump()* function is called on the received packet to dump the mbuf structure to *stdout*.

A summary of the verification setup is shown in Fig. 4.6. A packet is sent using ping from the VM and tcpdump is run to inspect the received packet. If the same packet is received then it confirms that DPDK loopback is working correctly.

The next step is to begin adding packet processing as network coding. Before the coding functions are created, a decision making process is designed to decide what to do with each of the received packets. We therefore continue the discussion onto the coding function selection process.

4.5.6 Coding function selection process

A coding function selection process is created that determines when to perform network coding operations. The tasks of encoding, decoding and forwarding need to be applied as necessary when packets are received. For multiple network coders to exist within a network, a selection process is established to maintain co-ordination.

To incorporate network coding traffic into the network, two changes are proposed. The first is the use of a unique Ethernet type 0x2020. The Ethernet type is used by switches or coding nodes to differentiate between coded and un-coded packets. An unlisted Ethernet type is used from the IEEE 802 standard to prevent conflicts between other networking protocols [61].

The second change is the addition of a *coding capable* field to the MAC forwarding table. An output port is flagged as coding capable if the switch has received a network coded packet through that port. The assumption is made that nodes capable of encoding can also decode RLNC packets. If an output port is coding capable then the switch can send encoded packets through the respective port. If the port is not coding capable, then the switch must not forward encoded packets to that port because there is no guarantee that they will be decoded.

The flow diagram in Fig. 4.7 shows the selection process used by the coding pipeline. Packets are received on the ingress queue of the network port. The coding pipeline reads and stores the Ethernet type and destination address of the packet. The destination address is found in the MAC table to determine if the port is coding capable. If the port is coding capable and the Ethernet type does not equal 0x2020, then the packet should be encoded. If the port is not coding capable and the packet is encoded or un-coded, then the packet is decoded or forwarded respectively.

Another scenario is when the destination address is coding capable and the packet type is already 0x2020. This implies that an encoded packet is being sent to a destination that can perform coding functions. The packet is then recoded. In the implementation in this thesis, however, no recoding function is created. While recoding in network coding is almost identical to the encoding process, this implementation is left for future work. With the current VNF implementation, the packet is therefore dropped.

The coding selection process does not run on the VNF coder, but on the SDN controller application. The DPDK coding function however is designed to be compatible with the selection process. Therefore the selection process is introduced here, and the controller implementation is given later in Chapter 6.

A separate coding pipeline is designed for the encoder and decoder respectively. The aim of the coding pipelines are to prepare the received packet data, and resultant processed data for each of the coding functions. We continue the discussion onto the encoder pipeline.

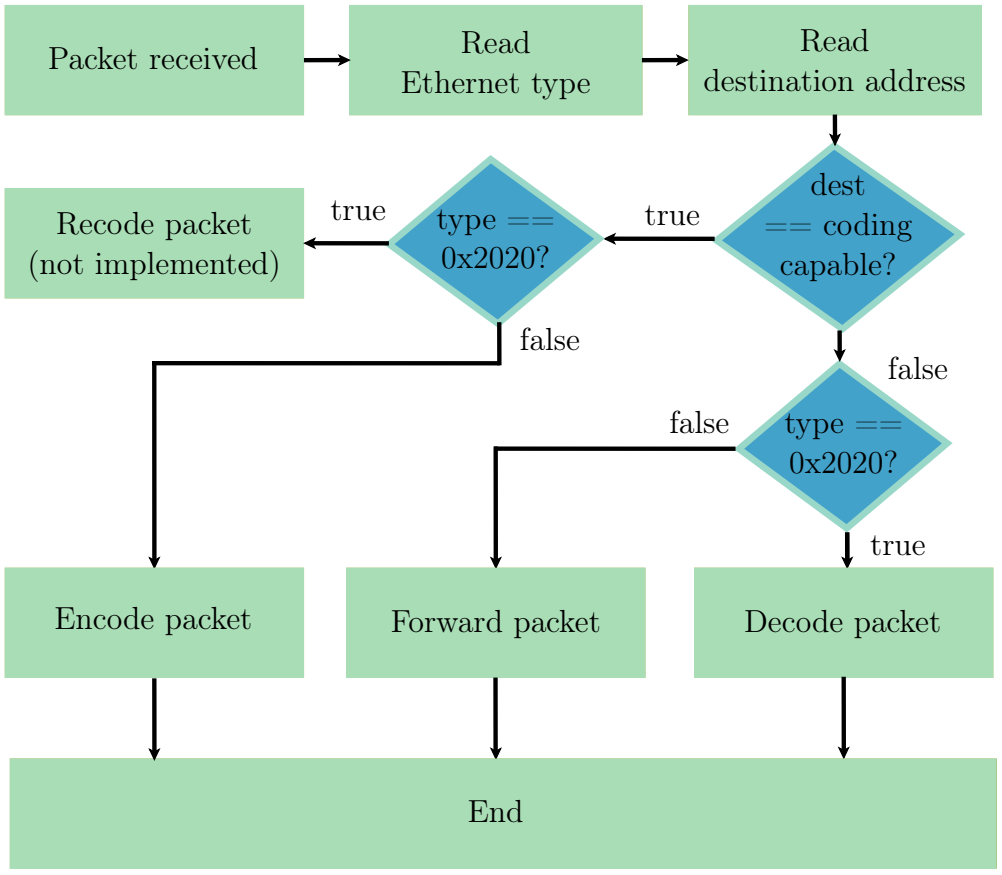

Figure 4.7: Selection process used by DPDK application to determine coding operations.

4.6 DPDK encoding pipeline

The encoder groups packets into a generation, based on the destination address from the Ethernet header. Once all the packet have been received the network encoding process is executed. The generation size is set to be the same as with the Python baseline, $h = 7$ packets. There is however, no algorithmic restriction on the generation size and packet size, and these values can be easily modified without rewriting the coding function. This is done to aid with the objective of keeping the implementation flexible for future work.

In a real network, incoming packets are not always received synchronously from a source. Scenarios occur where multiple senders are sending packets through the switch, and as a result packets may arrive in a jumbled order. To keep track of which packets belong to each generation, the encoder makes use of a MAC table and creates an encoding ring for each MAC table entry. The encoding ring stores incoming packets for each generation as they are sent to the VNF.

The encoding ring is created using the DPDK ring library. Rings are fixed sized, First In First Out (FIFO) data structures. Mempools that are used to store mbufs also make use of rings, and is the preferred way of storing packet data in DPDK [11]. Rings provide advantages such as a lockless implementation, and bulk enqueue and dequeue operations.

The structure of the MAC table and encoding rings are as follows,

```
#define MAC_ENTRIES 200
#define ENCODING_RINGS 128

/* Encoding rings array: Used to store all encoding rings based
   on dst_addr*/
struct rte_ring encoding_rings[ENCODING_RINGS];

static unsigned mac_counter = 0;

/* Mac Fwd TABLE to group generations by dst_addr */
struct mac_table_entry {
    struct ether_addr d_addr;
};
struct mac_table_entry *mac_fwd_table;

mac_fwd_table = (struct mac_table_entry*)calloc(MAC_ENTRIES,
   MAC_ENTRIES * sizeof(struct mac_table_entry));
```

The number of MAC entries in the MAC table and number of encoding rings can be set to any value, and are dependant on network traffic and coding speeds. The values are set high enough to prevent packets from getting dropped.

The MAC table is created as an array of MAC entry structs. The format is different to the traditional MAC forwarding table of a destination address and next hop, described in Chapter 2. Instead, the MAC table stores the destination address of the packets in the generation. This is used as a reference to determine if incoming packets need to be added to an existing encoding ring, or if a new ring should be created. The MAC table is therefore not used as a forwarding table, because all encoded packets are transmitted out again through the same port.

A flow diagram of the DPDK encoding pipeline is shown in Fig. 4.8.

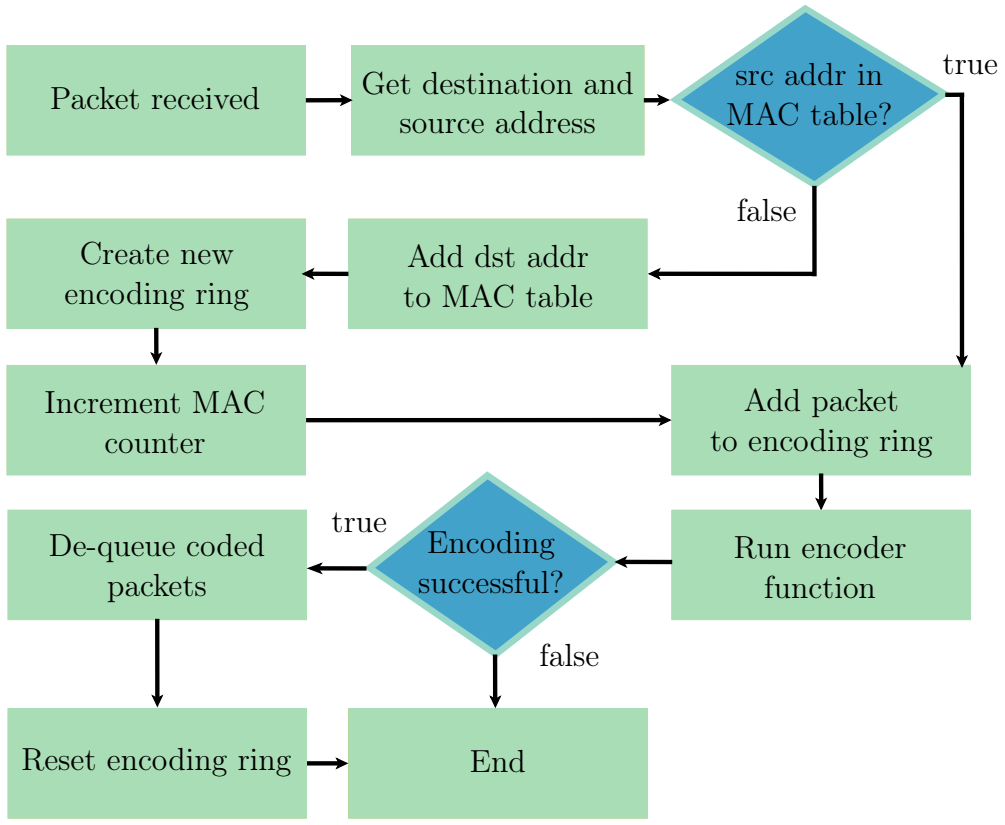

Figure 4.8: DPDK encoding pipeline flow diagram.

The process begins by receiving the packet and obtaining the destination and source MAC addresses. If the source address does not exist in the table, the destination address is added to the table. A new encoding ring is created using the *rte_ring_create()* function:

```
struct rte_ring* rte_ring_create(
          const char *  name,
          unsigned  count,
          int  socket_id,
          unsigned  flags
)
```

The name of the ring is set to *encoding_ringX* where X is the mac counter value used to keep track of how many MAC entries are in the table. The size of the encoding ring is defined by the *count* parameter and the socket ID is set to the default value of using any socket (In this case only one is available to the DPDK application).

The *flags* parameter is used to set single or multiple consumers and producers. Producers add objects to the ring, while consumers remove objects. Multiple producers or consumers are used when multiple processes want to

access the ring. In this implementation however, only a single producer and consumer are used as a single process application is used.

Kodo-C Function	**Description**
kodoc_factory_build_coder()	Initialises the encoder or decoder for network coding, based on the factory type.
kodoc_block_size()	Returns the block size of the encoder or decoder.
kodoc_rank()	Returns the current rank of the encoder or decoder. This provides an indicator of how many symbols have been encoded or decoded.
kodoc_delete_coder()	Deletes the encoder or decoder instance. This is done after every generation.
kodoc_delete_factory()	Deletes the encoder or decoder factory. This is done when the program ends.

Table 4.1: Kodo-C network coding functions: general

Kodo-C Function	**Description**
kodoc_new_encoder_factory()	Creates a new network encoder factory. The encoder factory is used to initialise encoder instances, using set coding parameters.
kodoc_set_systematic_off()	Sets systematic coding to off. This prevents the encoder from sending the original source symbols first.
kodoc_set_const_symbols()	Specifies the source data to be used for all symbols. The source data is set to the incoming un-coded data.
kodoc_write_payload()	Writes a symbol to the payload buffer. This is the encoded data.
kodoc_payload_size()	Returns the size of the generated, encoded payload. This is used to get the payload size when creating the output packets.

Table 4.2: Kodo-C network coding functions: encoder

After the new encoding ring is created, or if the source address is in the MAC table, the packet is added to the ring using the *rte_ring_enqueue()* function. The encoder function is then called, and if encoding is successful (and all packets dequeued), the ring is reset. The first packet from each generation will always take slightly longer to process than the remaining packets. This is something to take into consideration for smaller generation sizes.

The encoding function called during the encoding pipeline process is discussed next. The Python-based coding implementation is rewritten in DPDK using the Kodo RLNC-C library [62]. A summary of relevant, general Kodo functions and those used by the encoder is shown in tables 4.1 and 4.2 respectively.

4.7 DPDK encoding function

The DPDK-based encoding function flow diagram is shown in Fig. 4.9. The encoder function is called whenever a packet is added to an encoding queue. The function loops through each encoding ring and checks if the ring is full. If the ring is full, then there are a generations size h packets in the ring and the encoding process begins. The packets to be encoded are dequeued in bulk from the full encoding ring using the *rte_ring_dequeue_bulk()* function. The packets are processed individually from the entire block of the encoding ring, because the packet data must first be extracted from each packet. Once the packets are dequeued, they can be used by the encoder.

A Kodo RLNC encoder is created using the *kodoc_factory_build_coder()* function. Systematic coding is turned off using the *kodoc_set_systematic_off()* function. This is important because with systematic coding turned on, the encoder first outputs the original packets, and then the encoded packets. The implementation only requires the encoded packets and therefore systematic coding needs to be disabled.

A new mbuf is created to store the data to be encoded, as well as for the encoded result data. Next the input data is assigned to the encoder. The payload as well as the Ethernet type are to be encoded. This is because the encoded packet Ethernet type is set to 0x2020 to signal that it is an encoded packet. The original Ethernet type needs to be preserved and is therefore encoded along with the payload data.

A *generation ID* is created for the packets in the generation. The generation ID is appended to the encoded resultant packets, and is used by the decoder function to group packets for decoding. The generation ID is randomly generated using the *random()* function from character values in the range $A, .., Z$.

Once the generation ID is created, each of the packets in the encoding ring are encoded in a loop, and the resultant packet is transmitted out. The

resultant encoded packet format is shown in Fig. 4.10, for a generation size of $h = 7$ and packet size of $N = 64$.

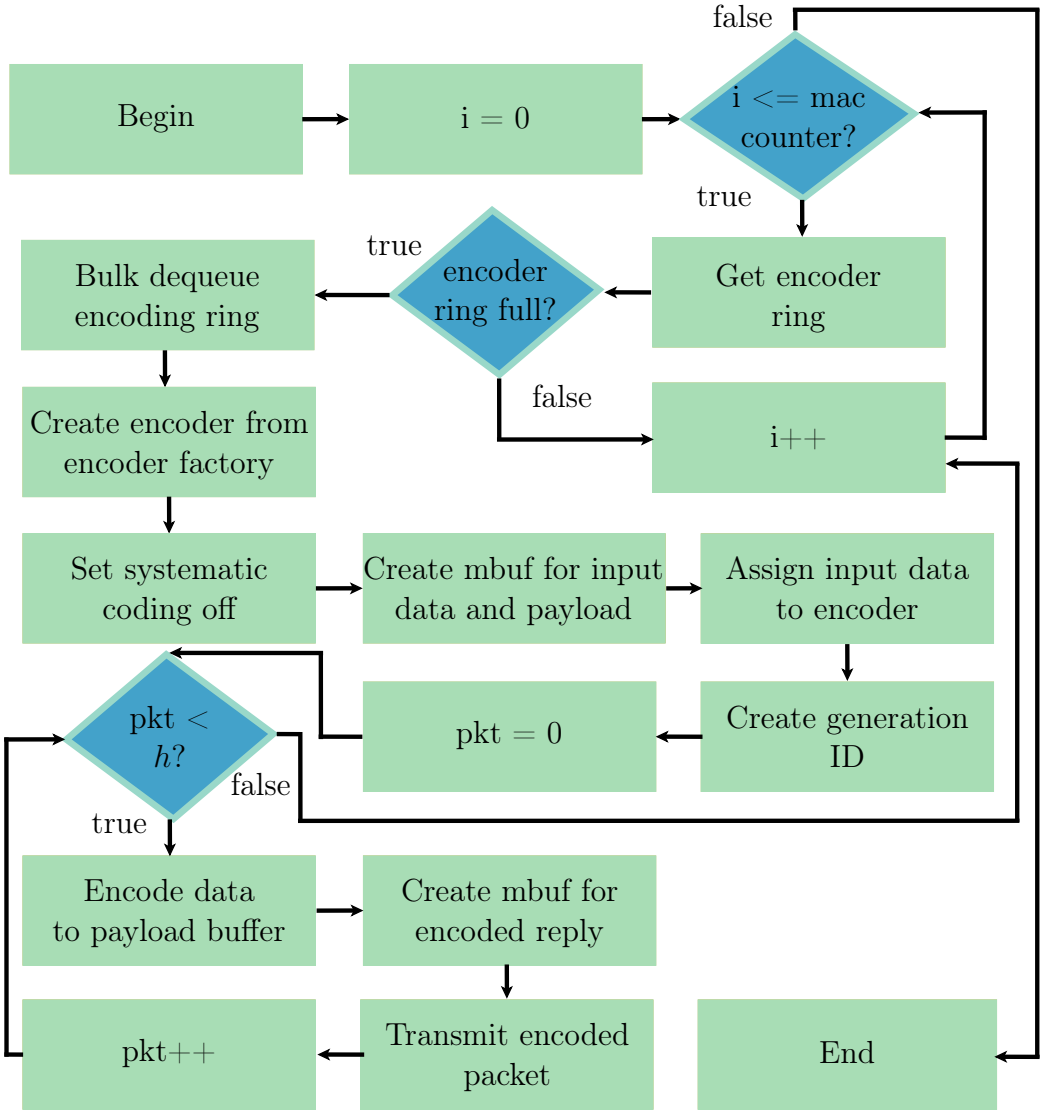

Figure 4.9: DPDK encoder function flow diagram.

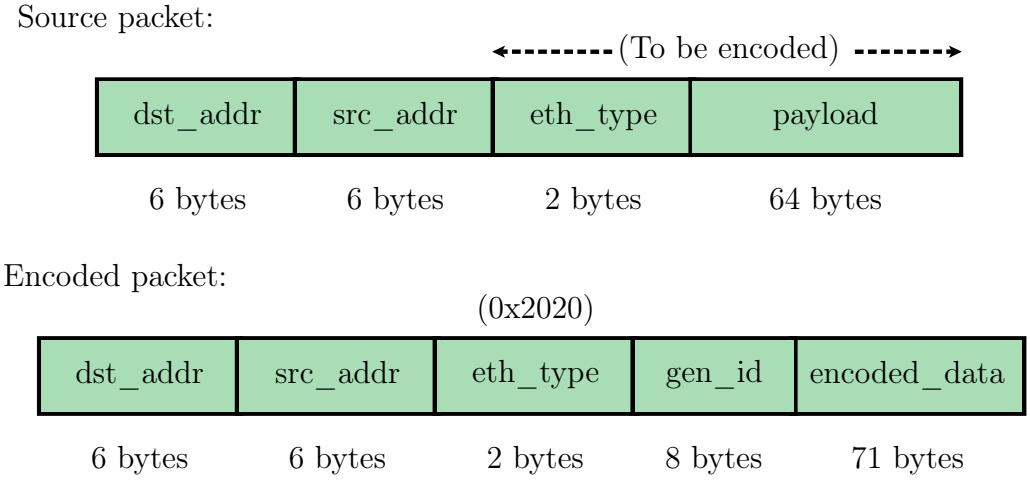

Figure 4.10: Source and encoded packet formats. The section of the source packet to be encoded is shown, as well as the 0x2020 Ethernet type used by the encoded packet.

4.8 DPDK decoding pipeline

Similarly with the encoder, a decoding pipeline is created for the decoder function. Incoming encoded packets are grouped by generation ID. Once h packets are received with the same generation ID, the decoding process begins. To keep track of all the packets in each generation, the decoder uses a generation ID table. A decoding ring is created to store all the incoming encoded packets, for each generation ID. The structure of the generation ID table is as follows,

```
#define GENID_LEN 8
static unsigned genIDcounter = 0;
struct generationID {
    char ID[GENID_LEN];
};
/* GenerationID table: Used to store list of all generations
   still being decoded. */
struct generationID *genID_table;

genID_table = (struct generationID*)calloc(MAC_ENTRIES,
   MAC_ENTRIES * sizeof(struct generationID));
```

The generation ID table is created as an array of *generationID* structs. Each generationID struct contains a char array as the ID.

The flow diagram of the DPDK decoding pipeline is shown in Fig. 4.11. The process begins by receiving the encoded packet and obtaining the generation ID. The generation ID is checked to be valid by looking at the length,

and if the ID is comprised of alphabetic characters. If the generation ID is not valid the packet is dropped. If the ID is valid, then it is looked up in the generation ID table.

The first packet received by the decoding process from a generation will not have the generation ID in the table. The decoding process adds the ID to the table and creates a new decoding ring using the *rte_ring_ create()* function. The parameter values are the same as with the encoding rings, except that the name is set to the generation ID.

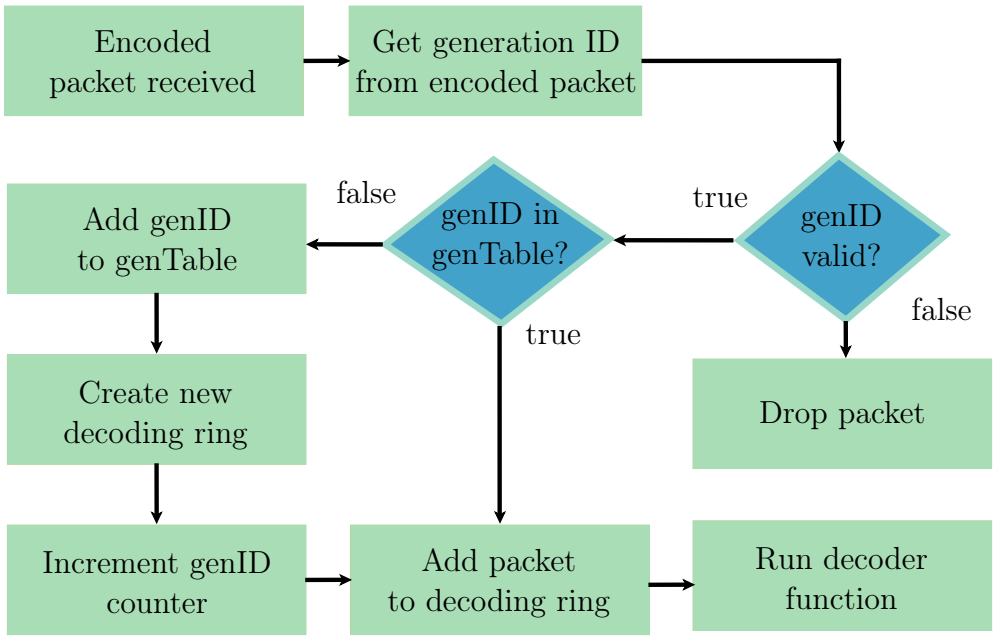

Figure 4.11: DPDK decoding pipeline flow diagram.

Once the new decoding ring is created, or if packet with an existing generation ID is received, the packet is added to the corresponding decoding ring. The decoder function is then called.

4.9 DPDK decoding function

The Kodo RLNC-C library is used to create the decoding function from the Python baseline implementation, in the DPDK environment. A summary of the relevant Kodo-C functions used to implement the decoder are given in table 4.3.

The DPDK-based decoding function flow diagram is shown in Fig. 4.12. The process begins by looping through each decoding ring in the generation ID table. If a decoding ring is full, then the decoding begins.

Kodo-C Function	**Description**
kodoc_new_decoder_factory()	Creates a new network decoder factory. The decoder factory is used to initialise decoder instances, using set coding parameters.
kodoc_set_mutable_symbols()	Specifies the data buffer where the decoder should store the decoded symbols.
kodoc_is_complete()	Checks the decoder to see if decoding has been completed. This is used in a while-loop to continue the decoding process until all packets are decoded.
kodoc_read_payload()	Passes the encoded payload to the decoder. This is called during the decoder loop.

Table 4.3: Kodo-C network coding functions: decoder

The encoded packets are dequeued in bulk from the decoding ring. Next, a Kodo RLNC decoder is created using the *kodoc_factory_build_coder()* function. A mbuf is created to store the resultant decoded data and the decoding loop is started. The loop cycles through each packet from the decoding ring, assigns the data to the decoder and decodes the data.

Once the decoding process is completed, the resultant un-coded packets are packetized for transmission. The original Ethernet type, that was encoded, is recovered and appended to the resultant packet. The encoding Ethernet type 0x2020 is removed from the packet and the result is identical to the original source packet from 4.10. After the packets are transmitted, the generation ID is removed from the generation ID table, and the decoding ring is freed.

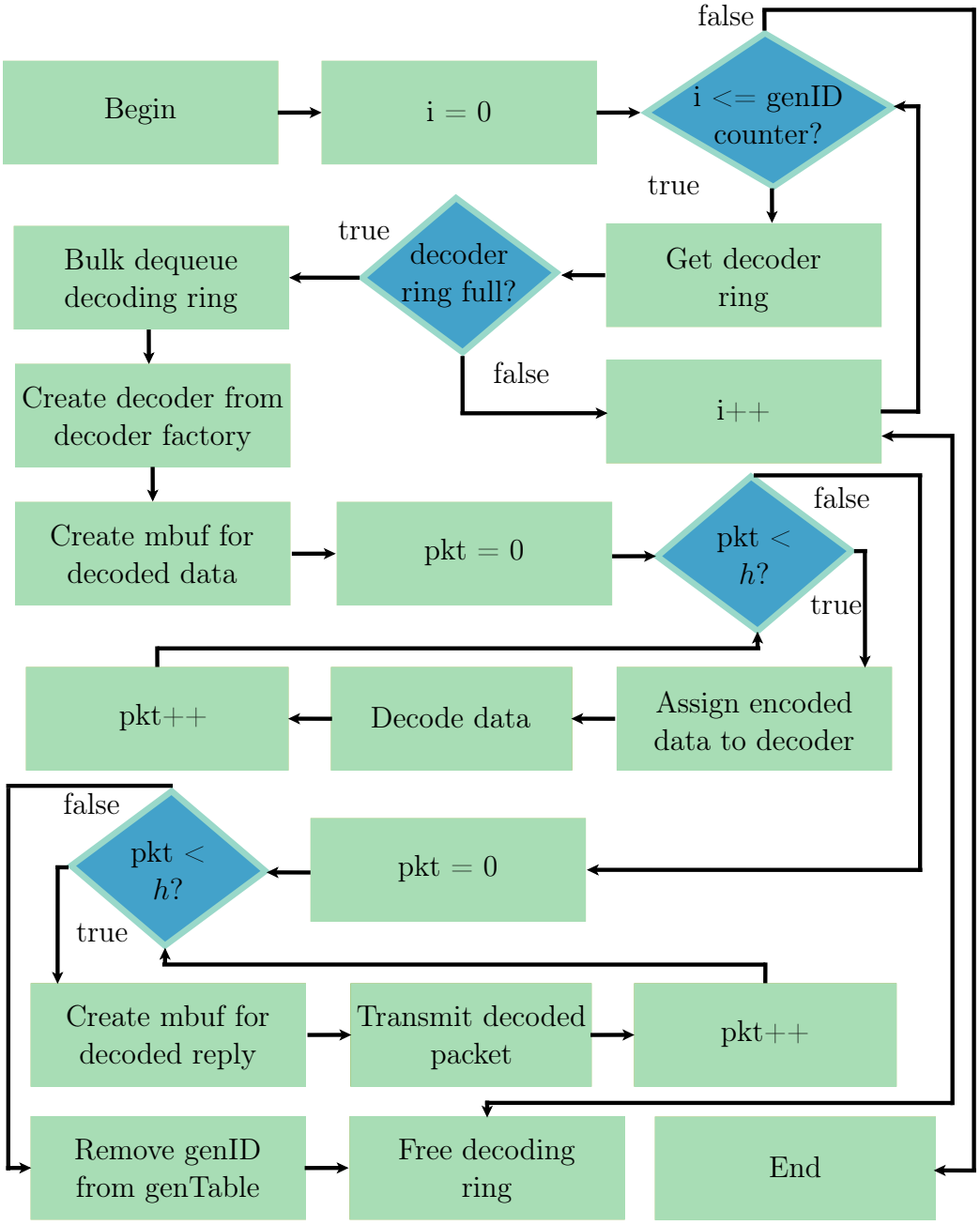

Figure 4.12: DPDK decoder function flow diagram.

4.10 Summary

This chapter discussed the design and implementation of the software-based VNF network coding functions. The discussion begins with the methodology used to design the software coding functions. The design process is abstracted into three steps: a Kodo Python baseline implementation, the DPDK network-

ing layers, and C-based Kodo RLNC functions incorporated into the DPDK networking layers.

The Kodo functions are implemented in Python to establish a functionality baseline for the network coding functions. The functionality baseline is used to showcase the correct encoding and decoding results. The baseline application is used as reference for the complete VNF and hardware implementations to determine that they are functioning correctly.

Once the Python baseline is complete, a DPDK loopback application is created to develop and confirm the DPDK networking layer functionality. The networking layer is constructed to perform packet processing. Specifically to receive data, perform network coding and then transmit the result.

A coding function selection process is presented to determine when to perform network coding operations. The coding selection process provides the use of a unique Ethernet type 0x2020 to differentiate between coded and un-coded packets within a network. The addition of a coding capable field to the MAC table is also proposed. The coding capable field is used to flag a network port that has received an encoded packet in the past. The switch will only forward encoded packets packets to a coding capable port to ensure that the packets can be successfully decoded.

The DPDK coding functions are designed to be compatible with the coding selection process. A separate coding pipeline is designed for the encoder and the decoder. The coding pipelines prepare received packet data and resultant processed data for each of the encoder and decoder coding functions.

The DPDK encoding pipeline groups incoming packets in a generation based on the packet destination address. The encoding process begins once all h packets have arrived. The encoder uses a MAC-based table to keep track of all the packets that belong to each generation. The encoding function is created using the Kodo RLNC functions. A generation ID is created by the encoder for all the packets in the generation. The generation ID is used by the decoder function to group packets for decoding.

The DPDK decoding pipeline groups incoming encoded packets by generation ID and begins decoding once h packets have arrived. The decoder uses a generation ID table to keep track of all the packets that belong to each generation. The Kodo RLNC library is also used to create the decoding function.

Chapter 5

Network coding in the hardware layer: Field Programmable Gate Array

> *"Later there will be, I hope, some people who will find it to their advantage to decipher all this mess."*
>
> — Evariste Galois, *May 29, 1832*

5.1 Introduction

In the previous chapter we discussed the design and implementation of network coding in the virtual layer, using virtual network functions. In this chapter we discuss how network coding is implemented in the physical layer, in FPGA hardware.

5.2 Methodology

A bottom-up design approach is used to implement RLNC coding modules for the Terasic OpenVINO Starter kit board, using Intel's Quartus Prime 18.1.0 lite edition design suite. Very High Speed Integrated Circuit Hardware Description Language (VHSIC-HDL) (VHDL) is used to implement the coders, and Verilog is used to interface the coding modules to the PCIe IP core. VHDL is used due to familiarity and previous experience, while Teraisc provides the PCIe interface in Verilog. The discussion on how the coding modules are interfaced with PCIe is done in chapter 6. The source code for the coding modules can be found on GitHub [57].

The main design priorities are functionality, modularity and adaptability. Functionality ensures that the hardware fulfils the required objectives and intentions. Specifically, being able to perform network encoding and decoding, using the RLNC algorithm. A proof of concept is created, that demonstrates the possibility and feasibility of successful network coding. The design is kept as modular as possible to facilitate any further expansion and optimization.

This ensures that the modules are adaptable for any future work and enables sub-component optimization, without a complete architectural change.

The encoder and decoder are created using a variety of sub-components. Some of these sub-components are shared, while others are specific to the encoder and decoder respectively. The sub-components are designed to be adapted for different constants. These include, Galois field size, generation size and packet size. However, within this thesis, they are constrained to values of eight bits, seven packets and 64 bytes respectively. This enables a seven by seven coding coefficient matrix to be used in hardware. These values are chosen to correspond with the software implementation of network coding as discussed in chapter 4.

Design objectives such as resource utilization and power consumption are taken into consideration. These however, are only to provide a suitable implementation, and are not the main design focus. If an embedded design is required, then these objectives would need to be considered.

5.3 Implementation: network encoder

5.3.1 Algorithm breakdown

The main advantage of using FPGA technology is parallel processing. Therefore, the encoder is designed with this in mind, utilizing as many parallel multiplication operations as possible. This is done by determining what data can be multiplied independently, and performing all those multiplication operations separately using multiple independent multiplier modules.

Equation 2.4 shows that $\boldsymbol{y(e_i)}$ is dependant on the global coding vector $\boldsymbol{g(e_i)}$ to obtain the coded output packet. Each symbol in the source packet $\boldsymbol{x_i}$ is multiplied by the respective coding coefficient $g_i(e_i)$. Therefore for a generation size of h, there exist Nh multiplication operations for a given output packet $\boldsymbol{y(e_i)}$. If *full vector* encoding is used, where the encoder waits for the entire generation to begin encoding, along h edges, there exist Nh^2 multiplication operations to complete the encoding process from equation 2.5. All of the multiplication operations are independent of one another and in theory there could be Nh^2 operations occurring in parallel.

In the practical system, the host computer needs to pipeline the incoming packets to the FPGA. The host computer word size is limited to 32-bits. Therefore, only four symbols of the incoming packet data can be processed at once. This presents a problem with full vector encoding because the encoder will need to wait for all the symbol segments to arrive before encoding can begin. Specifically, for a packet size of N symbols, $l = \frac{N}{4}$ segments are required. Therefore, it would take $lh = \frac{Nh}{4}$ loading operations for the encoder to receive the entire generation, to begin encoding. This is solved by performing the required multiplication operations on each incoming set of symbols and keeping

track of the results to be added together. Instead of waiting for the entire incoming packet generation to arrive, it is more affective to perform multiplication as the four byte packet segments are received. This is in contrast to full vector encoding and is a *on-the-fly* coding approach [63].

A coding vector, corresponding to a column vector from the coding matrix in equation 2.2,

$$\boldsymbol{g_i'(e)} = [g_i(e_1), g_i(e_2), ..., g_i(e_h)] \tag{5.1}$$

is generated for each incoming source packet in the source matrix 2.1, using a pseudo-random number generator. Each symbol in the generated coding vector is multiplied by each symbol from the source packet to produce h column vectors, for each edge e,

$$\boldsymbol{y_i'(e_1)} = \begin{vmatrix} g_i(e_1)x_{i,1} \\ \vdots \\ g_i(e_1)x_{i,N} \end{vmatrix}, \boldsymbol{y_i'(e_2)} = \begin{vmatrix} g_i(e_2)x_{i,1} \\ \vdots \\ g_i(e_2)x_{i,N} \end{vmatrix}, ..., \boldsymbol{y_i'(e_h)} = \begin{vmatrix} g_i(e_h)x_{i,1} \\ \vdots \\ g_i(e_h)x_{i,N} \end{vmatrix} \tag{5.2}$$

where h vectors are produced for each incoming packets as,

$$\boldsymbol{y_1'(e), y_2'(e), ..., y_h'(e)}$$

Summing the column vectors from 5.2 for each edge e gives the encoded output matrix from equation 2.5,

$$\boldsymbol{y(e)} = \begin{vmatrix} (\sum_{i=1}^{h} \boldsymbol{y_i'(e_1)})^T \\ (\sum_{i=1}^{h} \boldsymbol{y_i'(e_2)})^T \\ ... \\ (\sum_{i=1}^{h} \boldsymbol{y_i'(e_h)})^T \end{vmatrix} \tag{5.3}$$

This coding approach can be used with the segmented input source. The approach only requires $4h$ multipliers for the four incoming symbols, and the addition operation is performed as the incoming symbols are multiplied. Therefore, the encoded output is obtained once all the incoming packets are received and requires $\frac{Nh}{4}$ less multipliers for the same amount of time as a full vector encoder.

5.3.2 Overview

The flow diagram, illustrating the various hardware encoder modules is shown in Fig. 5.1. The pipeline is broken into the following sub-components: a pseudo-random number generator *prngen*, a 32x128 input data FIFO *data_ in*, Galois field multipliers *gf_ muls* and a Galois field adder *gf_ add*. The sub-components are controlled by six synchronous processes: *data_ in*, *count*, *pseudo*, *multiply*, *add* and *data_ out*.

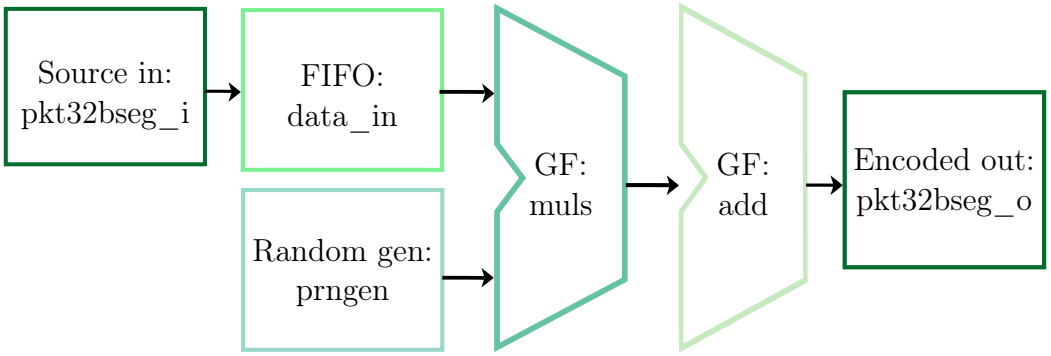

Figure 5.1: Hardware encoder flow diagram

The encoding process begins by piping segmented packet data into the *data_in* FIFO. The *data_in* process is used to control the FIFO read and write operations for the packet input segments. The first coding symbol vector is then generated from equation 5.1. The *pseudo* process is used to generate the pseudo-random numbers for the coding vectors, and output the coding vectors to the correct port. Once the FIFO is ready to be read, each of the four source symbols are multiplied by each value in the coding vector to produce,

$$\boldsymbol{y_i'(e_1)} = \begin{vmatrix} g_i(e_1)x_{1,1} \\ g_i(e_1)x_{1,2} \\ g_i(e_1)x_{1,3} \\ g_i(e_1)x_{1,4}, \end{vmatrix} \tag{5.4}$$

for each coding vector symbol,

$$g_1(e_1), g_2(e_1), ..., g_h(e_1)$$

This is done using the *multiply* process where the multiplier operands are assigned to the source and coding symbols. The processing continues until all l packet segments are received for the first packet, and the results are stored in an output vector. Further incoming segments are processed and the results of the proceeding multiplications are appended using XOR operations. This addition is done during the *add* process. When the first segment of the final packet is received, the output vector represents the fully encoded output packets, and a output flag is asserted to signal that the encoder output is ready. The encoder then outputs the result in four byte segments, just like the input. The *data_out* process controls when the output vectors should be output to the output port. To maintain synchronisation the *count* process is used. The *count* process controls all the counters responsible for delaying input to the multipliers, adder and output vector while information is being processed by the multipliers.

Further, more detailed discussions of the various sub-components are provided in the proceeding subsections.

5.3.3 FIFO buffer

The encoder takes longer to encode the incoming packet data, than the rate at which the data comes in. This is the overhead of network encoding, or the cost. Due to this, a buffer is required to store the incoming packet data while the packets are processed. One commonly used method in FPGA hardware design is to use First-In, First-Out (FIFO) buffers. FIFOs have a specified depth, which is the number of data elements, that can be stored. These data elements are constrained to a set width. Intel provides a FIFO Intellectual Property Core (IP Core) [64], that can be customized using the MegaWizard Plug-In tool in Quartus.

The design parameters are set using the MegaWizard Plug-In tool. The FIFO is designed to store all data required to successfully perform encoding, on a full generation, even if encoding begins before all the packets have arrived. In this instance the *width* and *depth* are set to 32 and 128 respectively. The *width* is set to equal the 32-bit input and output word size used by the host operating system. The host computer can only output data through PCIe to the FPGA in 32-bit word sizes. The *depth* of the FIFO is calculated based on the input packet size $N = 64$ bytes, and the network coding generation size $h = 7$. Dividing N x $h = 448$ bytes into 32-bits, equals $\frac{8Nh}{32} = 112$ segments. Since the MegaWizard Plug-In tool limits the depth of the FIFO to base-two numbers, the depth is therefore rounded up to 128 data elements. The input FIFO buffer pin-out is shown in Fig. 5.2.

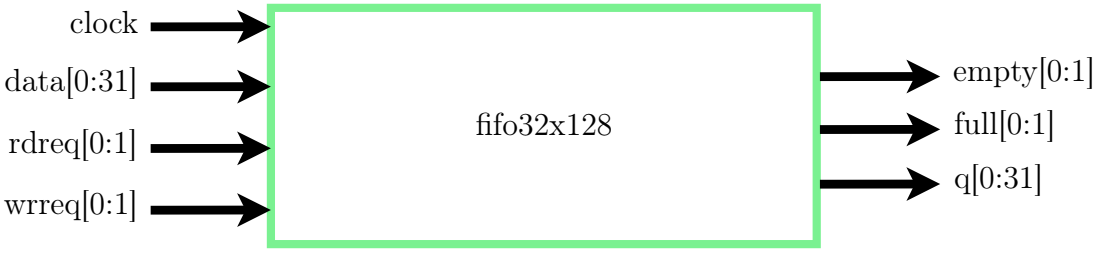

Figure 5.2: Hardware encoder entity: input FIFO buffer

The FIFO has four inputs: *clock*, *data*, *rdreq* and *wrreq*. The output side includes *empty*, *full* and *q*. The incoming packet data is input to *data* during write mode, and output to *q* during read mode. Both read and write operations are clocked by the *clock* input. Careful attention needs to be given, to prevent writing to the FIFO when full (Overflow), or reading from the FIFO when empty (Underflow). To prevent these errors, two simple FSMs are used to control the read and write states. The FSM diagrams for read and write are shown in Fig. 5.3. During start-up, both states are set to *idle*. While both state machines are in *idle*, the *wrreq* and *rdreq* flags are set to zero, and no read or write operations are performed. While in idle, the write state machine

checks to see if the *full* flag is set to zero. This indicates that there is space remaining and it is safe to write data. The state machine then enters the write state and sets the *wrreq* flag to one. Incoming packet data is written through the *data* port until the *full* flag is set back to zero. The read state machine functions in a similar way, except by monitoring the *empty* flag. The state machine remains in the read state while the FIFO is not empty, and the *empty* flag is set to zero. However, unlike the write state, the request flag *rdreq* is only set if the multiplier is also ready to process the data. Therefore to read the data from the *q* port, the read state machine must be in the read state, and the *mulready* flag must be set to one. This functionally enables the FIFO to be used to bottleneck the incoming data stream. The output rate is controlled by the multiplier counter, and the input is unrestricted while the FIFO has space.

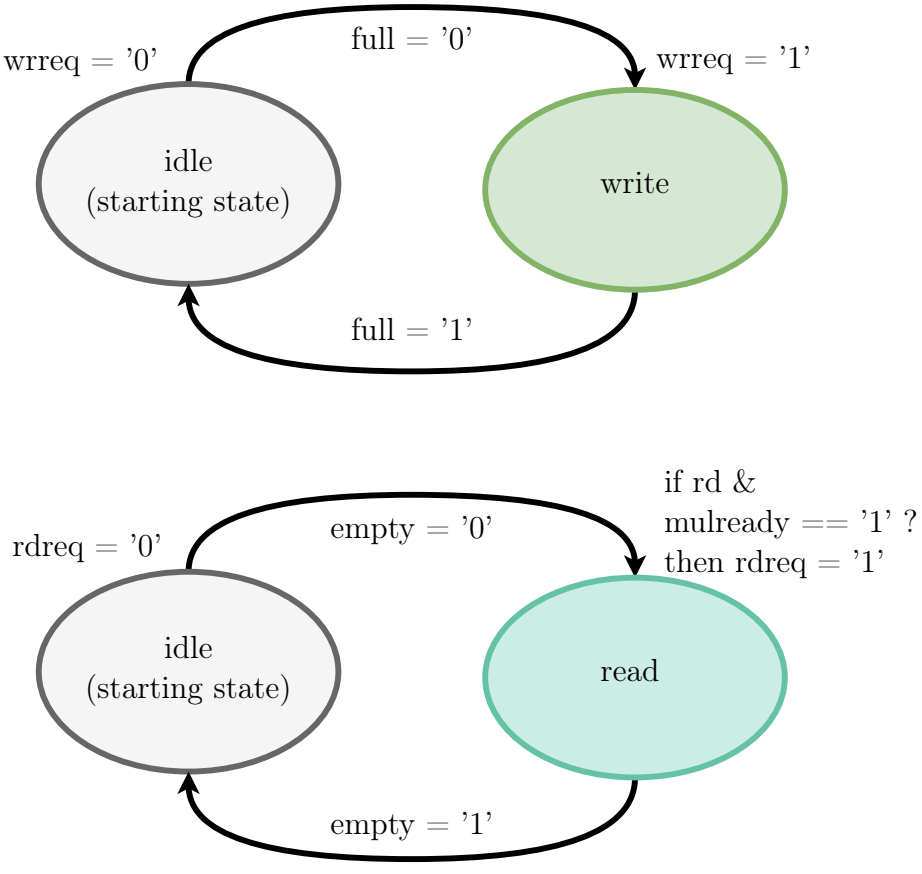

Figure 5.3: Hardware encoder input FIFO read and write FSMs

5.3.4 Pseudo-random number generator

A technique for generating random numbers is required to produce the coefficients for the coding vector in equation 5.1. These coding symbols are used to encode the source data through matrix multiplication. As a result, a hardware-based random number generator is required. It is not physically practical to generate true random numbers within hardware, and therefore a pseudo-random number generator is created instead. This is done using an 8-bit Galois Linear-Feedback Shift Register (LFSR).

According to Klein's book, LFSRs are an essential component in the fields of cryptography and coding theory [65]. A shift register is a sequence of flip-flops where the output of each flip-flop is connected to the input of another. Each flip-flop stores a single bit or state. When a clock signal is applied, the bits shift from one flip-flop to the other. A shift register of m flip-flops enables up to 2^m states. An $m = 8$ bit shift register is illustrated in Fig. 5.4 and consists of eight states $S_0, S_1, ..., S_7$. While a shift register itself is useful for converting between interfaces, this application requires something more interesting. This is done by introducing *feedback* to alter the state of the flip-flops. Fig. 5.5 shows how feedback is added to created a linear-feedback shift register, where the input bit is a linear function f of the previous state. In this application the linear function is the XOR operation. Therefore the input of the shift register is the XOR function of the current state bits.

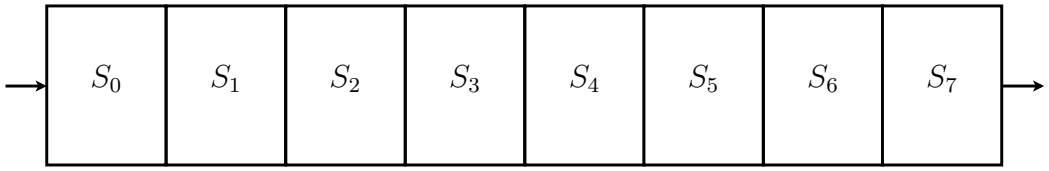

Figure 5.4: 8-bit shift register

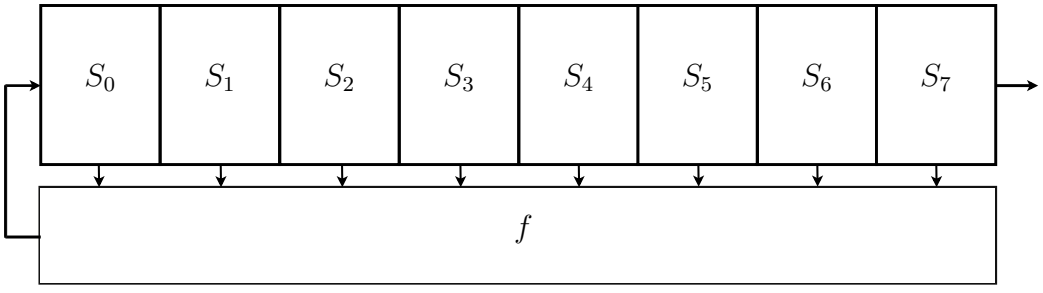

Figure 5.5: 8-bit linear-feedback shift register

A practical implementation of the LFSR can be realised in two ways. The first is to directly implement Fig. 5.5 in hardware. This is known as the Fibonacci implementation and makes use of external feedback, where the function is performed externally. An alternative to this is the Galois implementation that uses internal feedback. This is the preferred method as the Galois implementation only requires the use of two-input XOR operations, compared to multi-input of the Fibonacci. This enables a higher clock speed to be used for a Galois LFSR, while producing the same output as a Fibonacci LFSR [65]. Therefore a Galois LFSR is used for this implementation, as illustrated in Fig. 5.6.

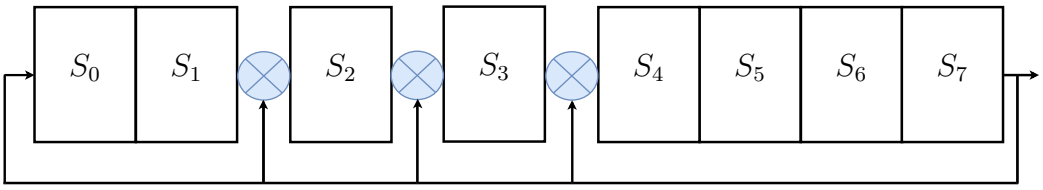

Figure 5.6: 8-bit Galois linear-feedback shift register

The LFSR uses three XOR operations as the internal feedback function. The location of the XORs is dependent on the bit positions used to modify the state. These bit positions are known as the *taps* and are determined using a primitive polynomial for an 8-bit Galois field $GF(2^8)$. While there are multiple primitive polynomials for the field, the one used in the implementation of the LFSR is,

$$p(x) = x^8 + x^4 + x^3 + x^2 + 1 \tag{5.5}$$

which is determined from the Steinwurf Fifi library. The "binary8.hpp" file used by [66] lists the default prime as 0x11D which in binary 100011101b equates to the primitive polynomial in equation 5.5. It is important to use the same primitive polynomial as the Fifi library because it is used by Kodo and would otherwise generate a different set of random numbers. This would lead to an inaccurate comparison between the hardware and Kodo implementation.

The starting state of a LFSR is called the *seed*, and is selected to be an arbitrary number, *31*. This provides a starting value for the LFSR, which will proceed to output a non-repeating sequence of numbers from one to 255. The sequence is determined by the seed, and will remain the same until changed. In this use case the seed is kept constant. If security is a priority, then the seed could be set to a dynamic value. One such value would be to use the input data at a set interval. Another option, if a real-time clock is available, is to use a segment of the current time. The only restriction on the seed is that it must be an 8-bit number. This being said, LFSRs are not cryptographically secure and additional work is required to generate a true secure sequence [65].

In the scope of this thesis however, this implementation is sufficient to obtain a random sequence.

The *prngen* pin-out is shown in Fig. 5.7.

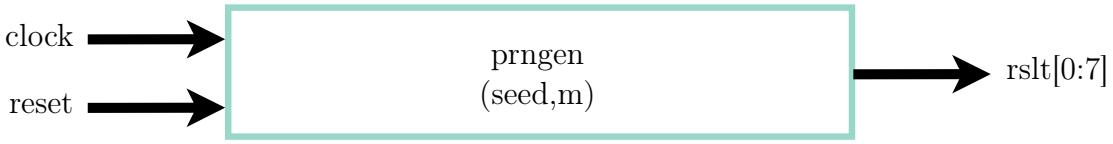

Figure 5.7: Hardware encoder entity: prngen

The module uses two constants: *seed* and *m*. The *seed* value is used as the starting seed for the LFSR, and *m* is the Galois field size of eight. After the *reset* flag is set to one, the *rslt* output vector will output a different 8-bit number for each clock cycle *clock*. This continues until all values in the interval $[1 : 255]$ have been generated, then the cycle repeats.

5.3.5 Galois field Multiplier

The encoder module requires the functionality of multiplying the source symbols with the coding symbols. This entity is also used during Gauss-Jordan elimination by the decoder. The multiplication needs to be done within a Galois field $GF(2^8)$, and therefore normal multiplication will not work. According to Kerl, multiplication within the Galois field can be done in two methods [22]. The one method *reduce-en-route* produces the result as following,

$$rslt(x) = (op_1(x) * op_2(x))mod(p(x)) \tag{5.6}$$

where the first operand op_1 is multiplied by the second op_2, and then the remainder is obtained by dividing by the primitive polynomial $p(x)$. This is done effectively using the same LFSR from the pseudo-random number generator in subsection 5.3.4.

Two processes are used within the module, the one *mul* begins by taking the binary multiplication of each bit in op_1 by the first bit in op_2, which is incremented by a counter. This result is fed to the LFSR process, which performs the modulo operation and the result is then fed back to the *mul* process. This continues until the binary multiplication has incorporated each bit from op_2. The Galois field multiplication result takes $m-1$ shift sequences for the overall state to represent the correct output.

The pin-out for the Galois field multiplier is shown in Fig. 5.8. The module has the inputs *clock*, *reset*, *operand_ 1*, *operand_ 2* and the output *rslt*. Using the module is fairly straightforward as *operand_ 1* is multiplied by *operand_ 2* and the output result is given by *rslt*.

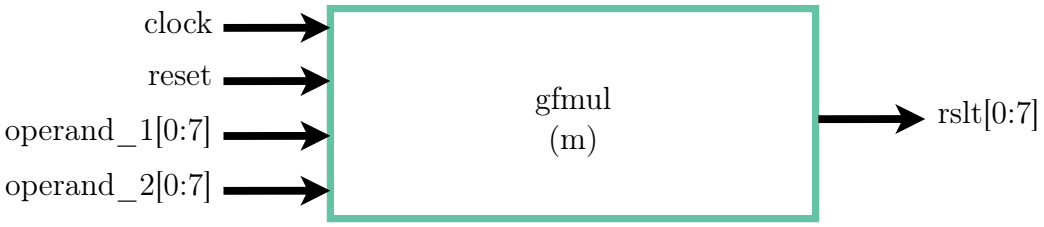

Figure 5.8: Hardware encoder entity: Galois field multiplier

5.3.6 Galois field Adder

The encoder module needs to sum the results obtained by the multiplier modules over the Galois field, $GF(2^8)$. Unlike multiplication, addition and subtraction are simply performed using the XOR operation. This is a lot easier to implement in hardware than the multiplier module. For this reason, a separate module is not created for the Galois field adder. The addition operations take place within the *add* process.

5.4 Implementation: network decoder

5.4.1 Algorithm breakdown

The main aim of the decoder, is to decode the incoming encoded packets. In order to solve for the original source packets,

$$\boldsymbol{x_1}, ..., \boldsymbol{x_h}$$

This requires the coding vectors, as well as the encoded vectors. Each encoded packet includes coding symbols in the header, along with the encoded data symbols. In a *full vector* coding approach, the entire generation of encoded packets and coding vectors is needed to begin the decoding process. As with the encoder, the symbols are grouped together as a *generation*, to construct two matrices: the coding matrix from equation 2.2,

$$G_t = \begin{vmatrix} g_1(e_1) & g_2(e_1) & ... & g_h(e_1) \\ \vdots & \vdots & \ddots & \vdots \\ g_1(e_h) & g_2(e_h) & ... & g_h(e_h) \end{vmatrix}$$

and the encoded output matrix from equation 2.5,

$$\boldsymbol{y(e)} = \begin{vmatrix} y_1(e_1) & y_2(e_1) & ... & y_N(e_1) \\ \vdots & \vdots & \ddots & \vdots \\ y_1(e_h) & y_2(e_h) & ... & y_N(e_h) \end{vmatrix}$$

The original source symbols are found by combining these matrices as a system of linear equations and solving for the inverse of the input data matrix to obtain, from [5],

$$\begin{aligned}\left|G_t \quad \boldsymbol{y}(\boldsymbol{e})\right| &= \begin{vmatrix} g_1(e_1) & g_2(e_1) & \dots & g_h(e_1) & y_1(e_1) & y_2(e_1) & \dots & y_N(e_1) \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1(e_h) & g_2(e_h) & \dots & g_h(e_h) & y_1(e_h) & y_2(e_h) & \dots & y_N(e_h) \end{vmatrix} \\ &= \begin{vmatrix} 1 & 0 & \dots & 0 & x_{1,1} & x_{1,2} & \dots & x_{1,N} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 & x_{h,1} & x_{h,2} & \dots & x_{h,N} \end{vmatrix} \\ &= \left|I_h \quad \boldsymbol{X}\right| \end{aligned} \tag{5.7}$$

One effective method to determine the inverse of a matrix in hardware, is Gauss-Jordan elimination. The details of how Gauss-Jordan elimination is implemented in FPGA hardware is discussed in the proceeding subsections.

5.4.2 Overview

Gauss-Jordan elimination is implemented in VHDL using a FSM approach. This is implemented as a module, and performs the necessary row operations on the input data and coefficient matrices. These row operations are performed on the entire packet length, until the coefficient matrix is in reduced-row echelon form. The host computer, sends the encoded packet data to the FPGA via PCIe. A problem occurs that the host word size of 32-bit is much less than the packet length used by the Gauss-Jordan elimination module. This is solved by encapsulating the module within a decoder FSM module, responsible for loading 32-bit packets into the Gauss-Jordan elimination module. The decoder state machine is shown in Fig. 5.9. There are four states, *load*, *gj_ elim*, *complete* and *idle*. The first state is the *load* state, and is where the 32-bit packet segments are loaded into the data-in and coefficient registers. The state machine then enters the *gj_ elim* state, executing the Gauss-Jordan elimination process. Once done, the *complete* state is entered to load the decoded data back into 32-bit data segments. Then finally, after all segments are loaded out, the *idle* stage is entered until the reset is triggered.

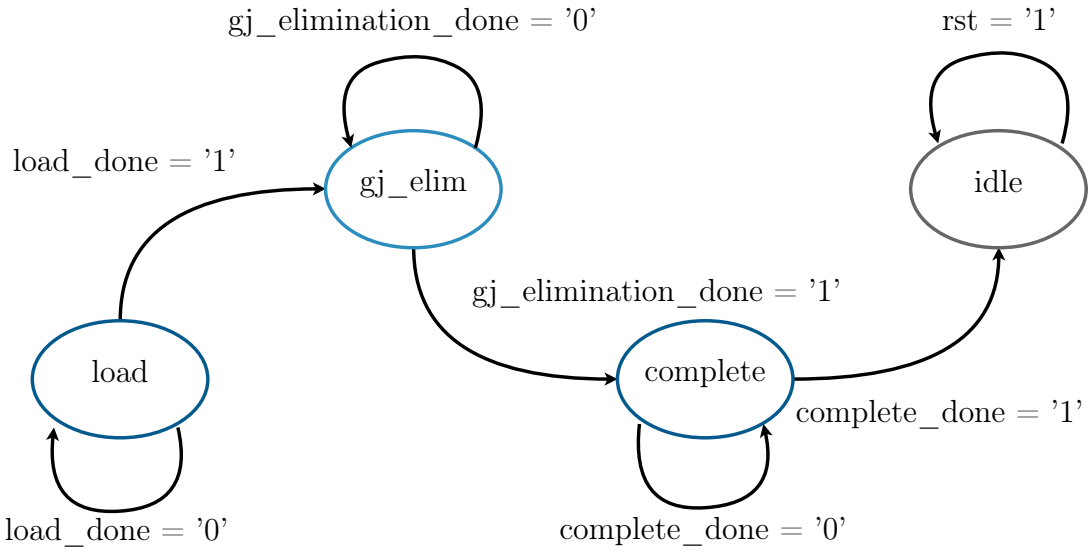

Figure 5.9: Hardware decoder finite-state machine

The data representation of the *load* state is shown in 5.10. The input data packet on the FPGA side, *data_in*, has a size of $N = 64$ bytes and is broken into 32-bit segments. The number of segments required is equal to $l = \frac{8N}{32} = 16$. The coefficient data for each packet, *coef_in*, is $k = 7$ bytes long and therefore uses less than two segments. The decoder FSM can only load one data segment per clock cycle on a single PCIe lane. Therefore, for a generation size of h, the number of clock cycles required to load the data in and out of the Gauss-Jordan elimination module is bottlenecked by the generation size and length of the input packet, as long $N > k$. As a result, the number of load operations required for segmentation is $2lh$, where $l = \frac{8N}{32}$.

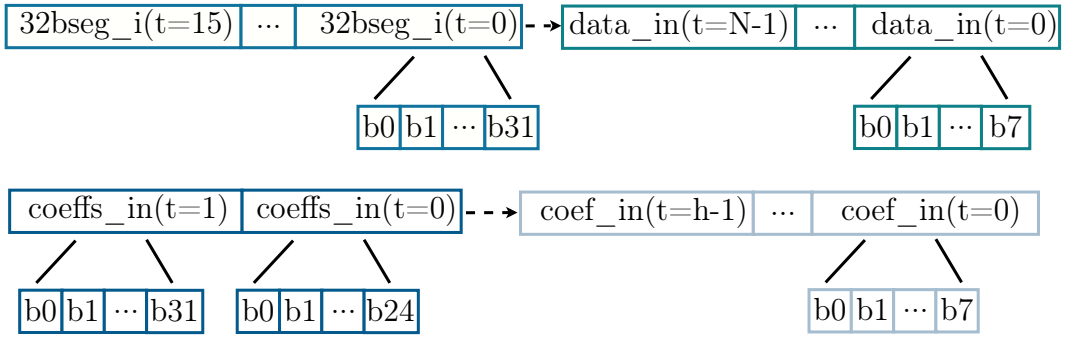

Figure 5.10: Hardware decoder data segmentation

This section is written from a *top-down* approach, where the subcomponent required in each module is discussed as the text progresses. Each row operation is designed as a separate hardware module, and discussed in subsections 5.4.4

and 5.4.5. These row operations are broken down into further Galois field inverter and divider modules, in subsections 5.4.7 and 5.4.6 respectively.

5.4.3 Gauss-Jordan elimination

The Gauss-Jordan elimination process is done by performing row operations, specifically row division and row multiplication-and-subtraction, on both the input (encoded) data and RLNC coefficient data matrices, until the coefficient data matrix is in reduced-row echelon form. Specifically since the coefficient matrix is an invertible h x h square matrix, it is non-singular, and the reduced-row echelon form will also be an identity matrix. Once this is achieved, the resultant output data matrix will be the inverse of the input matrix and represent the original decoded data. The process is broken into solving the *lower* and *upper* triangles of the coefficient matrix, obtaining an identity matrix.

The *lower* triangle is solved first and makes use of both row operations: division and multiply-and-subtract. The first step is to obtain a leading value of one in the first entry of the first row. This is done using the row division operation and dividing the first row by the leading value entry of that row. The next step is to make sure that all entries below the leading value of one are equal to zero. This is done using the multiply-and-subtract operation. The first row Ri, is multiplied by the leading non-zero entry in each of the following rows Rj and is then subtracted from that row. The process continues using the row division operation to make the leading value entry of the second row equal to one. This continues until all the entries in the *lower* triangle of the coefficient matrix are equal to zero, and the diagonal entries are all equal to one. The row division and multiplication-and subtraction operations are performed h and $(h-1)$ times respectively. The coefficient matrix is in row-echelon form, the result of using Gauss elimination.

While row-echelon form is sufficient to decode the encoded input data, a further step is taken to solve the *upper* triangle, to obtain a reduced-row echelon form. This provides a more explicit solution, where the resultant data matrix represents the exact required output of the decoded data. This result is the matrix inverse of the encoded data matrix. Solving the *upper* triangle of the coefficient matrix, only makes use of one of the row operations: multiply-and-subtract. This is performed for $(h-1)$ operations, until the resultant coefficient matrix is in reduced-row echelon form, and an identity matrix.

The Gauss-Jordan elimination FSM makes use of *lower* and *upper* triangle flags to help determine which state the Gauss-Jordan elimination module should be in. The states of the FSM are *load*, *divRow*, *mulsubRow*, *complete* and *idle*. Each state comprises of a separate pipeline and sub-components. The state machine diagram is shown in Fig. 5.11. The starting state is the *load* state. During this state, the encoded packet data and coding coefficients are loaded into the data and coefficient matrices respectively. Each matrix contains $h = 7$ number of rows, corresponding to the generation size. The

data matrix contains $N = 64$ byte length rows, and the coefficient matrix contains h byte length rows, stored as registers. Fig. 5.12 illustrates the matrices data representation. Once the *load* state is completed, the Gauss-Jordan elimination process is started.

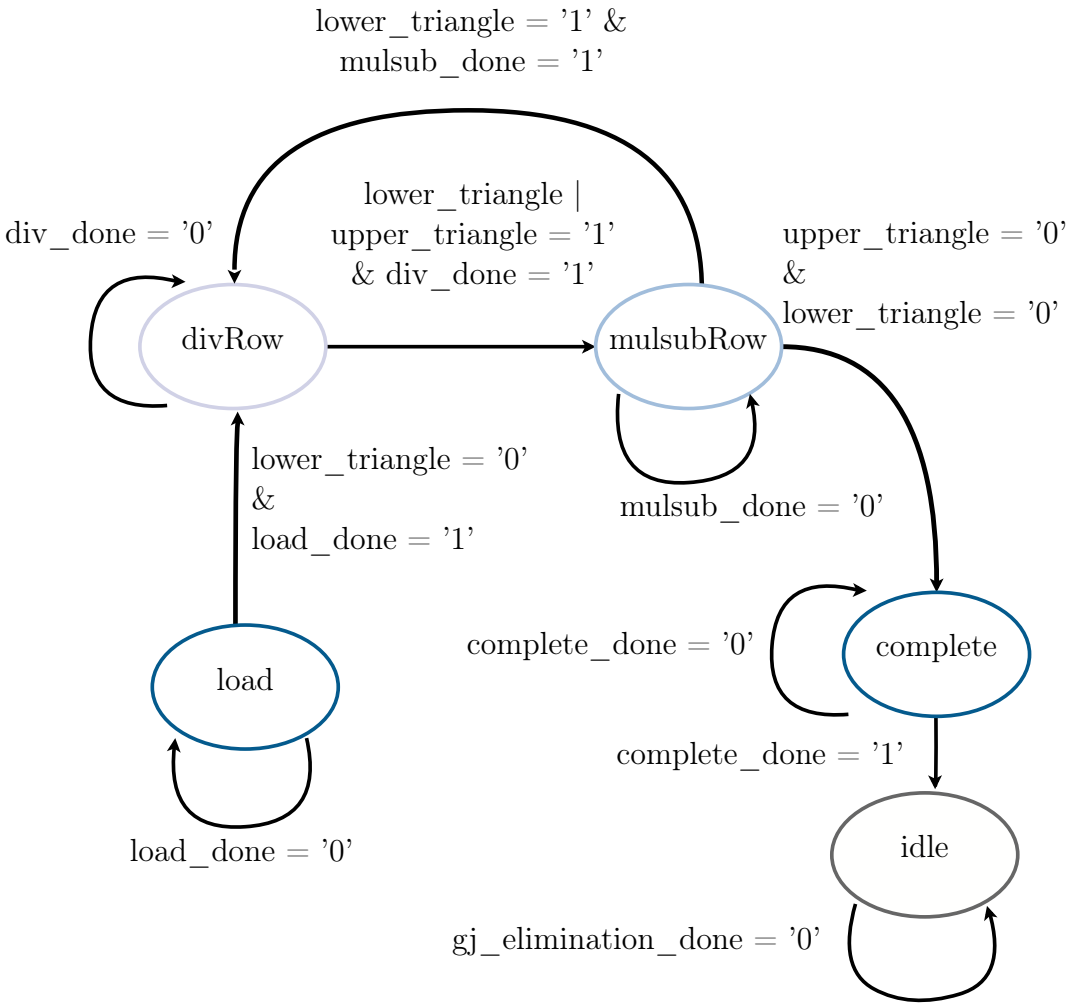

Figure 5.11: Gauss-Jordan elimination finite-state machine

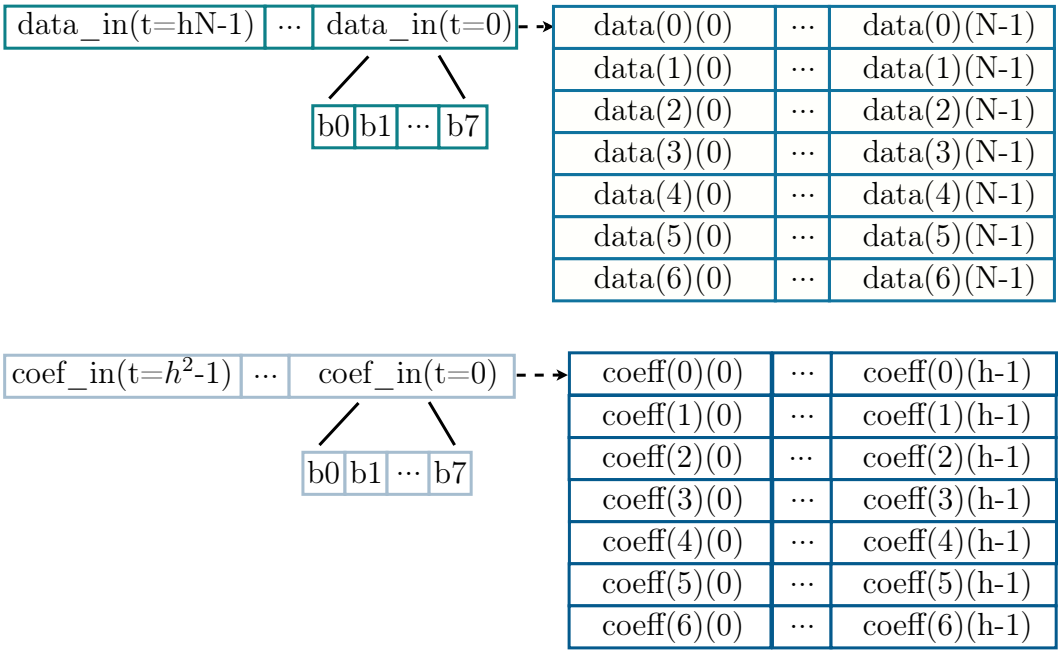

Figure 5.12: Gauss-Jordan elimination matrices representation

The states *mulsubRow* and *divRow* both represent a Gauss-Jordan elimination row operation. The *lower-triangle* is solved first and begins in the *divRow* state. The *divRow* state is entered when the *lower* triangle flag is set to zero. The state will change to *mulsubRow* when row division is completed and either of the triangle flags are set. The state machine will then cycle back to *divRow* if the *lower* triangle flag is still set. This cycle continues until the *lower-triangle* is solved. The next step is to solve the *upper-triangle*, which only makes use of the *mulsubRow* state. If the *upper* triangle is set to one, the *mulsubRow* state will be held until both triangle flags are set to zero. After Gauss-Jordan elimination is completed, the decoded data matrix is loaded into the output registers. This happens during the *complete* state. Finally, once the output loading is complete, the state machine enters the *idle* state.

The VHDL implementation of the Gauss-Jordan elimination is constructed using two module components, one for each row operation. A single row division module is used as row division only needs to be done on one row at a given time. The multiply-and-subtract row operations are normally done on more than one row during the *mulsubRow* state, and therefore there are h multiply-and-subtract modules, to allow multiple row operations in parallel. This is used to decrease the processing time as in theory, performing the multiply-and-subtract operation on one or h rows should take the same amount of time in hardware. The remaining VHDL includes four processes. The one process is for the FSM itself, and the other three are for *load* and *complete*, *divRow* and *mulsubRow* state processing.

5.4.4 Row operation: divide

Gauss-Jordan elimination requires a row division operation. This is used to divide an entire row by a set value, usually with the intention of obtaining a positive one for the leading value. The leading value being the first non-zero value in the row from left to right. This is used along with the multiply-and-subtract row operation to convert the coefficient matrix into an identity matrix form.

The row division module divides the input and coefficients data rows. The module consists of $h + N$ dividers for each byte in the input and coefficient data rows. This allows for the entire row to be divided in parallel. The remaining hardware in the module is used to segment the row register into byte length registers, and check if the row division is completed. The module has five inputs: *clock*, *reset*, *a*, *pkt_coef_in* and *pkt_data_in*. Inputs *clock* and *reset* are used for clock input and reset triggering respectively. The value to divide the entire data and coefficient rows is given my the value *a*. The data row is given by *pkt_data_in* and the coefficient row by *pkt_coef_in*. Once the module has completed the row division operation, the data and coefficients is sent to the *pkt_data_out* and *pkt_coef_out* outputs. The *done_o* output flag is set to one, signalling other modules that the operation is completed. A summary of the row division module, with input and output bit sizes, is show in Fig. 5.13.

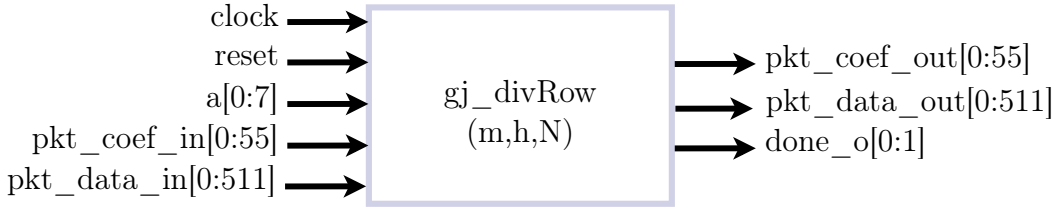

Figure 5.13: Hardware decoder entity: row division

5.4.5 Row operation: multiply-and-subtract

The second row operation required for the Gauss-Jordan elimination module is multiply-and-subtraction. During this operation, one row *Ri* is multiplied by an input value, and then subtracted from another row *Rj*. This is done with the intention of making all other entries in the row *Rj*, besides the leading value of one, equal zero. As with the row division operation, the multiply-and-subtract module is used to convert the coefficient matrix to an identity matrix.

The module multiplies both of the data and coefficient rows at row index *Ri* by a constant value and then subtracts them from the data and coefficient rows at row index *Rj*. The module contains $h + N$ multipliers for each byte in

the input and coefficient rows of row index *Ri*. Two processes are used to split the row data into individual bytes, as well as from individual bytes to single row registers. This enables the entire rows at row index *Ri* to be multiplied by the constant value in parallel. The subtraction of each row element is also done in parallel, using the XOR operation, in the Galois-field $GF(2^8)$. This is done in a separate process. Each of the multipliers use the same Galois-field multipliers as in the network encoder, discussed in subsection 5.3.5.

The multiply-and-subtract row module takes two sets of input data for each row. The inputs *pkt_coef_i_in* and *pkt_data_i_in* are for row *Ri*, while inputs *pkt_coef_j_in* and *pkt_data_j_in* are for row *Rj*. The value to multiply *Ri* with is given by the input *a*. Once the row operation is completed a single row output is provided by *pkt_coef_out* and *pkt_data_out*. The output flag *done_o* is set to one to signal that the row operation has completed successfully. A summary of the module is show in Fig. 5.14, illustrating the input and output bit sizes.

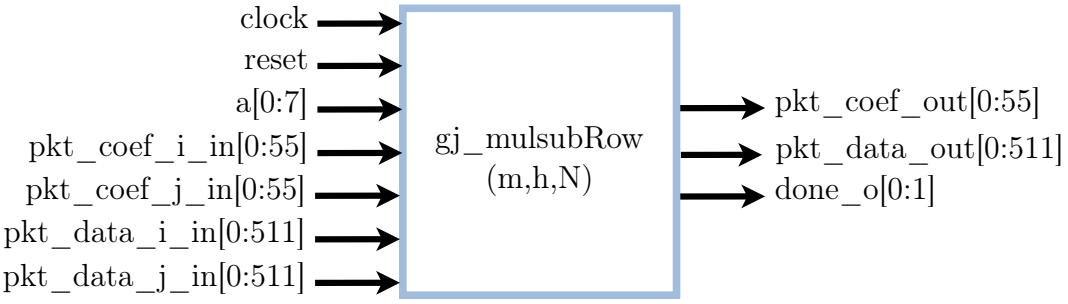

Figure 5.14: Hardware decoder entity: row multiply-and-subtract

5.4.6 Galois field divider

The row divider module requires a Galois field divider to divide each data and coefficient byte. The simplest way of doing this is to use the already constructed Galois field multiplier, discussed in subsection 5.3.5. The module pin-out is shown in Fig. 5.15. Instead of multiplying the two operands directly, the first input *operand_1* is multiplied by the inverse of the second input *operand_2*. The only additional component required is therefore a Galois field inverter. The Galois field inverter can easily be constructed using a lookup array, as discussed in subsection 5.4.7.

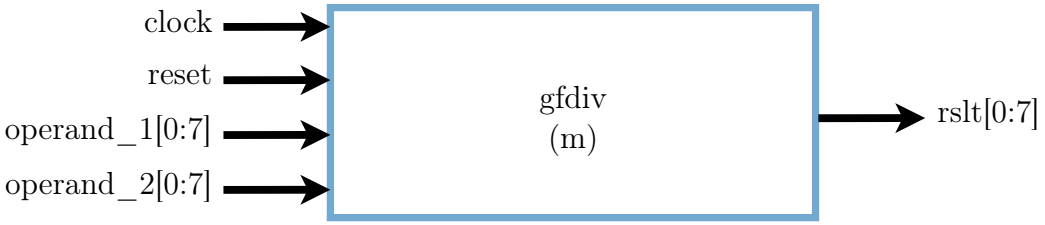

Figure 5.15: Hardware decoder entity: Galois field divider

5.4.7 Galois field inverter

The implementation of the Galois field divider requires a Galois field inverter. Creating a hardware inverter in a Galois field is generally considered to be a difficult task. Therefore, instead of using complicated logic, a simple lookup register array is constructed from pre-calculated values. These values are generated using Steinwurf's fifi-python library [66]. This is the same library that is used by Kodo, and therefore performs Galois-field calculations using the same primary polynomial as the implemented Galois field multiplier module. The array remains small for the Galois field $GF(8)$, containing 255 possible values. The array index is from one to 254 and each value corresponds to the inverse of the index. Zero is omitted from the array because the inverter results are used in division operations, which would cause an undefined output. Due to the small size of the array, this operation is favoured over more complex logic.

The pin-out of the module is show in Fig. 5.16. The operation is very straightforward. The number to be inverted is input to *operand*, and the output result is given by *rslt*.

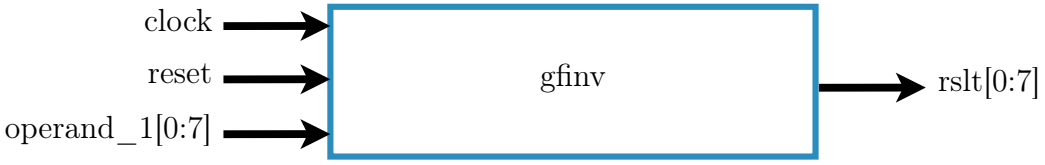

Figure 5.16: Hardware decoder entity: Galois field inverter

5.5 Module verification

Verification is conducted by writing *testbenches* for each of the sub-components, as well as the overall encoder and decoder. Input data is passed to the testbenches from text files to simulate incoming packet streams. The output data is stored in text files to analyse the completed results. Each of the modules

are evaluated using the Kodo RLNC Python library [59] and the Fifi Python library [66]. The outputs are then compared to the Kodo Python baseline from Chapter 4 to confirm that the coding operations, as well as the sub-modules are all functioning correctly.

5.6 Summary

This chapter presented details on the design and implementation of the FPGA-based hardware network coding functions. The discussion begins with the methodology used. The Quartus Prime design suite is used to develop the RLNC coding modules for the OpenVINO starter kit FPGA and the coding modules are written in VHDL. The design priorities are functionality, modularity and adaptability.

The encoder and decoder are created using a variety of sub-components that can be adapted and modified on an individual basis. In this thesis, however the modules are constrained to implement network coding on a generation size of seven packets, with a length of 64 bytes, within the Galois Field $GF(2^8)$.

The network encoder implementation begins by breaking down the RLNC encoder algorithm and figuring out what data can be multiplied independently, and performing all those multiplication operations separately using multiple multiplication modules. An overview of the various modules used by the encoder are provided. Each of the various sub-components are then discussed in detail.

The decoder implementation makes use of Gauss-Jordan elimination to decode the encoded packets. A FSM approach is used by the decoder implementation to perform the various row operations of Gauss-Jordan elimination, as well as load data to and from the decoder module. Each of the decoder sub-components are discussed in detail.

Chapter 6

Open vSwitch and network coding function integration

6.1 Introduction

Chapters 4 and 5 provide details on the VNF and FPGA network coding implementations respectively. The coders need to be integrated with OvS to be used in a real network. This chapter discusses the design and implementation of the interface between the coders and OvS bridge.

6.2 Software layer integration: OvS and VNF

The DPDK-based encoder and decoder from Chapter 4 need to be interfaced with the OvS bridge. While the use of VNFs is chosen as the method of offloading packets to the software-based coders, two other possible methods of offloading packets from OvS to the DPDK userspace program were investigated. The two methods involve direct use of OvS *dpdkr* and *vhost-user* ports [11].

The dpdkr and vhost-user ports are normally used to interface OvS with VM guests. It was investigated if it would be possible to directly access the network interfaces queues of the ports, and have the OvS switch forward packets to the ports for coding. This would possibly eliminate the overhead of packets needing to be passed through the coding VM.

The dpdkr port makes use of the DPDK ring library. This library is used extensively in the implementation in Chapter 4 and therefore shows promise of ease of integration. An interface was created using the dpdkr port, but a segmentation fault often occurred during packet processing. Reading through the DPDK source code reveals that the rings used by the dpdkr port are created as single producer and consumer. This therefore limits the interface as multiple processes can not access the port rx and tx ring simultaneously. This is a requirement because the interface is the second process accessing

the port while OvS is the primary process. The DPDK documentation also suggests that dpdkr should no longer be used for VM guest communication. The vhost-user port should be used instead.

Investigation into the vhost-user port shows that it is used by QEMU to interface OvS with VMs. An attempt was made to create a new interface by following the QEMU source code [67]. While no official documentation exists on how to do so, it was found to be difficult to figure out the exact communication between QEMU and OvS. Sniffing the Linux system calls during VM initialization provided more detailed insight, but the exact details of the necessary function calls were unknown. The process was unsuccessful and deemed out of the scope of this thesis. Therefore, the design decision was made to follow the work in [1] and [15] and implement the DPDK coding functions as VNFs.

The use of VNFs provide more flexibility than a direct interface to the OvS port. This allows for the coding implementations to still work if OvS makes structural changes to the ports. The use of VNFs also meet the objective requirements of open networking through the use of NFV and SDN.

6.2.1 Using OvS OpenFlow flows

The hypervisor from Chapter 3 connects the VNFs to the OvS bridge. To offload packets for processing, OpenFlow *flows* are added to the OvS bridge. The flows are used to direct traffic to the network coders based on a set of rule criteria and actions. The flows can be added to the OvS bridge using the built in OvS OpenFlow control *ovs-ofctl* utility, or by an SDN controller.

The *ovs-ofctl* utility is a command line tool used to monitor and administer the OvS bridge. The utility can be used with the *add-flow* argument to add a OpenFlow flow to the switch as following:

```
ovs-ofctl add-flow <switch> <flow>
```

The switch value is set to the name of the OvS bridge, in this case *br0*. The flow entry is specified by a specific flow syntax with key-value pairs. The first value is the match field and is followed by the actions field. An example of flows that could be used to offload packets for encoding and decoding are:

```
Encoder offload:
 ovs-ofctl add-flow br0 in_port="vhp0",actions=output:"encodeVNF
    "
Decoder offload:
 ovs-ofctl add-flow br0 dl_type=0x2020,actions=output:"decodeVNF
    "
```

In these example flows, all packets from the VM port *vhp0* would be forwarded to the encoder VNF. Encoded packets using the Ethernet type 0x2020 (from

the coding function selection process in Chapter 4) would be forwarded to the decoder VNF.

The ovs-ofctl command utility is useful for easily adding flows to the OvS bridge. This approach is good when a specific network topology and use case is required, such as with performing testing. For this reason the evaluation done in Chapter 7 uses the utility in some of the testing scripts. In an operational network however, with continuous packet flow, the network topology and configuration is not always known. There are many different use-case scenarios for a network switch, and therefore control plane controller support is implemented.

6.3 Control plane: Ryu SDN controller

One of the system requirements from Chapter 2 is that the network switch implementation needs to be able to connect to an SDN controller. This requirement is derived from the objective that the switch needs to be SDN compatible. Therefore, to verify the support and ability to use an SDN controller with the system, a control algorithm is designed and implemented that is capable of deploying network coding functions as necessary.

The requirements for network coding deployment vary with application. One of the benefits of using network coding on an SDN-based switch is flexibility of control. The SDN controller has the ability to make forwarding decisions based on user defined algorithms and criteria. The controller uses these algorithms to create the necessary flows on the OvS bridge.

The SDN switch sends all unknown packets to the controller. The controller analyses the packets and creates the necessary flows on the OvS bridge. Future packets that have matching fields are not sent to the controller and are forwarded using the dataplane.

The control plane is managed by a dedicated computer running the Ryu SDN controller software. The Ryn SDN controller is a Python-based framework that supports OpenFlow up to version 1.5. Ryu provides an extensive API that is open source and is officially integrated into OpenStack. The controller therefore meets the objective of open networking and the Ryu project is at the time of writing this thesis, well supported [68].

Another system requirement from Chapter 2 is that the switch needs to be able to perform multicast snooping to be used to evaluate network coding in a practical network.

Ryu is used to create a controller algorithm that implements multicast snooping. The specific multicast protocol used is IGMP. While other multicast protocols exists, IGMP is chosen because the controller algorithm is only created as a proof of concept to determine SDN controller support. The controller algorithm is implemented as an extension of the basic L2 learning switch

example used by Ryu. Network coding functionality is added using the coding function selection process from Chapter 4.

6.3.1 Multicast Snooping

There is no single method for conducting multicast snooping on a switch. Many switch manufactures use their own implementations for their specific needs. To create a more consistent standard, RFC 4541 in [69] provides a guideline of how to implement IGMP and Multicast Listner Discovery (MLD) on a network switch. The RFC 4541 recommendations require the use of an IGMP router (querier) to work correctly. In the context of this controller however, there is no IGMP router present. The scope of the thesis is to create a network switch, which is a LAN specific (L2) networking device and therefore router support is omitted. The RFC 4541 recommendation is therefore only used as a general guideline and reference.

Flow diagrams of the IGMP snooping functionality from the dataplane and controlplane is given in Fig. 6.1 and 6.2.

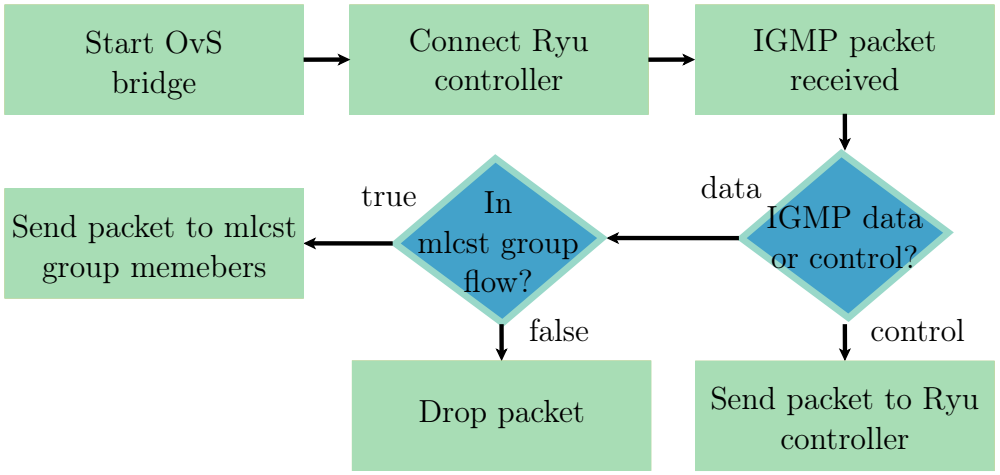

Figure 6.1: IGMP snooping functionality flow diagram from OvS bridge.

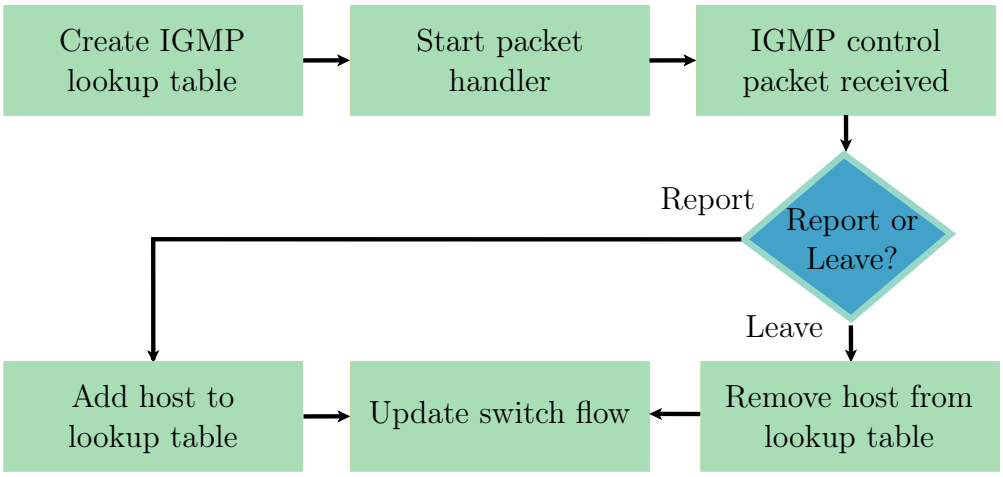

Figure 6.2: IGMP snooping functionality flow diagram from Ryu controller.

The IGMP snooping process begins on the OvS bridge. The bridge is started and connected to the Ryu controller. If an IGMP packet arrives, it is checked to be a control or data packet. Control packets refer to IGMP query, report or leave packets. IGMP data packets are multicast packets to be sent to the multicast group. If the packet is IGMP control, it is sent to the Ryu controller.

The Ryu controller begins by creating an IGMP lookup table of the structure:

```
grp_to_mac = defaultdict(list)
```

Each table entry is a multicast group that contains a list of ports that belong to the group. If a report message is received, then the port is added to the relevant multicast group. Similarly, if a leave message is received, the port is removed from the group. The lookup table is used to update the switch flows. If multiple OpenFlows flows exist with the same match, then the first flow will be followed. This will prevent the switch from sending to all members of the multicast group. To solve this, the controller only creates a single flow for each multicast group. The group members are modified by adding or removing OpenFlow output actions, where each output action corresponds to a multicast group member.

Once the switch flows are updated, IGMP data packets can be sent to the correct multicast group member ports. If no flow exists for the group address then the packets are dropped. IGMP control packets are therefore the only IGMP type packets sent to the controller, and IGMP data packets remain in the dataplane.

6.4 Hardware layer integration: OvS and FPGA

The OvS bridge needs the ability to offload packets to the FPGA. The hardware-based encoder and decoder from Chapter 5 cannot directly be used in a network. An interface is therefore required between the OpenVINO development kit and OvS.

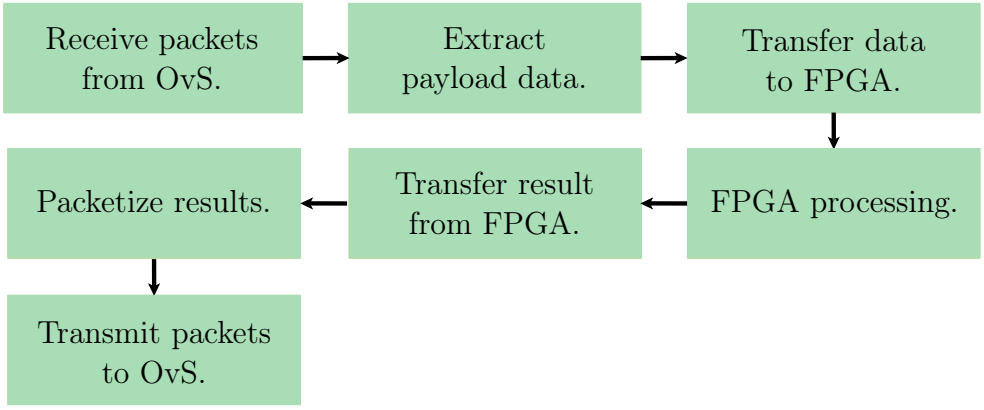

Figure 6.3: OvS and hardware-based network coding integration functionality.

An overview of the interface functionality is illustrated in Fig. 6.3. Packets to be coded are received from OvS. Once the packets are received, the payload data is extracted and transferred to the FPGA. The FPGA then processes the data using the network coders and the results are transferred back to the PC. The results are packetized and transmitted to the OvS bridge.

To meet the functional requirements from Fig. 6.3, the interface is designed in two parts. The interface is comprised of a software application and hardware modules, shown in Fig. 6.4.

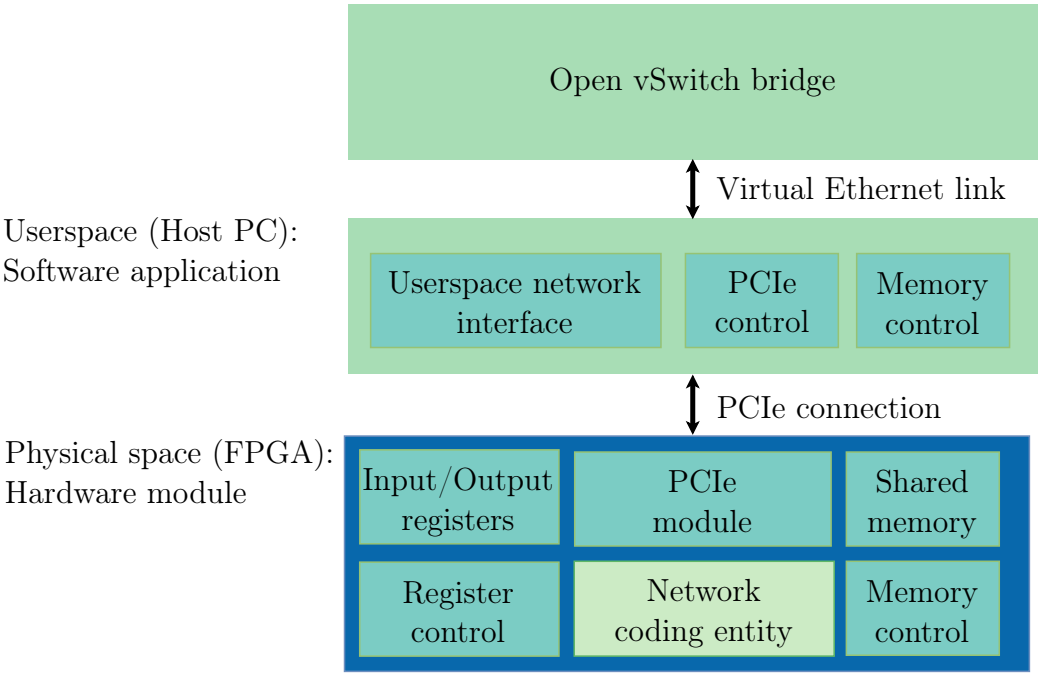

Figure 6.4: Hardware coding integration architecture.

The hardware modules are placed on the FPGA and encapsulate the coder entities. A separate hardware module is created for each coder. Only a single hardware module is placed on the FPGA at a time, and therefore a single coding entity. The hardware module requires the following to meet the functionality from Fig. 6.3:

- A *PCIe module* to implement PCIe connectivity, allowing the host PC to connect to the Cyclone V FPGA device. This will allow packet data to be transferred and offloaded to the FPGA for processing.
- A form of *shared memory* that is accessible by both the FPGA and the host computer. This memory is used to store packets to be processed by the coding entities.
- FPGA-based *memory control* logic to transfer packet data between the shared memory and the coding entities, on the FPGA.
- *Input and output registers* to control the FPGA coding entities from the host computer. These registers can be modified by both the FPGA and host computer to communicate control flags. The flags are used to tell the coder entity when to begin and tell the host computer when coding is finished.

- FPGA-based *register control* logic, to read and set the input and output registers from the FPGA.

The software application runs on the host computer and is written as a C application. Data transfer and communication between the OvS bridge and the FPGA is done through the software application. The application requires the following to meet the functionality from Fig. 6.3:

- A *network interface* that can be manipulated from the userspace. This will allow packets to be transferred between the software application and the OvS bridge.
- Software to provide *PCIe control*. This will allow the application to connect to the PCIe module on the FPGA.
- A form of *memory control* to read and write to the FPGA shared memory, from the userspace.
- A method of setting the *input and output registers* to control the FPGA coders.

We begin the discussion with the hardware module implementation. The Intel Quartus Prime 18.1.0 lite edition design suite is used to implement the FPGA hardware modules. The Quartus Platform Designer tool is used to add the necessary IP components to the design. The first step is to create the functionality required to connect to the FPGA device to the PCIe expansion bus.

6.4.1 Hardware module: PCIe module

Terasic, the manufacture of the OpenVINO starter kit provide a collection of reference designs to be used with their products [52]. Included in the collection is a PCIe reference design, *PCIe_Fundamental*. The PCIe_Fundamental reference design is used as a base for the hardware modules. The reference design contains the necessary PCIe IP core to interface the FPGA coding entities with the PCIe expansion bus. The IP core provided is specific to the Cyclone V FPGA device. All PCIe hardware logic is handled by the reference design. There is therefore no further PCIe logic implementation required. An overview of the reference design is shown in Fig. 6.5 from [52]. Avalon interfaces are used to provide the interconnection fabric to connect all of the components within the reference design.

Avalon interfaces are an interface protocol used to connect components within the Quartus Platform Designer software [70]. They are used to make system component integration easier within an Intel FPGA. The two Avalon interfaces used within the hardware modules are Avalon Memory-Mapped (MM)

and Avalon Conduit interfaces. The Avalon-MM interface is used for address-based read and write operations between a master and slave device. Avalon Conduit interfaces are used to represent signals, or groups of signals that can be exported from the Quartus Platform Designer, to be used with FPGA user logic written in other Hardware Descriptive Language (HDL) files.

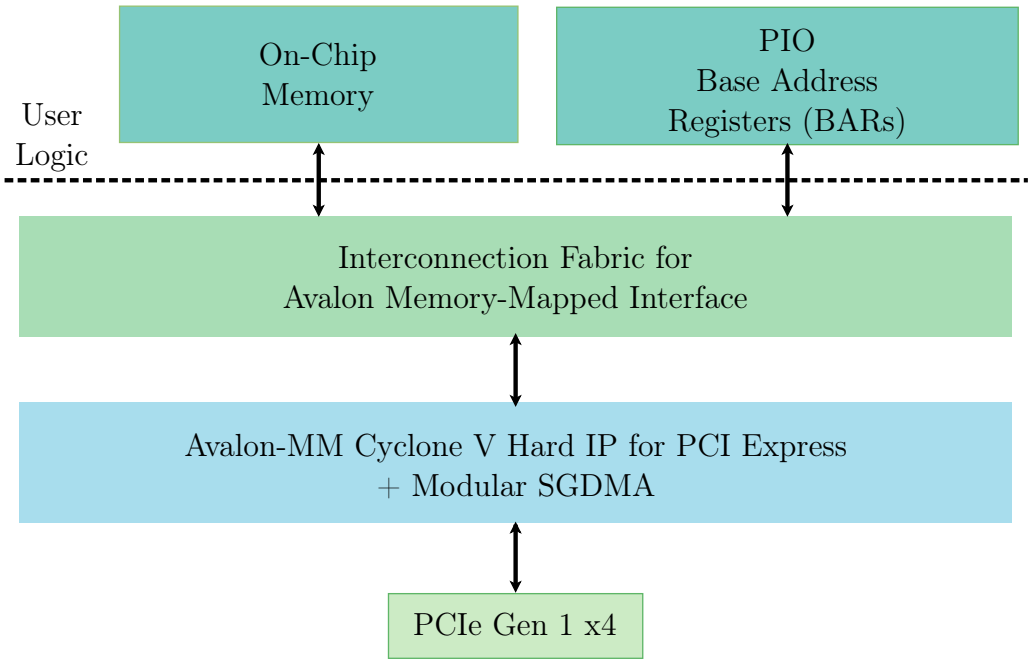

Figure 6.5: Hardware block diagram of PCIe reference design from [52]

6.4.2 Hardware module: network coding entity

The encoder and decoder entities from Chapter 5 are instantiated within the PCIe reference design. While the network coder entities are written in VHDL, the PCIe reference design is written in Verilog. Luckily, VHDL modules can be instantiated within Verilog (and vice-versa), and therefore the hardware modules are written in Verilog. A port map is performed to connect the coder entity ports to internal signals within the main PCIe_Fundamental Verilog file.

6.4.3 Hardware module: shared memory

Shared memory needs to be implemented on the FPGA, to store packets to be used by the coding entities. The PCIe reference design makes use of the on-chip Random Access Memory (RAM) IP core by Intel to store data. The IP core parameters are modified to set the data width to 32-bits and total memory size of 2048 bytes. The data width is set to match the word size used

by the host computer and the total memory size is chosen to accommodate for received, transmitted and coefficient packet data. The memory address map of the on-chip memory is set to the range 0x07000000 - 0x070007FF and shown in Fig. 6.6.

The memory map shows the address space used by the received, transmitted and coefficient data packets. The received packets refer to the packets received from the OvS bridge that need to be processed by the coding entity. The transmitted packet data is the result of the coding process, that need to be transmitted back to the OvS bridge. In both cases, coefficient data is present. The coefficient data address space is populated either from the FPGA or host PC, depending if encoding or decoding is done respectively. The receive and transmit data each occupy a total of $hN = 7 \times 64 = 448$ bytes, and the coefficient data occupies a total of $hh = 7^2 = 49$ bytes. These value are represented in hexadecimal for convenience as 0x1C0 and 0x31 respectively.

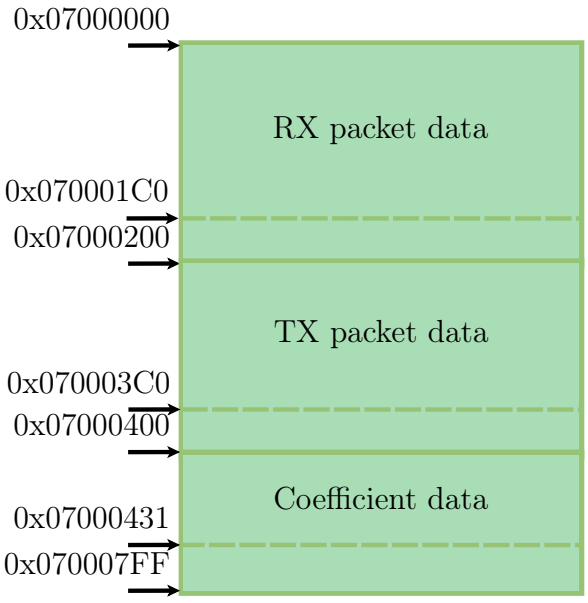

Figure 6.6: Memory map of FPGA on-chip RAM. The address space of the received, transmitted and coefficient data is shown.

The on-chip RAM provides an Avalon-MM slave signal which connects to the system interconnection fabric. Other modules can read and write to the RAM by using Avalon-MM master signals. We therefore continue the discussion onto how to control the on-chip memory.

6.4.4 Hardware module: memory control

The FPGA needs to transfer data between the on-chip RAM and the coding entities and therefore a memory controller is required. This is done using

the Avalon Memory-Mapped Master templates provided by Intel [71]. The templates provide the necessary logic to connect to the same interconnection fabric used by the PCIe reference design, using Avalon-MM master signals. The Avalon-MM master signals are connected to the on-chip RAM slave signal.

The templates include Verilog modules that are added to the PCIe reference design using the Quartus Platform Designer tool. A separate template is provided for the read and write functions. The read and write master templates are shown in Figs. 6.7 and 6.8 respectively from [71].

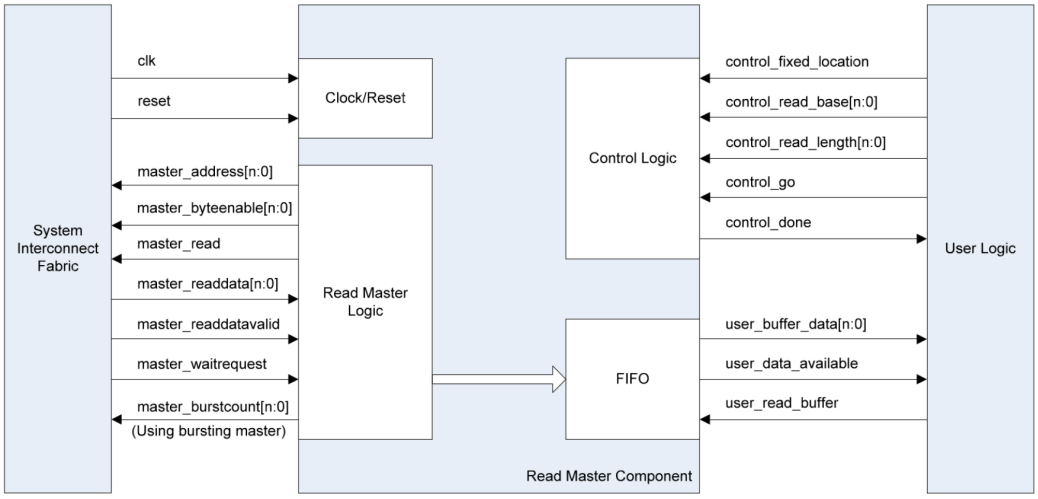

Figure 6.7: Avalon MM Master read template diagram showing user logic data and control signals, taken from [71].

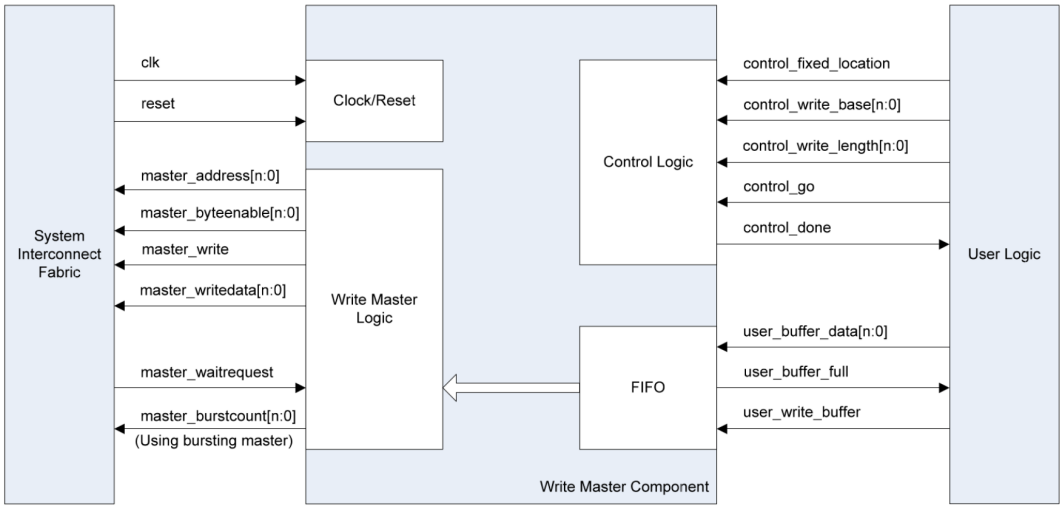

Figure 6.8: Avalon MM Master write template diagram showing user logic data and control signals, taken from [71].

The master templates make it easier to interface with the interconnect fabric, by providing control and data signals to connect to user logic. The signals are implemented using Avalon Conduit interfaces. This therefore allows signals to be exported to be used in the main HDL design files, outside of the Quartus Platform Design system. A summary of the exported signals for the read and write templates are given in tables 6.1 and 6.2 respectively.

Signal name	**Description**
control_fixed_location	Flag used to toggle read-address auto-increment.
control_read_base	32-bit read-address where the master reads from.
control_read_length	Number of 32-bit words to read.
control_go	Instructs the read master to begin reading.
control_done	Asserted by read master when all 32-bit words have been read.
user_buffer_data	32-bit word data read by read master.
user_data_available	Asserted by read master when read data is available.
user_read_buffer	User logic to acknowledge data has been read.

Table 6.1: Summary of control and data signals provided by the Avalon Memory-Mapped master read template.

Signal name	Description
control_fixed_location	Flag used to toggle write-address auto-increment.
control_write_base	32-bit write-address where the master writes to.
control_write_length	Number of 32-bit words to write.
control_go	Instructs the write master to begin writing.
control_done	Asserted by write master when all 32-bit words have been written.
user_buffer_data	32-bit word data written by write master.
user_buffer_full	Asserted by write master when buffer is full. No data should be written.
user_write_buffer	Asserted by user logic to write data to user buffer.

Table 6.2: Summary of control and data signals provided by the Avalon Memory-Mapped master write template.

The provided control and data signals are used to transfer data between the on-chip memory and the coding entities. Separate logic processes are created for the read and write functionality. The process pipelines for the read and write masters are shown in Figs. 6.9 and 6.10 respectively.

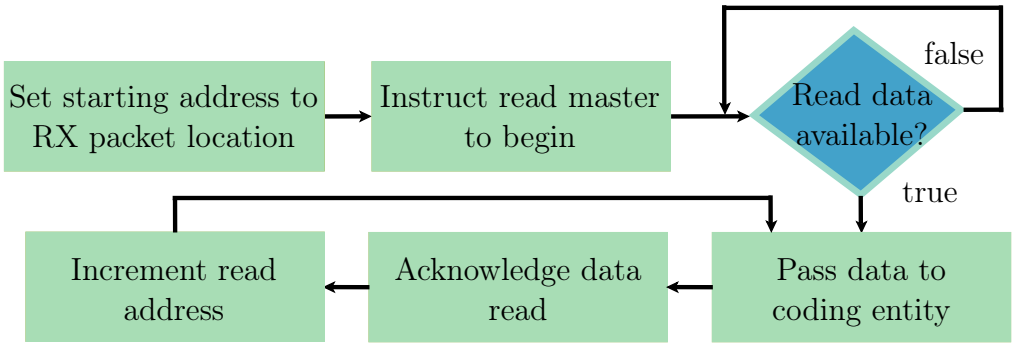

Figure 6.9: Master read flow diagram used to read data from on-chip memory. The data is passed to the coding entity to be processed.

Both the read and write process increment the address space manually within user logic, and do not make use of the auto-increment provided by the master templates. The *control_fixed_location* signal is deasserted to set auto-increment off. The read process begins by setting the starting memory address

to the receive packet location through the *control_read_base* signal. The read master is instructed to begin using the *control_go* signal. The process waits until the read data is available by monitoring the *user_data_available* flag. Once the data is available to read, it is passed to the coding entity from the *user_buffer_data* buffer. The user logic then acknowledges that the data has been read using the *user_read_buffer* flag, and increments the read address to read the next data segment. The loop continues until all data has been read successfully, then the read master is instructed to stop.

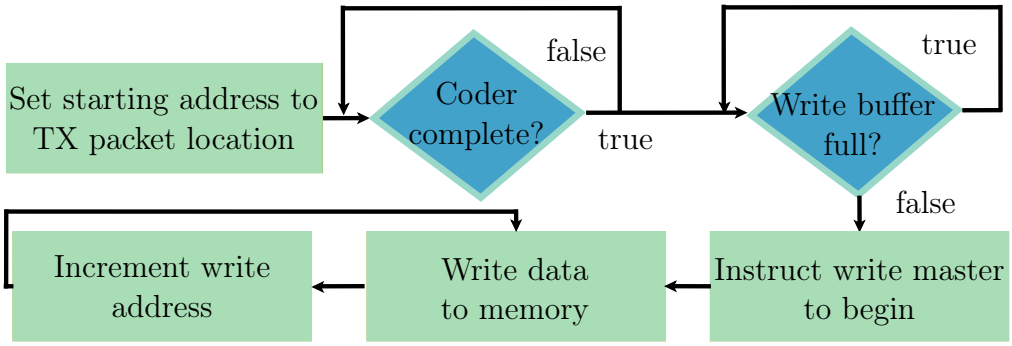

Figure 6.10: Master write flow diagram used to write data to the on-chip memory. The data is from the coding entity output.

The write process begins by setting the memory starting address to the transmitted packet location through the *control_write_base* signal. The user logic first checks to see if the coder is complete. If so, the *user_buffer_full* signal is used to check if the write buffer is not full. If the buffer is not full, the write master is instructed to begin by asserting the *control_go* signal. The data is written to memory from the *user_buffer_data* and the write-address is incremented. This continues until all data has been written to the on-chip memory.

6.4.5 Hardware module: control registers

While packet data transfer is handled by the on-chip memory and memory controller, another type of storage is needed to communicate coding control signals. Input and output registers on the FPGA need be made accessible to the host PC. The registers are used to set and get the coding start and finished flags. The start flag is set by the host PC, after the received packets have been written to the on-chip memory. The FPGA user logic waits until the start flag has been set to begin the coding process. The finished flag is set once the coding process is completed, to tell the host PC to read the results. Without these flags, the coding entity has no idea of when to begin, and might read the on-chip memory before packets have been received. Similarly, the host PC

does not know if the coding process is complete, and might read the on-chip memory before the results have been added.

A PCIe device makes use of Base Address Registers (BARs) to set the PCIe configuration address space between the host PC and the PCIe device. The operating system reads the BARs to configure the PCIe device to be used with the host PC. The BARs are also used to specify the amount of memory to be mapped into the host PC RAM. A typical PCIe device contains up to six 32-bit BARs. The BARs not used to specify memory mapping and configuration, are used to implement control signals between the FPGA user logic and the host PC.

The PCIe IP core is used to set and configure the BARs on the FPGA side, allowing the host PC access through the PCIe bus. To access the BARs from the FPGA user logic, the Parallel Input/Output (PIO) IP core is used. The PIO provides an Avalon-MM slave and Avalon Conduit signal. The slave signal is connected to the Avalon-MM master signals to access the BARs, and the Conduit signal is exported to connect to general input/output registers from the user logic. A separate PIO is created for the start and finished coding flags, using the Quartus Platform Designer.

Once the FPGA hardware module part of the hardware interface is completed, we can begin designing and implementing the software application that runs on the host PC. We start with the individual components to meet the software application requirements, and then combine them to create a overall application. The first component to implement is the userspace network interface.

6.4.6 Software application: userspace network interface

A method is required to communicate with OvS from the userspace. The universal TUN/TAP device driver, introduced in Chapter 2 allows for a userspace application to interface with the Linux networking stack. The OvS bridge can send packets to the TUN/TAP devices. The packet data can be accessed, and therefore manipulated to add network coding functionality.

Similarly to the VNF implementation in Chapter 4, a loopback function is created to verify the ability to receive and transmit packets. First a function is created to initialise the TAP interface. The initialization process is shown in Fig. 6.11.

In Linux, network devices are created using ioctl() system calls. An *ifreq* structure is used to contain information regarding the interface to be created [55]:

```
struct ifreq {
    char ifr_name[IFNAMSIZ]; /* Interface name */
    union {
        struct sockaddr ifr_addr;
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr;
        struct sockaddr ifr_netmask;
        struct sockaddr ifr_hwaddr;
        short ifr_flags;
        int ifr_ifindex;
        int ifr_metric;
        int ifr_mtu;
        struct ifmap ifr_map;
        char ifr_slave[IFNAMSIZ];
        char ifr_newname[IFNAMSIZ];
        char *ifr_data;
    };
};
```

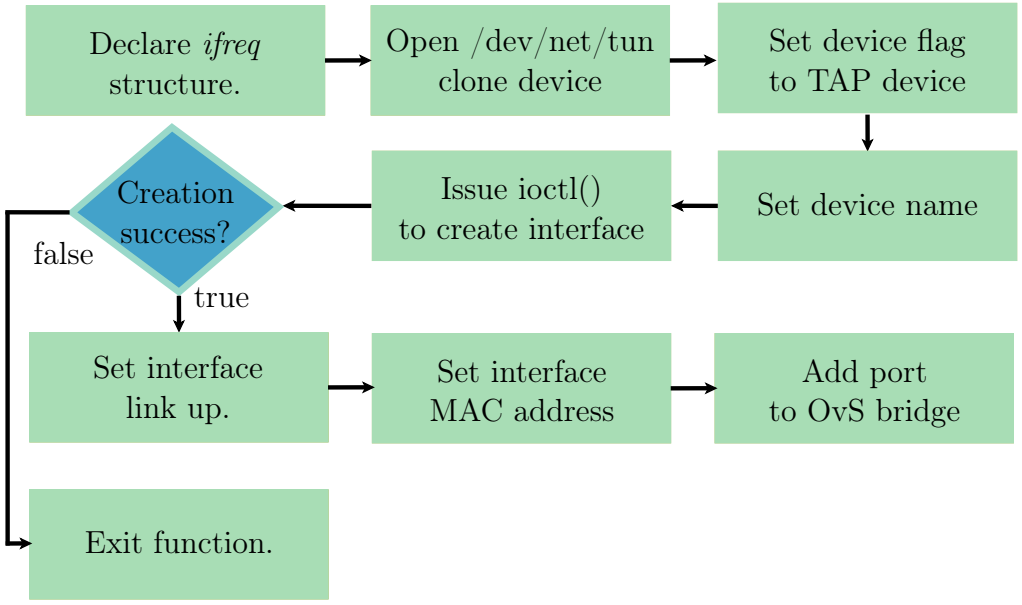

Figure 6.11: Flow diagram of TAP interface creation function.

The /dev/net/tun device is opened and returns a device file descriptor.

This device is known as a clone device, and is required when creating a TUN/-TAP interface. Next, the device name and TAP flag is set through the ifreq struct. The interface is created by performing a system call using the ifreq struct and clone device as parameters. If the interface is created successfully, the TAP device is configured to be used on the network. The *system()* C function is called as follows:

```
system("ip link set tapCoder up; ip link set tapCoder address
   02:01:02:03:04:08; ovs-vsctl add-port br0 tapCoder");
```

This sets the TAP device interface up, assigns a MAC address and then adds the interface port to the OvS bridge. The network interface can now be used to communicate with the OvS bridge.

Packet reception is done by calling the *read()* function on the TAP device file descriptor:

```
int nread = read(tapfd,tapBuffer,RX_BUFFER+TAP_HDR_LEN);
```

The function returns the number of bytes read, and stores the packet received in a separate buffer.

Packet transmission is done in a similar way using the *write()* function:

```
int nwrite = write(tapfd,tapBuffer,TX_BUFFER);
```

To verify the correct loopback functionality. One of the non-coding VMs is used to send ping packets to the TAP interface. The software application outputs the received packet and transmits it back. The VM runs the tcpdump utility to check the packet received. The packets received by the software application and non-coding VM are compared to the original packet to confirm that the loopback function is working as intended. Fig. 6.12 shows the network topology used to verify the loopback.

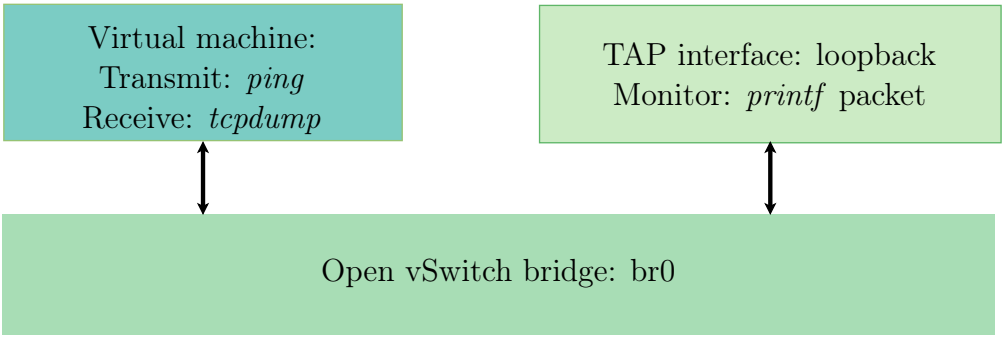

Figure 6.12: Network topology used to verify userspace TAP interface loopback function.

After the software application is capable of communicating with the OvS bridge, the next step is to communicate with the FPGA device. A method of interfacing the userspace with the PCIe bus is therefore required. The developers of the OpenVINO starter kit, Terasic provide a PCIe driver and C/C++ software library to connect to the FPGA device from the userspace.

6.4.7 Software application: PCIe, on-chip memory and register control

The Terasic PCIe library API provides a variety of functions to handle the PCIe connection, on-chip memory read and write, and BAR read and write. A summary of the important functions are given in table 6.3.

Terasic PCIe Function	**Description**
PCIE_Load()	Load PCIe driver and return handle.
PCIE_Open()	Open PCIe device with specified vendor ID, device ID and card index.
PCIE_Close()	Close handle associated with PCIe device.
PCIE_Read32()	Read 32-bit data from FPGA BAR address.
PCIE_Write32()	Write 32-bit data to FPGA BAR address.
PCIE_DmaRead()	Read DMA memory-mapped memory from PCIe device.
PCIE_DmaWrite()	Write data to DMA memory-mapped memory of PCIe device.

Table 6.3: Summary of Terasic PCIe library functions used to interface userspace software application with OpenVino starter kit.

The PCIe device is initialised using the PCIE_Load() function. The PCIE_Open() and PCIE_Close() functions are used to open and close the PCIe connection before and after the packet offload process respectively. Packets to be processed by the FPGA are transferred between the on-chip memory using the Terasic PCIE_DmaRead() and PCIE_DmaWrite() functions. Register control is implemented by modifying user logic BARs using the PCIE_Read32() and PCIE_Write32() functions.

6.4.8 Software application: design overview

The userspace networking interface is combined with the Terasic PCIe library to create an algorithm for the overall software application. Fig. 6.13 provides

an overview of the software application used to integrate OvS and the FPGA coders. First the TAP interface is created to allow for packet transfer. The PCIe driver is then loaded, and the connection is established. The application enters a loop and waits until all h packets are received.

The received packet data payloads are extracted and written to the on-chip memory using DMA. Next, the required coder entity is started by writing to a user implemented control logic start register. The software application waits until the coder is finished by reading the control logic finished register. Once the coder is completed, the resultant data is read from the on-chip memory, packetized and transmitted to the OvS bridge.

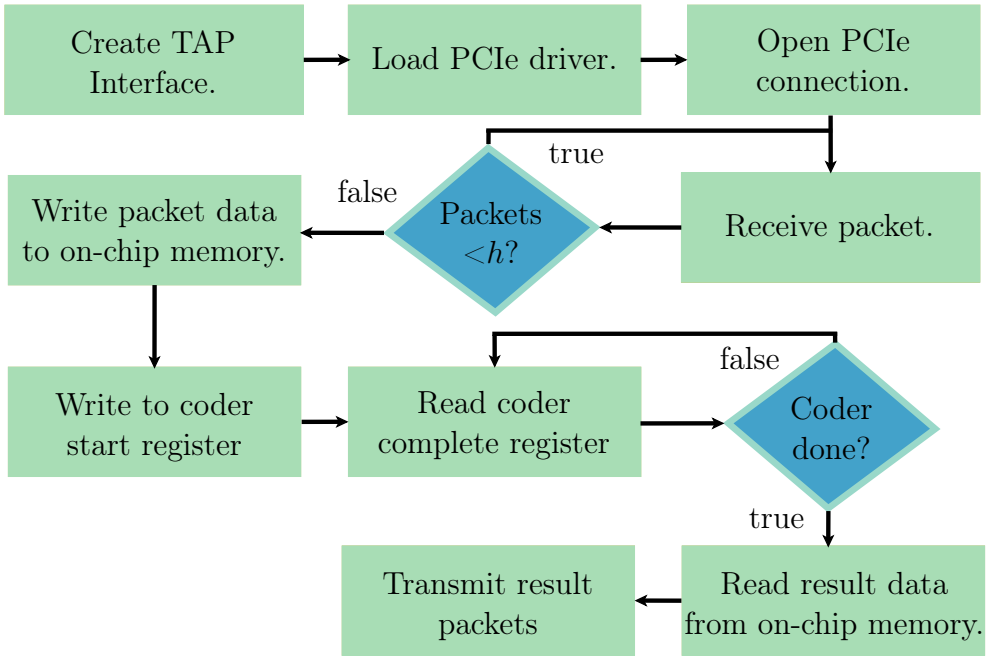

Figure 6.13: Flow of software application used to interface OvS with FPGA coders.

6.5 Summary

In this chapter we discussed the integration between OvS and the software and hardware network coding functions. The software layer integration makes use of the QEMU/KVM hypervisor to offload packets to the VNF for processing. OpenFlow flows are used to add the correct forwarding paths to the OvS bridge. The method of adding flows to the OvS bridge are discussed using the built in OvS command line utility.

The control plane is also discussed to meet the requirements of the switch being SDN compatible. The Ryn SDN controller is used to perform multicast snooping for the IGMP protocol. Details regrading the snooping algorithm are given.

The hardware layer integration is provided to offload packets to the FPGA and designed in two parts. The first part is a software application that runs on the host PC and is responsible for data transfer and communication between the OvS bridge and the FPGA. The software application implements the necessary network interface, PCIe and memory control.

Along with the software application, a hardware module is created for both the encoder and decoder that run on the FPGA device. The hardware modules encapsulate the coder entities and implement the PCIe module, shared memory, memory control and register control on the FPGA device to interface with the software application.

Chapter 7

Performance evaluation

7.1 Introduction

In the previous chapter we explained the integration between the Open vSwitch bridge and the network coding functions. In this chapter we use the integrated system to perform experimental tests on the final software and hardware network coding implementations, and evaluate the results. The evaluation of runtime, throughput, latency and packet delay variation is performed to meet objective three from section 1.4. The network coding configuration for both implementations use a generation size of $h = 7$ packets, containing $N = 64$ bytes in the Galois field $GF(2^8)$.

7.2 Runtime analysis: network coding only

An important specification for FPGA-based hardware design is the runtime. The runtime of both the FPGA and VNF coding functions are investigated and compared against each other. Runtime refers to the amount of time taken for the coding process to complete, independent of the network operations. This provides a metric based purely on the coding function performance. Runtime is an important measurement because network processing applications are timing critical and therefore smaller runtime values are desired to be able to process a greater amount of data.

Runtime measurements are not standardized and therefore to prevent misinformation, a simple diagram is constructed to convey what is meant by runtime in this context. Fig. 7.1 denotes the timing intervals for loading data in, coding, loading data out and overall operation.

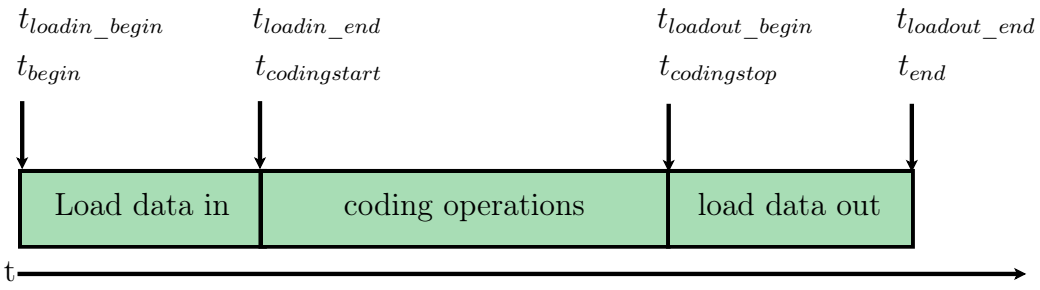

Figure 7.1: Coding runtime timing values.

The overall coding runtime is measured as,

$$t_{coding} = t_{codingstop} - t_{codingstart} \tag{7.1}$$

The coding runtime measurement is used to evaluate the throughput performance of the coding functionality. Throughput is the amount of data that is processed by the coder, within a set amount of time. The overall coding runtime is used to obtain the overall coding throughput measurement as,

$$T_{coding} = \frac{\text{number of bytes in generation}}{\text{coding runtime}} = \frac{hN}{t_{coding}} \tag{7.2}$$

Another useful metric is to determine the overhead of the loading operations. This overhead provides indication of the amount of time used loading data to and from the network coders, within the overall coding process. The overhead can be used to determine if loading is bottlenecking the coding process, and therefore optimised. The loading overhead is calculated as,

$$\begin{aligned} \text{Loading overhead} &= \frac{\text{overall time - coding only time}}{\text{overall time}} \\ &= \frac{(t_{end} - t_{begin}) - (t_{codingstop} - t_{codingstart})}{(t_{end} - t_{begin})} \end{aligned} \tag{7.3}$$

We begin by measuring the runtime performance of the VNF-based encoder and decoder function. The same tests are then performed on the FPGA coding implementation, and the results are compared.

7.2.1 VNF-based network coders

Aim: To determine the runtime of the VNF encoder and decoder coding operations. The runtime values are used to calculate the throughput and loading overhead. The VNF-based metrics are used as a comparative benchmark for the FPGA-based coding function runtime.

Test setup: The DPDK-based VNF coding function implementation from Chapter 4 is modified to record time values using the *clock()* C library function. The clock function returns the number of clock ticks since the start of the program. The number of seconds is obtained by dividing the clock ticks by the *CLOCKS_PER_SEC* C macro. The time values are recorded at the intervals shown in Fig. 7.1. The results are stored as comma separated values in a plain text file on the VNF VM.

The Scapy packet generator is used to send packets to the VNF coder, based on the coding values from Appendix 8.2. The VNF VM is allocated 4 logical cores and 8GB of RAM by the host PC hypervisor. A single generation of h packets is repetitively transmitted to the encoder and decoder VNF 20 000 times respectively. The final runtime values are obtained by using the mean value of all of the 20 000 tests.

Method:

1. Setup and start OvS bridge.
2. Assign internal port to bridge to stream packets to coding functions.
3. Start network coding VNF.
4. Bind VNF VM host network interface to DPDK.
5. Set OpenFlow flow on OvS bridge:

```
sudo ovs-ofctl del-flows br0
sudo ovs-ofctl add-flow br0 in_port="br0",actions=output
    :"vhp_codeVNF"
```

6. Start coding VNF with either encode or decode function.
7. Start Scapy packet generator script to send packets.
8. Run the Linux *awk* utility on the output benchmark file to get the mean of the runtime values:

```
awk -F',' '{sum+=$1; ++n} END { print "Mean: "sum"/"n"="
    sum/n }' < BENCHMARKS.txt
```

9. Use equations 7.1, 7.2 and 7.3 to determine the overall coding runtime, overall throughput and loading overhead for both the encoder and decoder VNF.

Result: The runtime timing intervals are given in table 7.1. The timing interval values are used to obtain the results for the overall coding runtime, overall

throughput and loading overhead in table 7.2. The start time $t_{begin} = 0$s in all test cases.

Coder	$t_{codingstart}$ (μs)	$t_{codingstop}$ (μs)	t_{end} (μs)
encoderVNF	9.180	30.945	44.076
decoderVNF	6.064	46.797	59.628

Table 7.1: Timing values for the encoder and decoder VNF coding functions.

Coder	t_{coding} (μs)	T_{coding} (Mbps)	**Loading overhead (%)**
encoderVNF	21.765	164.668	50.619
decoderVNF	40.733	87.988	31.688

Table 7.2: Runtime, throughput and loading overhead for the encoder and decoder VNF coding functions.

7.2.2 FPGA-based network coders

Aim: Determine the runtime of the FPGA encoder and decoder coding operations. The runtime values are used to calculate the throughput and loading overhead. The FPGA-based metrics are compared against the VNF-based coding functions.

Test setup: The OpenVINO starter kit is installed in the host PC PCIe slot. The FPGA hardware modules from the hardware layer integration in Chapter 6 are compiled and programmed onto the Cyclone V FPGA. The clock frequency of the modules are set to one of the default on board clocks, at 50MHz. The software application from Chapter 6 is run on the host PC. The Scapy packet generator is used to send a single generation of packets to the FPGA-based encoder and decoder. The generation is only sent once because due to the nature of FPGA hardware, the FPGA logic produces the same output timing intervals with subsequent iterations. The Quartus Signal Tap Logic analyzer is used to capture the signals on the FPGA during testing and the timing values from Fig. 7.1 are logged.

Method:

1. Start Quartus Prime software and open the coding hardware module project from Chapter 6.
2. Open the Signal Tap logic analyzer program from the Quartus suite.

3. Add coding entity signal nodes to Signal Tap logic analyzer. Set trigger to occur on the coding start register signal and set clock to the default 50MHz clock provided by the PCIe fundamental reference design, CLOCK_50_B3B.

4. Compile design to generate a SRAM object file (.sof) used to configure the FPGA device.

5. Program the FPGA using .sof file and the Quartus Programmer through JTAG.

6. Perform a soft reset of the host PC, to allow for the Linux OS to reinitialise the PCIe device. The FPGA must maintain a power source during the reset and therefore a hardware reset cannot be performed.

7. Once the host PC is restarted, re-open Quartus Prime and the Quartus Signal Tap logic analyzer.

8. Setup and configure the OvS bridge, and load Altera PCIe driver module.

9. Start the hardware integration software application from Chapter 6. This initialises the TAP device and the application waits for packets to be received.

10. Add OpenFlow flows to the OvS bridge. This is done after the TAP interface is initialised, otherwise OvS wont recognise the TAP port name.

11. Run the instance manager from the Signal Tap logic analyzer to begin the signal capture. The signal capture waits for the trigger to be set before beginning.

12. Run the Scapy packet generator script to send packets to the software application. The software application transfers the packet data to the FPGA and the trigger is set.

13. Open Signal Tap logic analyzer file to view timing results.

14. Calculate the overall coding runtime, overall throughput and loading overhead for both the FPGA-based encoder and decoder.

Result: The Signal Tap Logic analyzer result diagrams are given in Appendix 8.2. The runtime timing intervals for the FPGA-based network coders are shown in table 7.3. The timing interval values are used to obtain the results for the overall coding runtime, overall throughput and loading overhead are given in table 7.4. The start time $t_{begin} = 0$s in all test cases.

Coder	$t_{codingstart}$ (μs)	$t_{codingstop}$ (μs)	t_{end} (μs)
encoderFPGA	0.000	16.060	18.660
decoderFPGA	2.240	9.460	11.680

Table 7.3: Timing values for the encoder and decoder FPGA coding functions.

Coder	t_{coding} (μs)	T_{coding} (Mbps)	**Loading overhead (%)**
encoderFPGA	16.060	223.163	13.934
decoderFPGA	7.220	496.399	38.185

Table 7.4: Runtime, throughput and loading overhead for the encoder and decoder FPGA coding functions.

7.2.3 VNF and FPGA coder only: results comparison and conclusion

The runtime tests for the VNF-based coders show that the encoder and decoder take 21.765 and 40.733 microseconds to completed the coding process respectively. The overall coding throughput for the encoder is 164.668 Mbps, and 86.988 Mbps for the decoder. The decoding process therefore takes longer to execute than the encoder. This is expected from the literature review in Chapter 2, where it is mentioned that the decoding process is more computationally complex than the encoding process.

The loading overhead for the VNF-based encoder is 50.619%, and 31.688% for the decoder. This shows that the loading overhead takes up a large amount of the runtime of both coders, but especially the encoder. Reducing the loading overhead would enable the coding runtime to decrease, therefore increasing the throughput. Future work can aim to reduce the loading overhead as a result. Possible ways to reduce the loading overhead within the VNF implementation would be to perform coding operations directly from the receive queue buffers, and not copy the packets into separate encoding and decoding rings. This would reduce the operations of placing packets into the coding rings, and having to retrieve them at a later stage for coding. Not using separate coding rings would however, require a new way of keeping track of all the packets in each generation.

The runtime tests for the FPGA-based coders show that the encoder and decoder take 16.06 and 7.22 microseconds to complete the coding process respectively. The runtime tests are used to calculate the coding throughput of 223.163 and 496.399 Mbps for the encoder and decoder respectively. The FPGA coding throughputs are much higher than that of the VNF-based implementation. Specifically, the FPGA implementation provides a speedup of 1.36 and 5.71 over the VNF encoder and decoder respectively. Comparing

the FPGA-based coder throughput to past work done in [14], shows that the FPGA-based decoder achieves a speedup of 7.64 over the maximum decoder throughput by Kim, et al. of 64.98 Mbps. The work done by [14] does not implement an FPGA-based encoder to be used for comparison.

One interesting result to note with the FPGA implementation is that the encoder takes longer to perform the coding operation than the decoder. This is unexpected because in software implementations such as those with the VNF coders, the decoder almost always takes longer than the encoder. This is due to the increased complexity of the RLNC decoding process. The difference in runtime between the FPGA-based encoder and decoder is due to the different design approaches used.

The FPGA encoder design performs all the required matrix multiplication operations while the incoming data is segmented into the encoder module from the host PC. The encoder does not wait for the entire incoming packet generation to arrive and uses an *on-the-fly* coding approach. The original design goal of using this approach is that the encoder does not have to wait for the loading process to complete to begin the coding process. This design only requires $4h$ multipliers compared to the possible Nh^2 multipliers of using a *full-vector* encoding approach, therefore also reducing FPGA resource usage. The problem with this approach, as seen from the results, is that the maximum amount of parallel multiplication operations are not performed.

The FPGA decoder makes use of full-vector coding and only begins the decoding process once all packet data has been loaded in. The decoder is able to maximize the number of parallel multiplication operations and therefore achieves a lower runtime than the encoder. The decoder also however uses more FPGA resources. A trade-off is therefore observed between using an on-the-fly verses a full-vector approach. The full-vector approach increases the potential to implement more parallel multiplication operations, but at the expense of increased loading time and resource utilization. Future work could therefore be to implement a full-vector encoder and a on-the-fly decoder to further study the trade-off between the two approaches.

The loading overhead of the FPGA-based implementation is 13.934% for the encoder, and 38.185% for the decoder. The FPGA-based encoder loading overhead is relatively low compared to the FPGA decoder, and the VNF implementations because the FPGA encoder starts coding as the data is input to the FPGA and therefore the encoder t_codingstart is the same as t_begin, at 0s.

In conclusion, the FPGA-based encoder and decoder provide a substantial increase in coding throughput compared to the VNF implementations, as well as past FPGA-based network coding results. This result showcases that FPGAs are able to increase the performance of the RLNC algorithm and can be used to successfully accelerate network coding.

7.3 Network throughput, latency and packet delay variation

Throughput, latency and packet delay variation (PDV) are important network performance measurements to determine the quality of a network transfer. Network performance measurement definitions vary between different network configurations and testing scenarios. It is therefore important to begin by explicitly describing what exactly is meant by each term.

An end to end network topology is used for this test and includes a source, intermediate node and a receiver. Packets are transmitted from the source and either forwarded, encoded or decoded by the intermediate node. The resultant packets from the intermediate node are received by the receiver.

The transmitted packets,

$$p_{tx,0}, p_{tx,1}, ..., p_{tx,h}$$

are each transmitted by the source at the time intervals

$$t_{tx,0}, t_{tx,1}, ..., t_{tx,h}$$

corresponding to the first bit of each transmitted packet. Similarly at the receiving end, the last bit of the received packets,

$$p_{rx,0}, p_{rx,1}, ..., p_{rx,h}$$

are received at the time intervals

$$t_{rx,0}, t_{rx,1}, ..., t_{rx,h}$$

The timing intervals are used to obtain the overall network throughput, latency and PDV. The network throughput, in this context refers to the actual amount of data that is transferred through the network within a period of time. The network throughput is similar to the coding throughput, but includes consideration for the network integration, transfer and layers.

The network throughput is calculated over the timing interval between the last received packet and the first transmitted packet as,

$$T_{network} = \frac{hN}{t_{rx,h} - t_{tx,0}} \tag{7.4}$$

Latency is the amount of time taken for a packet to transfer from the source to the receiver. The time taken is denoted as $t_{latency}$ and calculated by subtracting the transmitted time t_{tx} from the received time t_{rx} to obtain,

$$t_{latency} = t_{rx} - t_{tx} \tag{7.5}$$

PDV, or "jitter" is the variation in latency and is measured in seconds. A network with consistent latency has no packet delay variation. Packet delay variation is an important network performance metric for applications that require consistently timed packet delivery, such as multimedia, VoIP and video streaming. Packet delay variation is calculated from all the latencies within a stream as,

$$PDV = \frac{\sum t_{latency}(i+1) - t_{latency}(i)}{h-1}, \qquad 0 \leq i < (h-1) \tag{7.6}$$

The throughput, latency and PDV are calculated in four different testing scenarios: VM using a VNF coder, physical host using a VNF coder, VM using an FPGA coder and physical host using an FPGA coder.

7.3.1 VNF-based network coders: end-to-end

Aim: To determine the throughput, latency and PDV of the VNF-based network coders in a real packet-based network.

Test setup: The DPDK-based coding VNF is connected to a non-coding VM and a physical host through the OvS bridge shown in Fig. 7.2. Packets are transmitted from the host to the VNF coding VM and back again to the host. The Scapy packet generator is run on the host and used to transmit packets to the VNF. The network performance metrics are calculated for the generation of h packets and repeated by streaming to the coding VNF 20 000 times. The tcpdump utility is run on the same host and is used to capture and record both the transmitted and received packet timestamps in *pcap* files. Two Scapy-based scripts are written to analyze the pcap files and calculate the network performance metrics. The packet generator and packet analyzer scripts are given in Appendix 8.2.

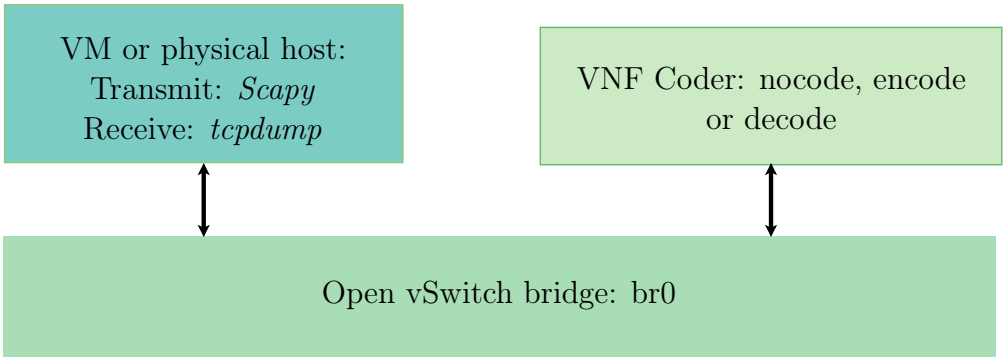

Figure 7.2: VNF-based network coding end-to-end test topology.

Method:

1. Setup and start OvS bridge.
2. Start non-coding VM or physical host used to transmit and receive test packets.
3. Start VNF VM to perform coding functions.
4. Set OpenFlow flows on OvS bridge:

```
sudo ovs-ofctl del-flows br0
sudo ovs-ofctl add-flow br0 in_port="hostport",actions=
   output:"VNFport"
sudo ovs-ofctl add-flow br0 in_port="VNFport",actions=
   output:"hostport"
```

5. Bind VNF internal network interface to DPDK.
6. Start VNF coding function with either nocode, encode or decode operations.
7. Start tcpdump on non-coding VM or physical host to log transmitted and received packet timestamps in a pcap file.

```
sudo tcpdump -tttt -i ens4 -w testresult.pcap
```

8. Start Scapy packet generator script to send packets.
9. Obtain pcap file from non-coding VM or physical host and use Scapy packet analyzer script to calculate throughput, latency and PDV.

Result:

Coder	$T_{network}$ (Mbps)	$t_{latency}$ (μs)	PDV (μs)
nocoderVNF	2.947	45.374	1.748
encoderVNF	2.858	658.813	318.587
decoderVNF	2.809	711.357	337.967

Table 7.5: Throughput, latency and jitter results for VNF-based coding function in PHY to PHY test.

Coder	$T_{network}$ (Mbps)	$t_{latency}$ (μs)	PDV (μs)
nocoderVNF	2.701	18.306	1.954
encoderVNF	2.844	647.191	338.373
decoderVNF	2.743	670.431	340.996

Table 7.6: Throughput, latency and jitter results for VNF-based coding function in VM to VM test.

7.3.2 FPGA-based network coders: end-to-end

To take note: The FPGA integration implementation from Chapter 6 is only implemented to provide a working proof of concept. The FPGA modules are only capable of producing the correct results from a single generation of packets, as done in the runtime tests. This is because the necessary logic to reset the encoder and decoder module registers is not implemented. However, the on-chip memory on the FPGA device can still be written to and read from consecutively. The coding operations still complete on the FPGA, but subsequent results are incorrect.

The assumption is made that implementing the necessary reset logic will not alter the end-to-end results. This is because the coding process on the FPGA completes faster than two consecutive C program function calls. Specifically in Fig. 6.13, when the function is called to write data to the on-chip memory using DMA, the FPGA encoder and decoder operations complete before the function to read the coder complete register, is called. As a result, the end-to-end stream tests are still conducted with the FPGA integration hardware modules.

Aim: To determine the throughput, latency and PDV of the FPGA-based network coders in a real packet-based network. To determine the overhead of the packet data memory transfer and networking operations. The test is used to evaluate the performance of the hardware layer integration used to connect the FPGA device to the OvS bridge.

Test setup: The OpenVINO starter kit is installed in the host PC PCIe slot. The FPGA hardware modules from the hardware layer integration in Chapter 6 are compiled and programmed onto the Cyclone V FPGA. The clock frequency of the modules are set to one of the default on board clocks, at 50MHz. The software application from Chapter 6 is run on the host PC.

The Scapy packet generator is run on a non-coding VM or physical host, to send a generation of packets to the FPGA-based encoder and decoder, repetitively 20 000 times. The tcpdump utility is run on the same host and is used to capture and record both the transmitted and received packet timestamps in *pcap* files. The network performance metrics are calculated from the pcap timestamps.

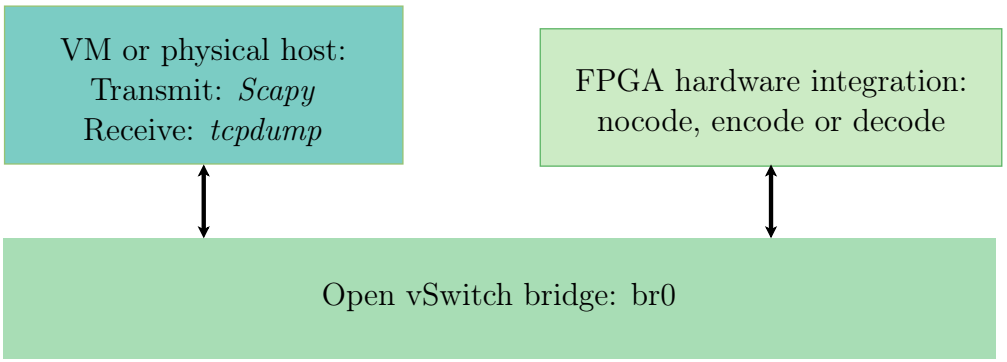

Figure 7.3: FPGA-based network coding end-to-end test topology.

Method:

1. Start Quartus Prime software and open the coding hardware module project from Chapter 6.
2. Compile design to generate a .sof used to configure the FPGA device.
3. Program the FPGA using .sof file and the Quartus Programmer through JTAG.
4. Perform a soft reset of the host PC, to allow for the Linux OS to reinitialise the PCIe device. The FPGA must maintain a power source during the reset and therefore a hardware reset cannot be performed.
5. Start non-coding VM or physical host used to transmit and receive test packets.
6. Setup and configure the OvS bridge, and load Altera PCIe driver module.
7. Start the hardware integration software application from Chapter 6. This initialises the TAP device and the application waits for packets to be received.
8. Add OpenFlow flows to the OvS bridge:

```
sudo ovs-ofctl del-flows br0
sudo ovs-ofctl add-flow br0 in_port="hostport",actions=
   output:"TAPport"
sudo ovs-ofctl add-flow br0 in_port="TAPport",actions=
   output:"hostport"
```

9. Start tcpdump on non-coding VM or physical host to log transmitted and received packet timestamps in a pcap file.

10. Run the Scapy packet generator script on the non-coding VM or physical host to stream packets to the software application. The software application transfers the packet data to the FPGA.

11. Obtain pcap file from non-coding VM or physical host and use Scapy packet analyzer script to calculate throughput, latency and PDV.

Result:

Coder	$T_{network}$ (Mbps)	$t_{latency}$ (μs)	PDV (μs)
nocoderFPGA	2.827	127.264	10.045
encoderFPGA	2.106	976.430	243.267
decoderFPGA	2.110	975.444	242.899

Table 7.7: Throughput, latency and jitter results for FPGA-based coding function in PHY to PHY test.

Coder	$T_{network}$ (Mbps)	$t_{latency}$ (μs)	PDV (μs)
nocoderFPGA	2.605	73.519	9.235
encoderFPGA	2.000	970.107	268.378
decoderFPGA	2.014	966.418	271.625

Table 7.8: Throughput, latency and jitter results for FPGA-based coding function in VM to VM test.

7.3.3 VNF and FPGA coder end-to-end test: results comparison and conclusion

Loopback (nocoder) result discussion:

The end-to-end test of the VNF-based coding functions is done with both a VM and a physical host. The first test performed evaluates the DPDK coding pipeline of the VNF, and the transmitted packets are looped back to the host without any coding operations (nocoder). This serves as a baseline for the throughput, latency and PDV to compare to the encoder and decoder functions. The nocoder throughput results for using the physical and VM hosts are 2.947 and 2.701 Mbps, with latencies of 45.374 and 18.306 microseconds respectively. The PDV of the VM and physical host are similar at 1.954 and 1.748 microseconds respectively. A low latency and PDV is expected of both hosts as no coding is being performed, and therefore the packets are not being held up for processing.

The nocoder network throughput results for the FPGA-based coding functions are given as 2.827 and 2.605 Mbps for the physical and VM hosts respectively. The latency of the physical host nocoder is 127.264 microseconds, and 73.519 microseconds for the VM host. The PDV values are low at 10.045 and 9.235 microseconds for the physical and VM host respectively.

The nocoder tests between the VNF and FPGA coders show that network loopback latency and PDV performance of the FPGA integration is worse than that of the VNF interfaces. This is most likely due to the VNF coders using DPDK, while the FPGA coder integration uses the TUN/TAP interface. DPDK provides a performance increase by bypassing the traditional Linux networking stack. Future work should therefore implement the FPGA network interface using DPDK instead of TUN/TAP.

Encoder and decoder results discussion:

The network throughput performance of both the VNF and FPGA-based coders, for both the VM and physical host test are shown to be substantially worse than those obtained from the coding only tests. All of the network throughput values are between 2 and 3 Mbps. This indicates that there is an overall limiting factor to either the tests or coding implementations. We therefore conduct further investigation before discussing additional results and conclusions.

The Pcap result files are analysed to calculate separate throughput rates. The specific rates calculated are input rate, output rate and the rate between receiving and transmitting packets (overall coding pipeline, without networking operations), as shown in equations 7.7, 7.8 and 7.9 respectively:

$$T_{input} = \frac{hN}{t_{tx,h} - t_{tx,0}} \tag{7.7}$$

$$T_{ouput} = \frac{hN}{t_{rx,h} - t_{rx,0}} \tag{7.8}$$

$$T_{between} = \frac{hN}{t_{rx,0} - t_{tx,h}} \tag{7.9}$$

The results obtained from the input, output and in-between throughputs are given in tables 7.9 and 7.10 for the VNF and FPGA tests respectively.

Coder	$T_{network}$ (Mbps)	T_{input} (Mbps)	T_{output} (Mbps)	$T_{between}$ (Mbps)
nocoderVNF_PHY	2.947	3.061	3.061	Nah
nocoderVNF_VM	2.701	2.740	2.739	Nah
encoderVNF_PHY	2.858	3.098	85.903	64.584
encoderVNF_VM	2.844	2.978	186.167	95.550
decoderVNF_PHY	2.743	3.022	552.587	30.469
decoderVNF_VM	2.809	2.987	1152.039	51.730

Table 7.9: Overall network, input, output and in-between throughputs for VNF coders end-to-end tests.

Coder	$T_{network}$ (Mbps)	T_{input} (Mbps)	T_{output} (Mbps)	$T_{between}$ (Mbps)
nocoderFPGA_PHY	2.827	3.141	3.142	Nah
nocoderFPGA_VM	2.605	2.752	2.753	Nah
encoderFPGA_PHY	2.106	3.098	11.698	15.039
encoderFPGA_VM	2.000	2.816	10.745	19.252
decoderFPGA_PHY	2.110	3.104	11.736	15.007
decoderFPGA_VM	2.014	2.827	11.261	18.545

Table 7.10: Overall network, input, output and in-between throughputs for FPGA coders end-to-end tests.

The results from the further investigation show that the input rate is the overall limiting factor for both the VNF and FPGA coding end-to-end tests. This is most likely due to the use of the Scapy packet generator. The Scapy packet generator was used because the decoder tests required using the 0x2020 Ethernet type and therefore full manipulation of the Ethernet layer was required, which Scapy provides. In future work, a possible alternative packet generator would be MoonGen. MoonGen is a high speed packet generator based on DPDK, that provides the same level of flexibility as with Scapy [72].

In both VNF and FPGA networking implementations, the VM host is able to obtain a faster throughput and lower latency than the physical host. This is likely because the physical host makes use of a 5400 RPM hard disc drive (HDD) compared to the VM host, which is run on a solid state drive (SSD), of the server. SSDs are known to provide faster read/write speeds compared to HDDs. Another possible reason for the performance difference is the processing speed of the logical cores of the VM hosts compared to the physical hosts. The VM is also on the same server as the OvS bridge, and therefore packets do not have to traverse through external networking equipment (server NIC, Ethernet cable and physical host NIC) as with the physical host.

The between throughput results are taken over the time period after all the packets have arrived, and before the resultant packets are transmitted back. This result is essentially the throughput over which the coding functions pipeline the packet data to and from the coders, including the time taken for the coding operations to complete. The between throughput provides evaluation regarding the integration between the coding functions and OvS, without consideration for the networking layers.

To determine the impact of the OvS integrations, we calculate the percentage in which the coding only throughput is reduced when integrating the coders with OvS, without consideration for the networking layers in table 7.11 and 7.12.

Coder	T_{coding} (Mbps)	$T_{between}$ (Mbps)	Reduction (%)
encoderVNF_PHY	164.668	64.584	60.78
encoderVNF_VM	164.668	95.550	41.97
decoderVNF_PHY	87.988	30.469	65.37
decoderVNF_VM	87.988	51.730	41.21

Table 7.11: Throughput performance reduction for VNF coders.

Coder	T_{coding} (Mbps)	$T_{between}$ (Mbps)	Reduction (%)
encoderFPGA_PHY	223.163	15.039	93.26
encoderFPGA_VM	223.163	19.252	91.37
decoderFPGA_PHY	496.399	15.007	96.98
decoderFPGA_VM	496.399	18.545	96.26

Table 7.12: Throughput performance reduction for FPGA coders.

The results from tables 7.11 and 7.12 show that the FPGA coding performance suffers the most from the OvS integration. The maximum reduction is with the FPGA decoder in the physical host test, at 96.98%. The smallest reduction is seen with the VNF decoder in the VM host test, at 41.21%.

The lower between-throughput of the FPGA coders could likely be due to the use of the Terasic provided PCIe and DMA library functions. Future work could implement custom PCIe drivers and software integration functions to try and reduce the overhead of the FPGA coder and OvS integration.

The output throughput does not limit the performance of the VNF-based coding functions ($T_{output} > T_{between}$), but does limit the possible FPGA coding functions ($T_{output} < T_{between}$). This is most again likely due to the VNF coders using DPDK, while the FPGA coder integration uses the TUN/TAP interface.

The FPGA encoder and decoder results for the physical test are the same, and similarly with the VM tests. This is because the FPGA coder and OvS

integration pipeline for the encoder and decoder are almost identical in operation. This result is therefore expected.

In conclusion, the end-to-end results show that both the VNF and FPGA coding functions are bottlenecked by their respective integrations with the networking layers. This indicates that future work should focus on improving techniques to reduce the amount of time taken to offload packets to the coding functions. The results also show that the use of VMs are a viable alternative to using physical machines and are therefore a more cost effective approach, in which equipment costs to research network coding can be reduced.

7.4 FPGA resource utilization

Aim: The resource utilization provides an indication of how much of the device is being used by each entity and how much space remains on the device for additional coders. The number of dedicated logic registers gives indication to how the FPGA is utilising dedicated resources to construct the coders. These metrics are useful as it allows for optimization in the HDL implementation as well as compiler settings to construct a more efficient design.

Test setup: The results are taken from the compilation report provided by Quartus. The values for the *net_encoder_pci* and *net_decoder_pci* entities are obtained from the compiler flow summary, and after a full compilation is completed. The other entities are calculated by summing the resource usage for each entity instance from the "Fitter Resource Utilization Usage by Entity" section of the compilation report. The results show the total resources used by each entity.

Result: A summary of the resource utilization for the FPGA network encoder and decoder are shown in tables 7.13 and 7.14 respectively, showcasing the number of resources used out of the possible total available by the device. The first column is the name of the HDL entity itself. The second column is the total number of Adaptive Logic Modules (ALMs) used by all the instances of that entity in the complete coder. ALMs are the basic building blocks used to construct FPGA hardware logic and are specific to the device family. In this case it is the Cyclone V device. According to Intel, each ALM supports up eight inputs and outputs, two register logic cells and two combinational logic cells [73]. The third column is the number of Dedicated Logic Registers (DLRs) used and is a representation of the logic that is dedicated for use as registers only. The DLRs are constructed using ALMs where a single ALM can be used to construct two dedicated registers. Therefore DLRs are simply ALMs that only use the register resources, where each register can store a single bit. The fourth column is the amount of dedicated M10K type memory blocks used in the implementation.

Entity	**ALMs (113560)**	**DLRs (227120)**	**Memory (12492800)**
fifo32x128	15 (<1%)	23 (<1%)	4096 (<1%)
prngen	15 (<1%)	61 (<1%)	0 (0%)
gfmul	160 (<1%)	273 (<1%)	0 (0%)
net_encoder	2905 (2.56%)	4504 (1.98%)	4096 (<1%)
net_encoder_pci	10507 (9.25%)	21608 (9.51%)	2365200 (18.93%)

Table 7.13: Encoder resource utilization cumulative summary

Entity	**ALMs (113560)**	**DLRs (227120)**	**Memory (12492800)**
gj_mulsubRow	6873 (6.05%)	20724 (9.12%)	0 (0%)
gj_divRow	915 (< 1%)	2160 (<1%)	0 (0%)
gj_elimination	19703 (17.35%)	37576 (16.54%)	0 (0%)
net_decoder	20170 (17.76%)	38303 (16.87%)	0 (0%)
net_decoder_pci	27590 (24.30%)	56422 (24.84%)	2860816 (22.90%)

Table 7.14: Decoder resource utilization cumulative summary

With regards to the encoder only (without PCIe logic) utilization, it shows that the only dedicated memory blocks used are by the FIFO. This is expected as the FIFO is used as a data buffer and is therefore would be regarded as a storage module. According to Intel, the ALM metric is best used to determine resource utilization [73]. The number of DLRs however, is also useful as it gives indication of how many registers are used to store the symbols while they are processed. The results of the multipliers to be added are stored as registers within the *net_ encoder* module and the result is shown by the number of DLRs used. The resource use of the pseudo-random number generator *prngen* and the Galois field multipliers *gfmul* are relatively minimal. This shows that the majority of the resource usage is for storing the the multiplier and summation results in registers.

The encoder entity only uses 2.56% of the available FPGA. Adding the PCIe integration logic from Chapter 6 to the encoder entity increases the ALM usage to 9.25% and the Memory usage to 18.93%. Extrapolating this result to use all resources on the device, and disregarding any input and output bottlenecks, the FPGA could potentially have 5 encoders running simultaneously.

The decoder resource usage is summarized by each type of row operation used in the Gauss-Jordan elimination entity *gj_elimination*. The resource usage of *gj_elimination* is shown with the overall decoder entity to showcase the resource usage by the 32-bit segmentation process. The decoder is significantly more resource intensive. This is expected because of the size and complexity of the decoding operation compared to the encoder. A decoder stores all the row elements used during Gauss Jordan elimination in registers and not in dedicated memory blocks. This is shown in table 7.14 as the memory utilization is 0% while the DLR usage is much higher at 16.87%. The ALM utilization is 17.76% and shows that the decoder uses almost seven times more resources than the encoder.

Adding the PCIe integration logic from Chapter 6, increases the decoder ALM usage to 24.30% and the memory usage to 22.90% percent. Extrapolating the result to use all resources, the FPGA could potentially have 4 decoders running simultaneously.

7.5 Maximum operating frequency

Aim: An important consideration for FPGA design are the timing characteristics. We investigate the maximum possible clock frequency (Fmax) of the FPGA encoder and decoder implementations. The clock frequency limits the processing speed of the coding modules. It is therefore important to obtain Fmax to determine the maximum possible throughputs.

Test setup: The Fmax values are taken from the Quartus Timing Analyzer. The values are presented as process, voltage and temperature variations for the lower (0C) and upper (85C) temperature limits of a slower chip at a lower voltage (1100mV).

Result: The Fmax values are given in table 7.15.

Entity	**Fmax (0C)**	**Fmax (85C)**
net_encoder_pci	84.03 MHz	81.65 MHz
net_decoder_pci	75.06 MHz	73.02 MHz

Table 7.15: Maximum operating frequencies of FPGA encoder and decoder implementations, for the slow 1100mV 0C and 85C models.

The runtime tests for the FPGA encoder and decoder are performed at 50 Mhz. The frequencies of the encoder and decoder implementation can be increased to run at those in table 7.15. Using the maximum possible frequencies, and making use of all four PCIe lanes to implement four encoders or decoders

on the FPGA at a time, the possible coding throughput of the encoder and decoder is calculated as following:

$$T_{coding\_max} = \frac{(\text{Number of coders})(\text{Fmax})(T_{coding})}{(\text{old clock frequency})}$$

to obtain a maximum possible coding throughput for the encoder and decoder as 1.5 and 2.98 Gbps respectively.

7.6 Summary

This chapter provides results and evaluates the VNF and FPGA network coding implementations. A runtime analysis is performed to evaluate the performance of the network coding functions, independent of networking operations. What is meant by runtime analysis is discussed, as well as how to use runtime to calculate the coding throughput and loading overhead.

The aim, test setup, method and runtime results are provided for both the VNF and FPGA-based network coders. The VNF-based encoder and decoder achieve a coding throughput of 164.668 and 87.988 Mbps respectively. The FPGA-based encoder and decoder achieve a coding throughput of 223.163 and 496.399 Mbps respectively. The results of the VNF and FPGA coder are compared and evaluated further.

Further end-to-end tests are performed regarding the network throughput, latency and packet delay variation. What is meant by each term, as well as how to calculate the network performance metrics are provided. The test are performed using both VM and physical hosts, for both the VNF and FPGA coding implementations. The results show that both the VNF and FPGA-based network coders are bottlenecked by their respective integration pipelines.

The FPGA-based coder resource utilization and maximum frequency are evaluated. The results show that five and four respective encoder or decoder modules can fit on the OpenVINO starter kit at a time. The maximum frequencies of the encoder and decoder module are used along with the number of possible coders to determine the maximum possible coding throughput. The result shows that the encoder and decoder can achieve a maximum coding throughput of 1.5 and 2.98 Gbps, given the resource limitations of this device.

Chapter 8

Conclusion

8.1 Overview

In conclusion, the thesis achieves the objectives of designing and implementing a network coding capable switch in both a software and hardware-based environment. The software-based network coding functions are created as VNFs that are deployed in an SDN environment as required, to meet objective 1. The hardware-based networking coding functionality is implemented using an FPGA device, to meet objective 2. Both software and hardware implementations are integrated together using the OpenFlow-based SDN bridge, OvS. The overall platform is designed to be run on a general purpose PC and allows for network coding to be evaluated in both physical and virtual network environments, with physical or VM hosts. The platform allows for objective 4 to be met.

The network coding implementations are evaluated together within a real packet-based network, to meet objective 3. The VNF-based network coding functions are able to achieve a coding throughput of 164.668 and 87.988 Mbps for the encoder and decoder respectively. The FPGA-based network encoder and decoder achieve a throughput of 223.163 and 496.399 Mbps respectively, at a clock speed of 50 MHz, using a single PCIe 1.0 lane. The FPGA resource utilization and maximum clock frequency results show that the encoder and decoder could be implemented to achieve a throughput of 1.5 and 2.98 Gbps respectively.

The VNF and FPGA-based network coding functions are shown to be able to increase network coding performance by offloading packets to the coding functions. The coding functions are limited however by the integration with the OvS bridge. Future work should therefore focus on methods of alleviating this bottleneck to improve techniques for offloading packet data to the network coding functions with as little delay as possible.

Links to the software and HDL components of this thesis can be obtained on GitHub for *OvS-DPDK-Coding-Switch* [57] and *RLNC_ VNF* [58].

8.1.1 Contributions

The following contributions are made throughout this thesis:

- An FPGA network coding encoder and decoder that can be used within a practical SDN network. This thesis presents FPGA logic for a RLNC encoder which has not yet been shown in literature. The thesis also showcases the design and implementation of integrating FPGA network coding encoder and decoder functions with the OvS OpenFlow SDN switch, and therefore a real packet-based network.
- A VNF-based network encoder and decoder that are implemented using the DPDK software library.
- An all in one research platform to evaluate network coding in a real packet-based network, using both physical and virtual networking spaces. The networking functions are integrated with OvS within an SDN environment.
- A network coding selection scheme to determine when to perform network coding in a real packet-based network.
- An SDN controller program design to implement multicast snooping within an SDN environment.

8.1.2 Limitations and improvements

The network coding functions are created with functionality as the primary objective. The system is created to determine the feasibility of implementing software and hardware-based network coding within a network switch. The coding implementations are therefore not perfect and have several limitations. The most important limitations are discussed, as well as some possible methods of improving or rectifying them. The limitations are as follows:

- The FPGA integration can only offload a single generation of packets to the FPGA-based encoder and decoder modules successfully. Further streams of packets received by the coding functions do not yield the correct coding results. This is because the necessary reset logic is not implemented to reset all the coding sub-modules and therefore previous generation packet data remains. This can be solved be implementing consistent reset logic on all the subcomponents, that can be triggered from the top level coding entity.
- The decoder integration coefficients are hard coded onto the FPGA device. This was done to save time during the FPGA implementation. The functionality of loading packet data to and from the FPGA device

are implemented and demonstrated to provide a proof of concept, and the coefficient data is still written to the shared on-chip memory. The decoder produces the correct coding output, but is therefore limited to only being able to decode packets that use the same coding coefficients as on the device. To fix this, the same logic that is used to load packet data to the coding entity from the on-chip memory needs to be implemented to do the same for the coefficient data. This can be done using the Avalon-MM Master Read template discussed in Chapter 6.

- The FPGA coder modules are limited by the use of the TUN/TAP device used by the FPGA coding integration. The TUN/TAP device was used to save time during the FPGA interface implementation, because TUN/-TAP devices are the generic method of interfacing userspace applications with networking layer functionality. The end-to-end results showed that the VNFs were able to achieve better loopback performance, even while in a VM, compared to the TUN/TAP integration. This was done using the DPDK packet processing library. To reduce the limitations of the Linux networking stack on the FPGA integration, the DPDK TUN/TAP PMD could be used instead. This would allow for the same userspace functionality to exist as with the current FPGA integration implementation, but with a performance benefits of DPDK.

- The DPDK VNF implementation is only developed as a single core application. The implementation only makes use of a single port on the core and therefore only a single queue. The potential coding performance could be increased by using more ports to receive incoming packet data from multiple streams simultaneously. The OvS bridge could be used to divide the incoming packet streams evenly amongst the ports, and grouped by generation to reduce any queueing delays on the coder itself. The use of more cores on the VNF could be used to perform more simultaneous coding operations on stored packets and be used to reduce the overall coding delay.

- The FPGA coding implementations only make use of a single PCIe lane provided by the OpenVINO starter kit, out of the potential four. The coding implementations could make use of all four PCIe lanes to provide four streams of incoming packet data. This could be used to load packets for four coding modules simultaneously, as discussed in Chapter 7, or to load packet segments for a single coding module at a faster rate.

8.2 Future work

This thesis demonstrates that FPGA-based network coding is feasible and provides a significant performance increase over software-based implementations.

The results in Chapter 7 showcase however, that the performance is reduced dramatically when integrated with a real packet-based network. This is due to the delays introduced from the coder and OvS integration implementations. Future work should therefore focus more on the integration aspect of software and hardware-based network coding functions with real packet-based networks. Alternative techniques for offloading packet data to network coding functions need to be developed.

Bibliography

[1] Hansen, J., Lucani, D.E., Krigslund, J., Médard, M. and Fitzek, F.H.: Network coded software defined networking: Enabling 5G transmission and storage networks. *IEEE Communications Magazine*, vol. 53, no. 9, pp. 100–107, 2015.

[2] Ahlswede, R., Ning Cai, Li, S.-Y. and Yeung, R.: Network information flow. *IEEE Transactions on Information Theory*, vol. 46, no. 4, pp. 1204–1216, 2000.

[3] Ho, T., Médard, M., Koetter, R., Karger, D.R., Effros, M., Shi, J. and Leong, B.: A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, vol. 52, no. 10, pp. 4413–4430, 2006.

[4] Fragouli, C. and Soljanin, E.: Network Coding Fundamentals. *Foundations and Trends® in Networking*, vol. 2, no. 1, pp. 1–133, 2007.

[5] Chou, P., Wu, Y. and Jain, K.: Practical Network Coding. In: *Allerton conference on communication control and computing.*, pp. 40–49. IEEE, 2003.

[6] Keller, L.: NCUtils. 2015.
Available at: `https://lokeller.github.io/ncutils/`

[7] Visegradi, A.: Github: rnc-lib. 2014.
Available at: `https://github.com/avisegradi/rnc-lib`

[8] Steinwurf ApS: Steinwurf tecnical documentation: Kodo. 2018.
Available at: `http://docs.steinwurf.com/kodo.html`

[9] Kreutz, D., Ramos, F.M.V., Esteves Verissimo, P., Esteve Rothenberg, C., Azodolmolky, S. and Uhlig, S.: Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, jan 2015.

[10] Open Networking Foundation: OpenFlow Switch Specification. 2015.
Available at: `https://www.opennetworking.org/`

[11] DPDK Project: Data Plane Development Kit Documentation. 2019.
Available at: `https://core.dpdk.org/doc/`

[12] Robin, G.: Open vSwitch* with DPDK Overview. 2016.
Available at: `https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview`

[13] Pfaff, B., Pettit, J., Koponen, T., Jackson, E.J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K. and Casado, M.: The design and implementation of open vSwitch. *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015*, pp. 117–130, 2015.

[14] Kim, S., Jeong, W.S., Ro, W.W. and Gaudiot, J.L.: Design and evaluation of random linear network coding accelerators on FPGAs. *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 1, pp. 13:1–13:24, 2013.

[15] Gabriel, F., Nguyen, G.T., Schmoll, R.S., Cabrera, J.A., Muehleisen, M. and Fitzek, F.H.: Practical deployment of network coding for real-time applications in 5G networks. In: *CCNC 2018 - 2018 15th IEEE Annual Consumer Communications and Networking Conference*, vol. 2018-Janua, pp. 1–2. 2018. ISBN 9781538647905.

[16] Dordal, P.: *An Introduction to Computer Networks*. 2018.

[17] Fenner, W.: Internet Group Management Protocol, Version 2. 1997. Available at: `https://tools.ietf.org/html/rfc2236`

[18] Li, S.Y.R., Yeung, R.W. and Cai, N.: Linear network coding. *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371–381, 2003.

[19] Koetter, R. and Médard, M.: An algebraic approach to network coding. *IEEE/ACM Transactions on Networking*, vol. 11, no. 5, pp. 782–795, 2003.

[20] Ho, T., Koetter, R., Medard, M., Karger, D. and Effros, M.: The benefits of coding over routing in a randomized setting. In: *IEEE International Symposium on Information Theory*. IEEE, 2003.

[21] Ho, T., Karger, D.R., Médard, M. and Koetter, R.: Network coding from a network flow perspective. In: *IEEE International Symposium on Information Theory*, p. 441. 2003.

[22] Kerl, J.: *Computation in finite fields*. 2004.

[23] Keller, L., Le, A., Cici, B., Seferoglu, H., Fragouli, C. and Markopoulou, A.: MicroCast: Cooperative video streaming on smartphones. In: *MobiSys'12 - Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pp. 57–69. 2012.

[24] Angelopoulos, G., Paidimarri, A., Chandrakasan, A.P. and Medard, M.: Experimental study of the interplay of channel and network coding in low power sensor applications. In: *IEEE International Conference on Communications*, pp. 5126–5130. 2013.

[25] Pedersen, M.V., Heide, J. and Fitzek, F.H.: Kodo: An open and research oriented network coding library. In: *IFIP International Federation for Information Processing*, pp. 145–152. 2011.

[26] Pahlevani, P., Lucani, D.E., Pedersen, M.V. and Fitzek, F.H.: PlayNCool: Opportunistic network coding for local optimization of routing in wireless mesh networks. In: *2013 IEEE Globecom Workshops, GC Wkshps 2013*, pp. 812–817. 2013.

[27] Németh, F., Stipkovits, Á., Sonkoly, B. and Gulyás, A.: Towards smartFlow: Case studies on enhanced programmable forwarding in openFlow switches. *Computer Communication Review*, vol. 42, no. 4, pp. 85–86, aug 2012.

[28] Yang, J., Dai, B., Lv, L. and Xu, G.: Coding Openflow: Enable Network Coding in SDN Networks. *International journal of Computer Networks & Communications*, vol. 7, no. 5, pp. 29–38, 2015.

[29] Zhu, D., Yang, X., Zhao, P. and Yu, W.: Towards effective intra-flow network coding in software defined wireless mesh networks. In: *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, vol. 2015-Octob. 2015.

[30] Krasnyansky, M. and Thiel, F.: Universal TUN/TAP device driver. 2002. Available at: `https://www.kernel.org/doc/Documentation/networking/tuntap.txt`

[31] Barbette, T., Soldani, C. and Mathy, L.: Fast userspace packet processing. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 5–16. 2015. ISBN VO -.

[32] Xia, W., Zhao, P., Wen, Y. and Xie, H.: A Survey on Data Center Networking (DCN): Infrastructure and Operations. *IEEE Communications Surveys and Tutorials*, vol. 19, no. 1, pp. 640–656, 2017. ISSN 1553877X.

[33] Mellanox Technologies: ConnectX®-6 EN Single/Dual-Port Adapter Supporting 200Gb/s Ethernet. 2019. Available at: `https://www.mellanox.com/page/products{_}dyn?product{_}family=266{&}mtag=connectx{_}6{_}en{_}card`

[34] Herrin, G.: Linux IP Networking: A Guide to the Implementation and Modification of the Linux Protocol Stack. 2000. Available at: `https://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html`

[35] Kourtis, M., Xilouris, G., Riccobene, V., McGrath, M.J., Petralia, G., Koumaras, H., Gardikis, G. and Liberal, F.: Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pp. 74–78. 2015. ISBN VO -.

[36] Intel Corporation: Open vSwitch* Enables SDN and NFV Transformation, 2015. Available at: `https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview`

[37] OpenStack: Open vSwitch with DPDK datapath. 2019.
Available at: `https://docs.openstack.org/newton/networking-guide/config-ovs-dpdk.html`

[38] Vazquez, N.: Openvswitch with DPDK support on CloudStack. 2018.
Available at: `https://www.shapeblue.com/openvswitch-with-dpdk-support-on-cloudstack/`

[39] The Linux Foundation: Huge Pages.
Available at: `https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt`

[40] Chu, X., Zhao, K. and Wang, M.: Massively parallel network coding on GPUs. In: *Conference Proceedings of the IEEE International Performance, Computing, and Communications Conference*, pp. 144–151. 2008. ISBN 9781424433674.

[41] Shojania, H., Li, B. and Wang, X.: Nuclei: GPU-Accelerated Many-Core Network Coding. In: *IEEE INFOCOM 2009*, pp. 459–467. 2009. ISBN VO -.

[42] Gibb, G., Lockwood, J.W., Naous, J., Hartke, P. and McKeown, N.: NetFPGA - An open platform for teaching how to build gigabit-rate network switches and routers. *IEEE Transactions on Education*, vol. 51, no. 3, pp. 364–369, aug 2008.

[43] NetFPGA.
Available at: `https://netfpga.org/site/{#}/`

[44] Naous, J., Erickson, D., Covington, G.A., Appenzeller, G. and McKeown, N.: Implementing an OpenFlow switch on the NetFPGA platform. *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS '08*, pp. 1–9, 2008.

[45] Viet, A.N., Van, L.P., Minh, H.A.N., Xuan, H.D., Ngoc, N.P. and Huu, T.N.: Mitigating HTTP GET flooding attacks in SDN using NetFPGA-based OpenFlow switch. In: *ECTI-CON 2017 - 2017 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, pp. 660–663. jun 2017.

[46] Wellem, T., Lai, Y.K., Cheng, C.H., Liao, Y.C., Chen, L.T. and Huang, C.Y.: Implementing a heavy hitter detection on the NetFPGA OpenFlow switch. In: *IEEE Workshop on Local and Metropolitan Area Networks*, vol. 2017-June, pp. 1–2. 2017.

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

[48] Liu, T.: *Implementing Open flow switch using FPGA based platform.* Masters, Norwegian University of Science and Technology, 2014.

[49] Kalyaev, A. and Melnik, E.: FPGA-based approach for organization of SDN switch. In: *9th International Conference on Application of Information and Communication Technologies, AICT 2015 - Proceedings*, pp. 363–366. 2015.

[50] Xilinx: Digilent NetFPGA-1G-CML Kintex-7 FPGA Development Board. Available at: `https://www.xilinx.com/products/boards-and-kits/1-4le3gu.html`

[51] Cao, J., Zheng, X., Sun, L. and Jin, J.: The Development Status and Trend of NetFPGA. In: *Proceedings - 2015 International Conference on Network and Information Systems for Computers, ICNISC 2015*, pp. 101–105. jan 2015.

[52] Terasic Inc.: OpenVINO Starter Kit User Manual, 2019.

[53] Intel: PCI Express High Performance Reference Design. 2018. Available at: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an456.pdf`

[54] The Kernel Development Community: The Linux Kernel documentation. 2016. Available at: `https://www.kernel.org/doc/html/v4.10/driver-api/80211/mac80211.html`

[55] The Linux Foundation: Open vSwitch Documentation. 2019. Available at: `http://docs.openvswitch.org/en/latest/`

[56] Weil, S.: QEMU User Documentation. 2019. Available at: `https://qemu.weilnetz.de/doc/qemu-doc.html`

[57] de Villiers, D.: GitHub: OvS-DPDK-Coding-Switch. 2019. Available at: `https://github.com/danieldevill/OvS-DPDK-Coding-Switch`

[58] de Villiers, D.: GitHub: RLNC_VNF. 2019. Available at: `https://github.com/danieldevill/RLNC{_}VNF`

[59] Steinwurf Aps.: GitHub - steinwurf/kodo-python: Kodo python bindings. . Available at: `https://github.com/steinwurf/kodo-python`

[60] Traynor, K.: Open vSwitch-DPDK: How Much Hugepage Memory? 2018. Available at: `https://developers.redhat.com/blog/2018/03/16/ovs-dpdk-hugepage-memory/`

[61] Eastlake, D. and Abley, J.: IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters. 2013. Available at: `https://www.rfc-editor.org/rfc/rfc7042.txt`

[62] Steinwurf Aps.: GitHub - steinwurf/kodo-c: Kodo C bindings. 2019. Available at: `https://github.com/steinwurf/kodo-c`

[63] Sundararajan, J.K., Shah, D., Médard, M., Mitzenmacher, M. and Barros, J.: Network coding meets TCP. In: *IEEE INFOCOM*, pp. 280–288. 2009.

[64] Intel: FIFO Intel® FPGA IP User Guide. 2018.
Available at: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug{_}fifo.pdf`

[65] Klein, A.: *Stream ciphers*, vol. 9781447150. Springer, 2013.

[66] Steinwurf Aps.: GitHub - steinwurf/fifi-python: Fifi python bindings. .
Available at: `https://github.com/steinwurf/fifi-python`

[67] QEMU: GitHub - qemu/qemu: Official QEMU mirror. 2019.
Available at: `https://github.com/qemu/qemu`

[68] Ryu SDN Framework Community: Ryu SDN Framework. 2017.
Available at: `https://osrg.github.io/ryu/`

[69] Christensen, M., Kimball, K., Solensky, F., Thrane & Thrane, Hewlett-Packard and Calix: Considerations for Internet Group Management Protocol (IGMP) and Multicast Listener Discovery (MLD) Snooping Switches. 2006.
Available at: `https://tools.ietf.org/html/rfc4541`

[70] Intel Corporation: Avalon® Interface Specifications. 2019.
Available at: `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl{_}avalon{_}spec.pdf`

[71] Intel Corporation: Avalon-MM Master Templates Readme, 2008.
Available at: `https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/intellectual-property/embedded/nios-ii/exm-avalon-mm.html`

[72] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F. and Carle, G.: MoonGen: A Scriptable High-Speed Packet Generator. In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, oct 2015.

[73] Altera: Cyclone V Device Handbook Volume 1: Device Interfaces and Integration, 2019.

Appendices

System setup

Network configuration scripts outputs

A collection of bash scripts are used to setup the network configuration. These are provided in the case of reproducibility. Anyone setting up the GitHub project can use the screenshots to validate their configuration.

```
[switch@ovsswitch netconfig]$ ./setup_ovsdpdk.sh
[sudo] password for switch:
ovsdb-server: no process found
ovs-vswitchd: no process found
Starting OVS with dpdk config enabled..
Inserting openvswitch module                               [  OK  ]
Starting ovs-vswitchd                                      [  OK  ]
Enabling remote OVSDB managers                             [  OK  ]
2019-11-03T14:56:31Z|00001|vlog|WARN|failed to open /var/log/openvswitch/ovs-vswitchd.log for logging: No such fil
Adding bridge br0..

Network devices using kernel driver
===================================
0000:07:00.0 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' if=enp7s0f0 drv=e1000e unused=vfio-pci
0000:07:00.1 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' if=enp7s0f1 drv=e1000e unused=vfio-pci
0000:08:00.0 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' if=enp8s0f0 drv=e1000e unused=vfio-pci
0000:08:00.1 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' if=enp8s0f1 drv=e1000e unused=vfio-pci
0000:0a:00.0 'I350 Gigabit Network Connection 1521' if=enp10s0f0 drv=igb unused=vfio-pci *Active*
0000:0a:00.1 'I350 Gigabit Network Connection 1521' if=enp10s0f1 drv=igb unused=vfio-pci

No 'Crypto' devices detected
============================

No 'Eventdev' devices detected
==============================

No 'Mempool' devices detected
=============================

No 'Compress' devices detected
==============================
d5bbda64-46f0-4ea2-996d-4a0db6cfa462
    Bridge "br0"
        Controller "tcp:10.10.11.117:6633"
        Port "br0"
            Interface "br0"
                type: internal
ovs-vswitchd (Open vSwitch) 2.11.0
DPDK 18.11.2
Is DPDK initialized:
true
AnonHugePages:    331776 kB
HugePages_Total:   32768
HugePages_Free:    32512
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

Figure 1: Screenshot of output from OvS-DPDK networking setup script *setup_ ovsdpdk.sh*.

```
[switch@ovsswitch netconfig]$ ./setup_nichosts.sh

Network devices using DPDK-compatible driver
============================================
0000:07:00.0 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' drv=vfio-pci unused=e1000e
0000:07:00.1 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' drv=vfio-pci unused=e1000e
0000:08:00.0 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' drv=vfio-pci unused=e1000e
0000:08:00.1 '82571EB/82571GB Gigabit Ethernet Controller (Copper) 10bc' drv=vfio-pci unused=e1000e

Network devices using kernel driver
===================================
0000:0a:00.0 'I350 Gigabit Network Connection 1521' if=enp10s0f0 drv=igb unused=vfio-pci *Active*
0000:0a:00.1 'I350 Gigabit Network Connection 1521' if=enp10s0f1 drv=igb unused=vfio-pci

No 'Crypto' devices detected
============================

No 'Eventdev' devices detected
==============================

No 'Mempool' devices detected
=============================

No 'Compress' devices detected
==============================
d5bbda64-46f0-4ea2-996d-4a0db6cfa462
    Bridge "br0"
        Controller "tcp:10.10.11.117:6633"
        Port "enp7s0f1"
            Interface "enp7s0f1"
                type: dpdk
                options: {dpdk-devargs="0000:07:00.1"}
        Port "enp7s0f0"
            Interface "enp7s0f0"
                type: dpdk
                options: {dpdk-devargs="0000:07:00.0"}
        Port "enp8s0f0"
            Interface "enp8s0f0"
                type: dpdk
                options: {dpdk-devargs="0000:08:00.0"}
        Port "enp8s0f1"
            Interface "enp8s0f1"
                type: dpdk
                options: {dpdk-devargs="0000:08:00.1"}
        Port "br0"
            Interface "br0"
                type: internal
```

Figure 2: Screenshot of output from physical hosts networking setup script *setup_nichosts.sh*.

```
[switch@ovsswitch netconfig]$ ./setup_vmhosts.sh 4
qemu-system-x86_64: -chardev socket,id=char0,path=/tmp/dpdkvhostclient0,server: info: QEMU waiting for connection on:
qemu-system-x86_64: -chardev socket,id=char1,path=/tmp/dpdkvhostclient1,server: info: QEMU waiting for connection on:
qemu-system-x86_64: -chardev socket,id=char2,path=/tmp/dpdkvhostclient2,server: info: QEMU waiting for connection on:
[switch@ovsswitch netconfig]$ qemu-system-x86_64: -chardev socket,id=char3,path=/tmp/dpdkvhostclient3,server: info: QE
rver

[switch@ovsswitch netconfig]$ sudo ovs-vsctl show
d5bbda64-46f0-4ea2-996d-4a0db6cfa462
    Bridge "br0"
        Controller "tcp:10.10.11.117:6633"
        Port "vhp3"
            Interface "vhp3"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient3"}
        Port "vhp1"
            Interface "vhp1"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient1"}
        Port "vhp0"
            Interface "vhp0"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient0"}
        Port "enp7s0f1"
            Interface "enp7s0f1"
                type: dpdk
                options: {dpdk-devargs="0000:07:00.1"}
        Port "enp7s0f0"
            Interface "enp7s0f0"
                type: dpdk
                options: {dpdk-devargs="0000:07:00.0"}
        Port "enp8s0f0"
            Interface "enp8s0f0"
                type: dpdk
                options: {dpdk-devargs="0000:08:00.0"}
        Port "enp8s0f1"
            Interface "enp8s0f1"
                type: dpdk
                options: {dpdk-devargs="0000:08:00.1"}
        Port "br0"
            Interface "br0"
                type: internal
        Port "vhp2"
            Interface "vhp2"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient2"}
```

Figure 3: Screenshot of output from virtual machine hosts networking setup script *setup_ vmhosts.sh*.

```
[switch@ovsswitch netconfig]$ ./setup_vmvnfs.sh
rm: cannot remove '/tmp/dpdkvhostclient9': No such file or directory
rm: cannot remove '/tmp/dpdkvhostclient10': No such file or directory
ovs-vsctl: no port named vhp_encodeVNF
ovs-vsctl: no port named vhp_decodeVNF
[switch@ovsswitch netconfig]$ qemu-system-x86_64: -chardev socket,id=char10,path=/tmp/dpdkvhostclient10,server: info:
,server
qemu-system-x86_64: -chardev socket,id=char9,path=/tmp/dpdkvhostclient9,server: info: QEMU waiting for connection on:

[switch@ovsswitch netconfig]$ sudo ovs-vsctl show
d5bbda64-46f0-4ea2-996d-4a0db6cfa462
    Bridge "br0"
        Controller "tcp:10.10.11.117:6633"
        Port vhp_encodeVNF
            Interface vhp_encodeVNF
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient9"}
        Port "vhp3"
            Interface "vhp3"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient3"}
        Port "vhp1"
            Interface "vhp1"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient1"}
        Port "vhp0"
            Interface "vhp0"
                type: dpdkvhostuserclient
                options: {vhost-server-path="/tmp/dpdkvhostclient0"}
        Port "enp7s0f1"
            Interface "enp7s0f1"
                type: dpdk
                options: {dpdk-devargs="0000:07:00.1"}
        Port "enp7s0f0"
            Interface "enp7s0f0"
                type: dpdk
                options: {dpdk-devargs="0000:07:00.0"}
        Port "enp8s0f0"
            Interface "enp8s0f0"
                type: dpdk
                options: {dpdk-devargs="0000:08:00.0"}
        Port "enp8s0f1"
            Interface "enp8s0f1"
                type: dpdk
                options: {dpdk-devargs="0000:08:00.1"}
        Port "br0"
```

Figure 4: Screenshot of output from virtual machine VNFs networking setup script *setup_ vmvnfs.sh*.

Physical network setup

The physical network is created using six computers. The physical setup is shown in Fig. 5. The main computer is a dual Intel Xeon E5-2670 CPU server with 120GB of DDR3 memory. The switching component, hypervisor, VMs and all interfacing is done on this server.

The remaining five computers are Dell OptiPlex desktop computers with Intel Core-i5 2400 CPUs and 4GB of DDR3 memory. Linux Mint is installed as the OS. These computer are used for the four physical host machines, as well as the SDN controller.

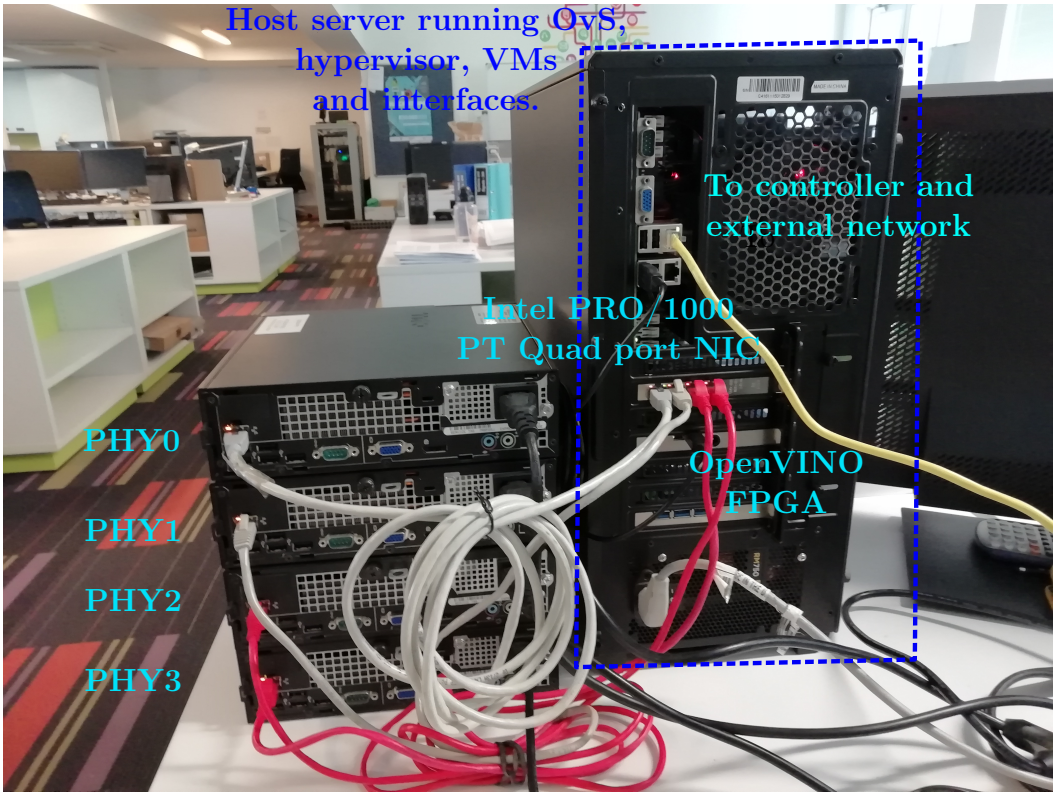

Figure 5: Physical layer lab setup.

Kodo Python baseline

A Python implementation of the Kodo RLNC algorithm is created to showcase the correct encoding and decoding results. The software code used to implement the Kodo Python baseline is presented in 8.2.

Kodo Python code

```
#! /usr/bin/env python
import os
import random
import sys
import numpy
import kodo
import re
import fifi

def main():

 #Paramaters
 field = kodo.field.binary8
 packets = 7
 packet_size = 64

 #Create an encoder/decoder factory that are used to build the
 #actual encoders/decoders
 encoder_factory = kodo.RLNCEncoderFactory(field, packets,
    packet_size)
 encoder = encoder_factory.build()
 decoder_factory = kodo.RLNCDecoderFactory(field, packets,
    packet_size)
 decoder = decoder_factory.build()

 #Block Data
 data_in = bytearray([
```

```
  251,153,198,77,15,71,255,10,138,213,33,37,212,104,200,106,
  228,160,133,83,152,161,233,93,127,176,172,34,98,151,95,229,
  191,26,68,16,100,221,65,234,5,62,36,99,31,235,153,123,
  24,55,25,116,255,236,0,24,137,77,82,205,199,113,153,131,
  109,23,110,212,112,187,112,55,74,70,161,87,11,180,134,39,
  178,19,148,9,34,249,144,187,41,88,108,219,185,139,234,114,
  78,25,123,17,191,44,80,48,95,255,113,90,173,121,39,34,
  245,150,6,161,64,28,66,210,212,18,238,93,132,218,142,114,
  73,62,213,141,0,62,100,11,149,180,210,112,144,206,176,117,
  55,7,156,13,172,9,84,104,47,58,43,169,119,215,171,206,
  255,171,125,77,40,19,18,133,247,141,191,245,160,94,66,217,
  133,185,149,77,146,102,175,222,159,69,216,48,108,34,64,253,
  36,170,244,95,177,230,28,49,180,142,108,136,90,47,7,164,
  115,83,237,119,83,181,180,134,248,163,205,207,208,146,152,126,

  223,57,185,71,115,161,161,31,40,44,239,72,107,71,126,7,
  25,165,77,92,29,85,128,201,100,90,169,146,195,56,205,95,
  206,49,60,45,244,59,224,109,219,99,215,211,250,145,135,232,
  250,226,232,4,126,107,42,235,114,202,219,197,196,113,73,178,
  22,24,231,159,200,75,180,171,53,104,160,91,104,211,174,19,
  110,250,42,89,146,250,66,64,95,61,202,202,224,242,141,77,
  242,33,241,248,65,172,159,115,13,121,202,232,165,56,199,139,
  247,246,233,66,195,119,67,9,62,250,190,236,27,63,238,177,
  3,101,19,88,14,5,199,102,248,151,170,108,133,4,255,85,
  203,65,53,104,223,159,131,230,200,167,121,195,209,197,206,217,

  1,124,108,61,179,145,208,237,186,96,195,102,217,202,155,32,
  15,62,94,239,16,174,82,27,249,212,75,49,29,139,6,213,
  50,225,206,2,96,18,49,34,12,53,218,117,166,49,114,131,
  82,221,45,236,68,102,143,199,106,136,125,127,31,195,193,30])
print("Data in symbols:")
packet_number = 0
for d in data_in:
 print('{:02X} '.format(d),end = '')
 packet_number+=1
 if packet_number == packet_size:
  print()
  packet_number = 0

#Coefficient Data
coef0 = bytearray([157,181, 214, 252, 153, 49, 207])
coef1 = bytearray([19, 109, 185, 21, 255, 84, 105])
coef2 = bytearray([20, 170, 19, 69, 7, 83, 131])
coef3 = bytearray([213, 47, 207, 139, 226, 14, 107])
```

```
coef4 = bytearray([244, 95, 247, 36, 116, 154, 33])
coef5 = bytearray([92, 183, 235, 115, 11, 71, 241])
coef6 = bytearray([201, 224, 155, 217, 41, 113, 151])
coefs = [coef0,coef1,coef2,coef3,coef4,coef5,coef6]
print("\nCoding coefficients:")
for i in range(0,7):
 coefficients = bytearray(coefs[i])
 print("".join('\\x{:02x}'.format(c) for c in coefficients))

#Set encoder input array
encoder.set_const_symbols(data_in)

#Set decoder output array
data_out = bytearray(decoder.block_size())
decoder.set_mutable_symbols(data_out)

packet_number = 0

#Coding loop
print("\nCoding symbols:")
for i in range(0,7):
 #Get coding coefficients
 coefficients = bytearray(coefs[i])

 #Write a coded symbol to the symbol buffer
 symbol = encoder.write_symbol(coefficients)

 #Print encoded symbols
 for s in symbol:
  print('\\x{:02x}'.format(s),end = '')
  packet_number+=1
  if packet_number == 64:
   print()
   packet_number = 0

 # Pass that symbol and the corresponding coefficients to the
    decoder
 decoder.read_symbol(symbol, coefficients)

print("Coding finished")

#Check decoded results with original data
print("Checking results...")
if data_out == data_in:
```

```
  print("Data decoded correctly")

if __name__ == "__main__":
 main()
```

Raw RLNC data values

The raw source data, coefficient data and encoded data used in the Kodo Python implementation is provided. The same data is used to verify the encoder and decoder results of the VNF and FPGA implementations. The values are presented in hexadecimal form, for each of the $h = 7$ packets of $N = 64$ bytes in size. The data is grouped together in blocks.

Source, un-coded data

b0	FB	99	C6	4D	0F	47	FF	0A	8A	D5	21	25	D4	68
b14	C8	6A	E4	A0	85	53	98	A1	E9	5D	7F	B0	AC	22
b28	62	97	5F	E5	BF	1A	44	10	64	DD	41	EA	5	3E
b42	24	63	1F	EB	99	7B	18	37	19	74	FF	EC	0	18
b56	89	4D	52	CD	C7	71	99	83	6D	17	6E	D4	70	BB
b70	70	37	4A	46	A1	57	0B	B4	86	27	B2	13	94	9
b84	22	F9	90	BB	29	58	6C	DB	B9	8B	EA	72	4E	19
b98	7B	11	BF	2C	50	30	5F	FF	71	5A	AD	79	27	22
b112	F5	96	6	A1	40	1C	42	D2	D4	12	EE	5D	84	DA
b126	8E	72	49	3E	D5	8D	0	3E	64	0B	95	B4	D2	70
b140	90	CE	B0	75	37	7	9C	0D	AC	9	54	68	2F	3A
b154	2B	A9	77	D7	AB	CE	FF	AB	7D	4D	28	13	12	85
b168	F7	8D	BF	F5	A0	5E	42	D9	85	B9	95	4D	92	66
b182	AF	DE	9F	45	D8	30	6C	22	40	FD	24	AA	F4	5F
b196	B1	E6	1C	31	B4	8E	6C	88	5A	2F	7	A4	73	53
b210	ED	77	53	B5	B4	86	F8	A3	CD	CF	D0	92	98	7E
b224	DF	39	B9	47	73	A1	A1	1F	28	2C	EF	48	6B	47
b238	7E	7	19	A5	4D	5C	1D	55	80	C9	64	5A	A9	92
b252	C3	38	CD	5F	CE	31	3C	2D	F4	3B	E0	6D	DB	63
b266	D7	D3	FA	91	87	E8	FA	E2	E8	4	7E	6B	2A	EB
b180	72	CA	DB	C5	C4	71	49	B2	16	18	E7	9F	C8	4B
b194	B4	AB	35	68	A0	5B	68	D3	AE	13	6E	FA	2A	59
b308	92	FA	42	40	5F	3D	CA	CA	E0	F2	8D	4D	F2	21
b322	F1	F8	41	AC	9F	73	0D	79	CA	E8	A5	38	C7	8B
b336	F7	F6	E9	42	C3	77	43	9	3E	FA	BE	EC	1B	3F
b350	EE	B1	3	65	13	58	0E	5	C7	66	F8	97	AA	6C
b364	85	4	FF	55	CB	41	35	68	DF	9F	83	E6	C8	A7
b378	79	C3	D1	C5	CE	D9	1	7C	6C	3D	B3	91	D0	ED
b392	BA	60	C3	66	D9	CA	9B	20	0F	3E	5E	EF	10	AE
b406	52	1B	F9	D4	4B	31	1D	8B	6	D5	32	E1	CE	2
b420	60	12	31	22	0C	35	DA	75	A6	31	72	83	52	DD
b438	2D	EC	44	66	8F	C7	6A	88	7D	7F	1F	C3	C1	1E

Coding coefficient data

b0	9D	B5	D6	FC	99	31	CF
b7	13	6D	B9	15	FF	54	69
b14	14	AA	13	45	7	53	83
b21	D5	2F	CF	8B	E2	0E	6B
b28	F4	5F	F7	24	74	9A	21
b35	5C	B7	EB	73	0B	47	F1
b42	C9	E0	9B	D9	29	71	97

Encoded data

b0	7B	AE	15	C9	39	4C	1A	FD	7	4D	25	87	54	BF
b14	DB	CB	CD	56	6C	29	E3	FB	F5	76	9	7C	C6	98
b28	EA	D2	BC	90	74	C9	50	7	75	2D	1E	98	8C	5
b42	87	EE	F9	E4	41	B0	21	4D	8	44	B7	5D	DA	76
b56	97	1C	49	B5	E3	F9	EB	BE	CF	3A	CC	E8	53	D3
b70	B7	5E	67	AF	1B	15	C2	65	B7	E8	8E	67	F5	27
b84	CF	DE	EE	79	A1	6C	50	15	C9	B5	64	B1	A1	68
b98	B1	BB	6F	2B	A4	92	95	A1	6D	2A	8E	DA	FE	DC
b112	AD	63	4D	AC	9C	78	B7	89	E2	E2	D9	DE	8B	6C
b126	73	36	9B	4E	5C	70	24	DA	68	2F	B7	F3	9F	9B
b140	78	48	8C	84	DE	CC	42	3	C3	D5	23	95	F3	D7
b154	C2	D8	13	55	24	C6	3D	EB	9	C3	E6	48	13	39
b168	2D	A8	B4	70	12	D4	73	53	27	1B	B0	3E	3D	E2
b182	F7	34	B8	C4	EA	DF	AB	C4	39	B1	20	FC	7	62
b196	D4	14	60	4B	93	CE	0D	5	7F	2D	4B	BC	67	74
b210	D2	81	A9	63	EA	52	B1	A9	98	A2	FD	25	14	7C
b224	A6	D5	73	64	EE	C1	8B	3D	F3	B5	2C	6F	89	8B
b238	0F	65	FD	71	F7	CA	49	12	3F	E5	63	2D	3A	62
b252	A9	8C	DB	35	6	8D	44	C8	1F	3F	AE	25	43	4
b266	1C	9A	CD	65	98	F9	F9	EC	15	22	E5	60	28	26
b180	5C	F0	E9	B1	4F	79	91	46	94	33	48	7F	98	D8
b194	E2	91	70	4F	4D	21	34	50	57	FB	DF	C3	14	E5
b308	52	10	A4	A7	93	6B	67	F9	AB	7F	30	13	DF	85
b322	E8	BC	DA	A9	6E	69	FD	DC	DD	82	BB	E1	C0	4E
b336	C1	35	3A	C6	3E	AC	FE	E3	DB	FB	FB	3D	D4	74
b350	A6	F0	0E	67	51	9E	9F	B8	0D	F4	57	23	2E	2
b364	B6	CC	E9	CC	0A	E7	0B	28	1B	14	B4	38	C1	92
b378	84	C9	0E	84	4D	84	37	29	2F	F4	92	F8	C2	0E
b392	BE	B4	99	88	F5	21	B3	36	ED	50	43	D6	50	A1
b406	CE	84	F5	31	26	94	8F	DC	5A	C5	22	4A	4D	84
b420	38	11	E2	9D	0C	9	F5	85	47	A4	18	67	3F	EB
b438	72	84	C8	89	6	3A	85	7A	10	CB	40	7D	A9	5

Decoded data

Same as source data.

RLNC VNF GitHub project

The code for the DPDK based coding functions can be found in the GitHub project *RLNC_ VNF* at [58]. The GitHub project folder is structured as shown in Fig. 6.

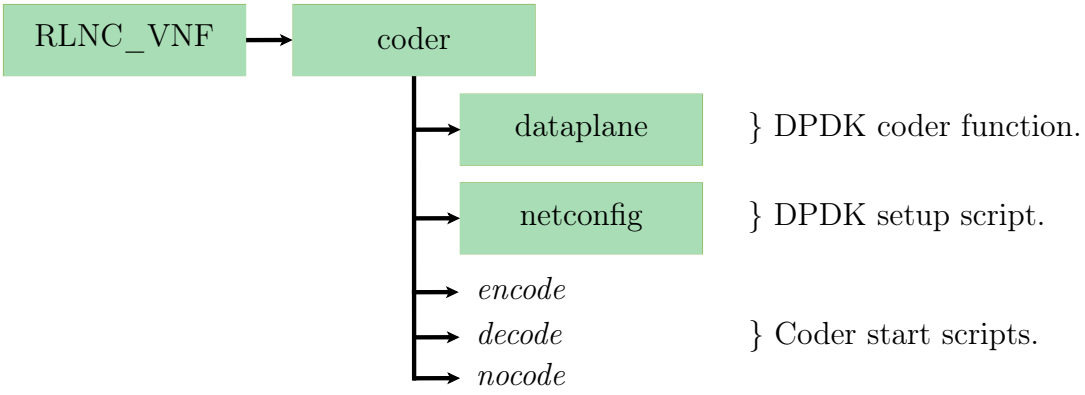

Figure 6: File structure of the *RLNC_ VNF* GitHub project.

The *dataplane* folder contains the DPDK coder function C program that implements the encoder and decoder functions. The script used to bind the VM networking interface to DPDK is placed under *netconfig*. Three start scripts *encode*, *decode*, *nocode* are created to start the DPDK coder in encoding, decoding or loopback mode respectively.

VNF DPDK networking setup script

The DPDK setup script, "setup_all.sh" code is given below. A screenshot of the output produced from the script is shown in Fig. 7. The output shows the VM network device bound to the DPDK compatible driver and the amount of Hugepages used before the coder is started.

```
#!/bin/bash

#Start DPDK setup.

#Export $RTE_SDK
export RTE_SDK=/usr/src/dpdk-stable-18.11.2

#Bind device to DPDK
echo Binding devices to DPDK..
sudo ip link set ens4 down

#Enable PMD
sudo modprobe uio_pci_generic
#sudo modprobe igb_uio
#sudo insmod /usr/src/dpdk-stable-18.11.1/build/kmod/igb_uio.ko
#sudo modprobe igb_uio

#Bind NICs to DPDK
sudo $RTE_SDK/usertools/dpdk-devbind.py --bind=uio_pci_generic
   ens4

#Print outputs of dpdk drivers and ovs-vsctl/ovs-ofctl to
   confirm.
sudo $RTE_SDK/usertools/dpdk-devbind.py -s

#Print number of Hugepages
grep Huge /proc/meminfo
```

```
coder@networkcoder:~/RLNC_VNF/coder$ netconfig/setup_all.sh
Binding devices to DPDK..
[sudo] password for coder:

Network devices using DPDK-compatible driver
============================================
0000:00:04.0 'Virtio network device 1000' drv=uio_pci_generic unused=

Network devices using kernel driver
===================================
0000:00:03.0 '82540EM Gigabit Ethernet Controller 100e' if=ens3 drv=e1000 unused=uio_pci_generic *Active*

No 'Crypto' devices detected
============================

No 'Eventdev' devices detected
==============================

No 'Mempool' devices detected
=============================

No 'Compress' devices detected
==============================
AnonHugePages:         0 kB
ShmemHugePages:        0 kB
HugePages_Total:    2048
HugePages_Free:     2048
HugePages_Rsvd:        0
HugePages_Surp:        0
Hugepagesize:       2048 kB
```

Figure 7: Screenshot of output from VNF DPDK networking setup script.

Results

The Signal Tap logic analyzer diagram for the FPGA encoder and decoder results are given in Figs. 8 and 9 respectively.

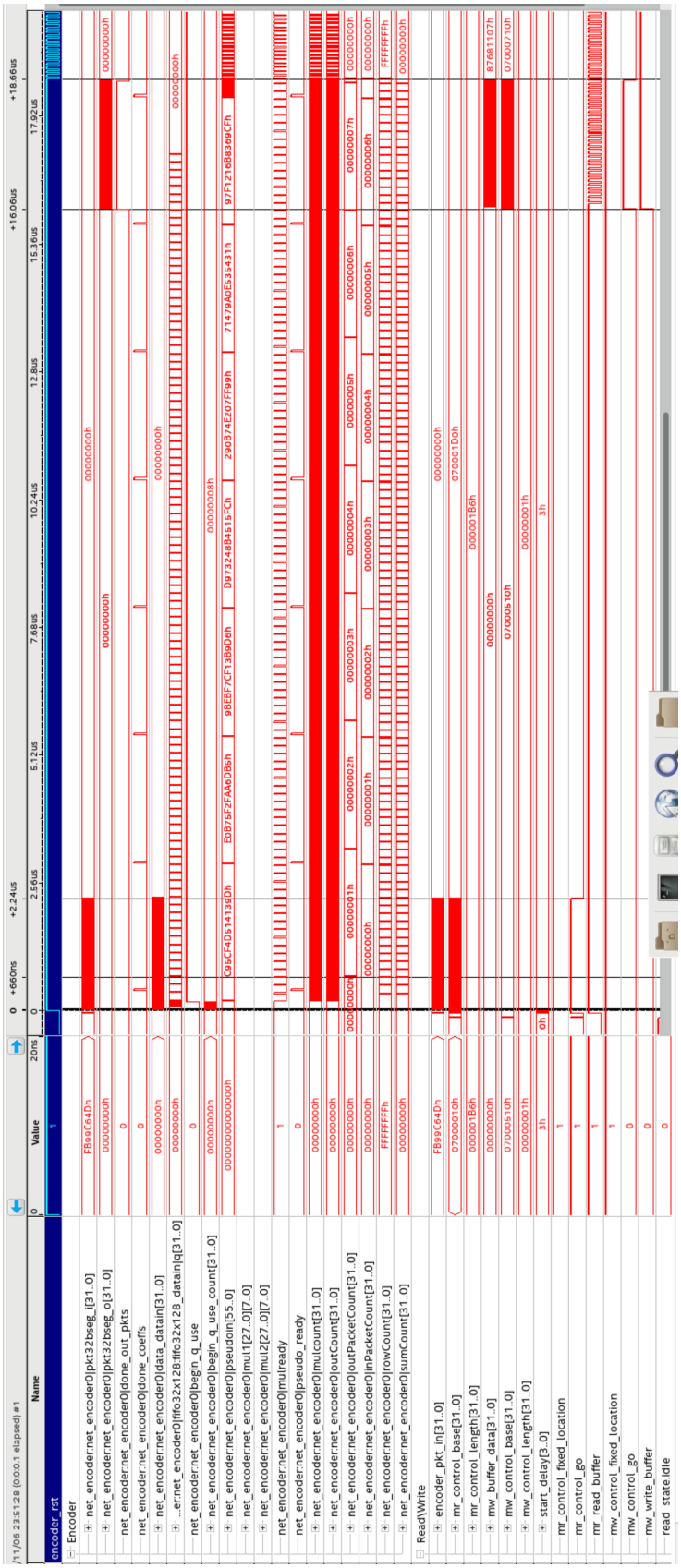

Figure 8: FPGA encoder signal tap results showing runtime of the encoding process.

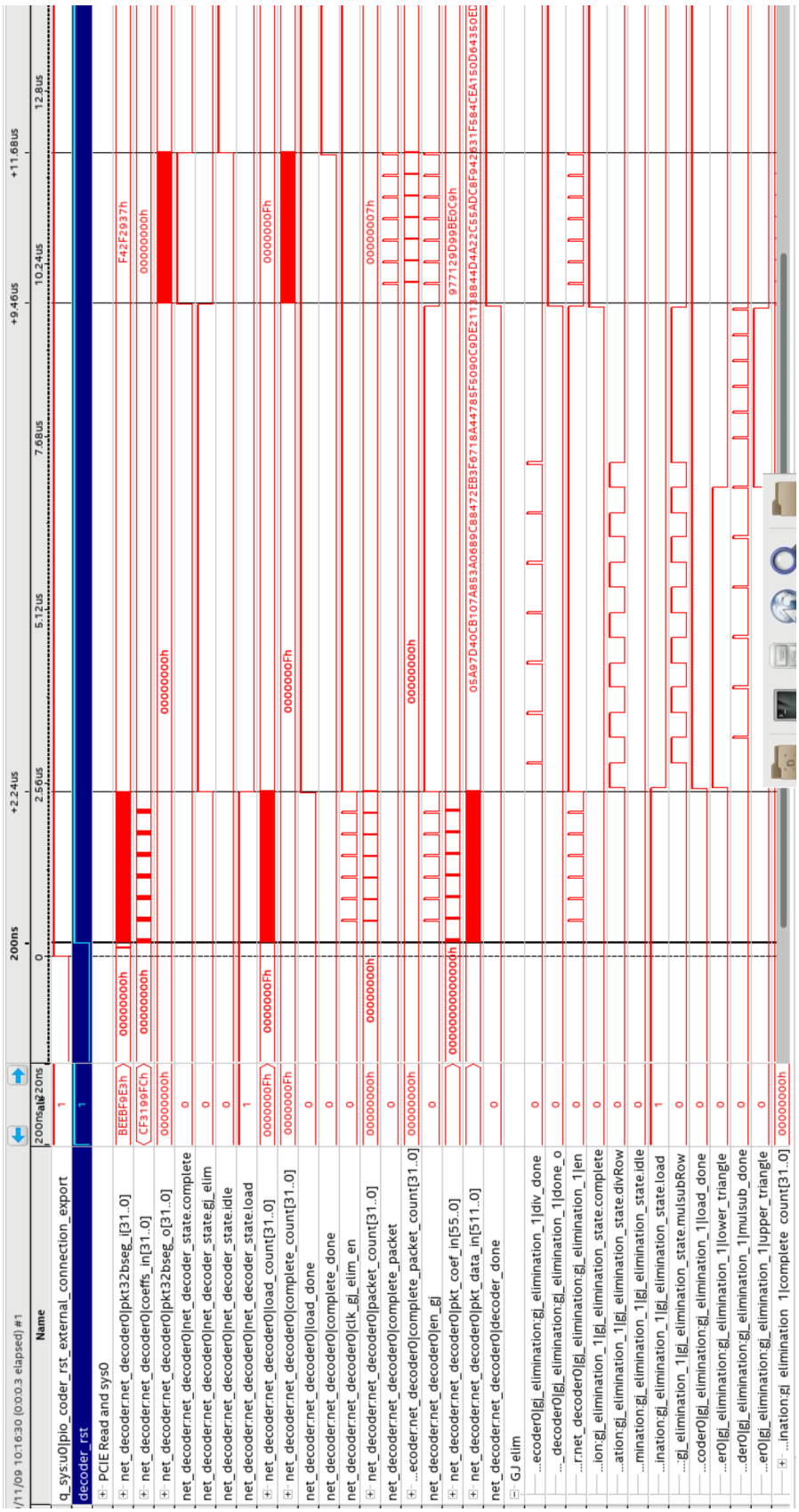

Figure 9: FPGA decoder signal tap results showing runtime of the decoding process.

Scapy test scripts

Scapy packet generator

The Scapy packet generator program used for the encoder and decoder are given below respectively:

```
import scapy.all as scapy
import sys

eth_src = "42:43:5d:3f:df:44"
eth_dst = "de:ad:be:ef:00:09"
eth_type = 0x8000

p0 = "\xfb\x99\xc6\x4d\x0f\x47\xff\x0a\x8a\xd5\x21\x25\xd4\x68\
   xc8\x6a\xe4\xa0\x85\x53\x98\xa1\xe9\x5d\x7f\xb0\xac\x22\x62\
   x97\x5f\xe5\xbf\x1a\x44\x10\x64\xdd\x41\xea\x05\x3e\x24\x63\
   x1f\xeb\x99\x7b\x18\x37\x19\x74\xff\xec\x00\x18\x89\x4d\x52\
   xcd\xc7\x71\x99\x83"
p1 = "\x6d\x17\x6e\xd4\x70\xbb\x70\x37\x4a\x46\xa1\x57\x0b\xb4\
   x86\x27\xb2\x13\x94\x09\x22\xf9\x90\xbb\x29\x58\x6c\xdb\xb9\
   x8b\xea\x72\x4e\x19\x7b\x11\xbf\x2c\x50\x30\x5f\xff\x71\x5a\
   xad\x79\x27\x22\xf5\x96\x06\xa1\x40\x1c\x42\xd2\xd4\x12\xee\
   x5d\x84\xda\x8e\x72"
p2 = "\x49\x3e\xd5\x8d\x00\x3e\x64\x0b\x95\xb4\xd2\x70\x90\xce\
   xb0\x75\x37\x07\x9c\x0d\xac\x09\x54\x68\x2f\x3a\x2b\xa9\x77\
   xd7\xab\xce\xff\xab\x7d\x4d\x28\x13\x12\x85\xf7\x8d\xbf\xf5\
   xa0\x5e\x42\xd9\x85\xb9\x95\x4d\x92\x66\xaf\xde\x9f\x45\xd8\
   x30\x6c\x22\x40\xfd"
p3 = "\x24\xaa\xf4\x5f\xb1\xe6\x1c\x31\xb4\x8e\x6c\x88\x5a\x2f\
   x07\xa4\x73\x53\xed\x77\x53\xb5\xb4\x86\xf8\xa3\xcd\xcf\xd0\
   x92\x98\x7e\xdf\x39\xb9\x47\x73\xa1\xa1\x1f\x28\x2c\xef\x48\
   x6b\x47\x7e\x07\x19\xa5\x4d\x5c\x1d\x55\x80\xc9\x64\x5a\xa9\
   x92\xc3\x38\xcd\x5f"
p4 = "\xce\x31\x3c\x2d\xf4\x3b\xe0\x6d\xdb\x63\xd7\xd3\xfa\x91\
   x87\xe8\xfa\xe2\xe8\x04\x7e\x6b\x2a\xeb\x72\xca\xdb\xc5\xc4\
   x71\x49\xb2\x16\x18\xe7\x9f\xc8\x4b\xb4\xab\x35\x68\xa0\x5b\
```

```
    x68\xd3\xae\x13\x6e\xfa\x2a\x59\x92\xfa\x42\x40\x5f\x3d\xca\
    xca\xe0\xf2\x8d\x4d"
p5 = "\xf2\x21\xf1\xf8\x41\xac\x9f\x73\x0d\x79\xca\xe8\xa5\x38\
    xc7\x8b\xf7\xf6\xe9\x42\xc3\x77\x43\x09\x3e\xfa\xbe\xec\x1b\
    x3f\xee\xb1\x03\x65\x13\x58\x0e\x05\xc7\x66\xf8\x97\xaa\x6c\
    x85\x04\xff\x55\xcb\x41\x35\x68\xdf\x9f\x83\xe6\xc8\xa7\x79\
    xc3\xd1\xc5\xce\xd9"
p6 = "\x01\x7c\x6c\x3d\xb3\x91\xd0\xed\xba\x60\xc3\x66\xd9\xca\
    x9b\x20\x0f\x3e\x5e\xef\x10\xae\x52\x1b\xf9\xd4\x4b\x31\x1d\
    x8b\x06\xd5\x32\xe1\xce\x02\x60\x12\x31\x22\x0c\x35\xda\x75\
    xa6\x31\x72\x83\x52\xdd\x2d\xec\x44\x66\x8f\xc7\x6a\x88\x7d\
    x7f\x1f\xc3\xc1\x1e"

l2packet0 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p0)
l2packet1 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p1)
l2packet2 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p2)
l2packet3 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p3)
l2packet4 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p4)
l2packet5 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p5)
l2packet6 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
    scapy.Raw(load=p6)

s = scapy.conf.L2socket(iface="br0")

for i in range(0,20000):
 s.send(l2packet0)
 s.send(l2packet1)
 s.send(l2packet2)
 s.send(l2packet3)
 s.send(l2packet4)
 s.send(l2packet5)
 s.send(l2packet6)
```

```
import scapy.all as scapy
import sys

eth_src = "42:43:5d:3f:df:44"
eth_dst = "02:01:02:03:04:08"
```

```
eth_type = 0x2020

#Encoded packet data
e0 = "\x7b\xae\x15\xc9\x39\x4c\x1a\xfd\x07\x4d\x25\x87\x54\xbf\
   xdb\xcb\xcd\x56\x6c\x29\xe3\xfb\xf5\x76\x09\x7c\xc6\x98\xea\
   xd2\xbc\x90\x74\xc9\x50\x07\x75\x2d\x1e\x98\x8c\x05\x87\xee\
   xf9\xe4\x41\xb0\x21\x4d\x08\x44\xb7\x5d\xda\x76\x97\x1c\x49\
   xb5\xe3\xf9\xeb\xbe"
e1 = "\xcf\x3a\xcc\xe8\x53\xd3\xb7\x5e\x67\xaf\x1b\x15\xc2\x65\
   xb7\xe8\x8e\x67\xf5\x27\xcf\xde\xee\x79\xa1\x6c\x50\x15\xc9\
   xb5\x64\xb1\xa1\x68\xb1\xbb\x6f\x2b\xa4\x92\x95\xa1\x6d\x2a\
   x8e\xda\xfe\xdc\xad\x63\x4d\xac\x9c\x78\xb7\x89\xe2\xe2\xd9\
   xde\x8b\x6c\x73\x36"
e2 = "\x9b\x4e\x5c\x70\x24\xda\x68\x2f\xb7\xf3\x9f\x9b\x78\x48\
   x8c\x84\xde\xcc\x42\x03\xc3\xd5\x23\x95\xf3\xd7\xc2\xd8\x13\
   x55\x24\xc6\x3d\xeb\x09\xc3\xe6\x48\x13\x39\x2d\xa8\xb4\x70\
   x12\xd4\x73\x53\x27\x1b\xb0\x3e\x3d\xe2\xf7\x34\xb8\xc4\xea\
   xdf\xab\xc4\x39\xb1"
e3 = "\x20\xfc\x07\x62\xd4\x14\x60\x4b\x93\xce\x0d\x05\x7f\x2d\
   x4b\xbc\x67\x74\xd2\x81\xa9\x63\xea\x52\xb1\xa9\x98\xa2\xfd\
   x25\x14\x7c\xa6\xd5\x73\x64\xee\xc1\x8b\x3d\xf3\xb5\x2c\x6f\
   x89\x8b\x0f\x65\xfd\x71\xf7\xca\x49\x12\x3f\xe5\x63\x2d\x3a\
   x62\xa9\x8c\xdb\x35"
e4 = "\x06\x8d\x44\xc8\x1f\x3f\xae\x25\x43\x04\x1c\x9a\xcd\x65\
   x98\xf9\xf9\xec\x15\x22\xe5\x60\x28\x26\x5c\xf0\xe9\xb1\x4f\
   x79\x91\x46\x94\x33\x48\x7f\x98\xd8\xe2\x91\x70\x4f\x4d\x21\
   x34\x50\x57\xfb\xdf\xc3\x14\xe5\x52\x10\xa4\xa7\x93\x6b\x67\
   xf9\xab\x7f\x30\x13"
e5 = "\xdf\x85\xe8\xbc\xda\xa9\x6e\x69\xfd\xdc\xdd\x82\xbb\xe1\
   xc0\x4e\xc1\x35\x3a\xc6\x3e\xac\xfe\xe3\xdb\xfb\xfb\x3d\xd4\
   x74\xa6\xf0\x0e\x67\x51\x9e\x9f\xb8\x0d\xf4\x57\x23\x2e\x02\
   xb6\xcc\xe9\xcc\x0a\xe7\x0b\x28\x1b\x14\xb4\x38\xc1\x92\x84\
   xc9\x0e\x84\x4d\x84"
e6 = "\x37\x29\x2f\xf4\x92\xf8\xc2\x0e\xbe\xb4\x99\x88\xf5\x21\
   xb3\x36\xed\x50\x43\xd6\x50\xa1\xce\x84\xf5\x31\x26\x94\x8f\
   xdc\x5a\xc5\x22\x4a\x4d\x84\x38\x11\xe2\x9d\x0c\x09\xf5\x85\
   x47\xa4\x18\x67\x3f\xeb\x72\x84\xc8\x89\x06\x3a\x85\x7a\x10\
   xcb\x40\x7d\xa9\x05"

#Coefficient data
c0 = "\x9d\xb5\xd6\xfc\x99\x31\xcf"
c1 = "\x13\x6d\xb9\x15\xff\x54\x69"
c2 = "\x14\xaa\x13\x45\x07\x53\x83"
c3 = "\xd5\x2f\xcf\x8b\xe2\x0e\x6b"
```

```
c4 = "\xf4\x5f\xf7\x24\x74\x9a\x21"
c5 = "\x5c\xb7\xeb\x73\x0b\x47\xf1"
c6 = "\xc9\xe0\x9b\xd9\x29\x71\x97"

#GENID
g0 = "\xaa\xbb\xcc\xdd\xee\x11\x22\x33\x00"

#Packet ex + cx for x in 0-6:
p0 = g0 + c0 + e0
p1 = g0 + c1 + e1
p2 = g0 + c2 + e2
p3 = g0 + c3 + e3
p4 = g0 + c4 + e4
p5 = g0 + c5 + e5
p6 = g0 + c6 + e6

l2packet0 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p0)
l2packet1 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p1)
l2packet2 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p2)
l2packet3 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p3)
l2packet4 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p4)
l2packet5 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p5)
l2packet6 = scapy.Ether(type=eth_type,src=eth_src,dst=eth_dst)/
   scapy.Raw(load=p6)

s = scapy.conf.L2socket(iface="br0")

for i in range(0,20000):
 s.send(l2packet0)
 s.send(l2packet1)
 s.send(l2packet2)
 s.send(l2packet3)
 s.send(l2packet4)
 s.send(l2packet5)
 s.send(l2packet6)
```

Scapy pcap analyzer

```
from scapy.all import *
import numpy as np

tx_times = []
rx_times = []

print("Reading pcap file..")
packets = sniff(offline=str(sys.argv[1]))

print("Reading done.\nAnalyzing captured packets..")
gencount = 0
packetcount = 0

txcount = 0

flag = 0;

for packet in packets:

 if flag == 1:
  flag = 0;
  continue;

 if hex(packet[Ether].type) == "0x0806":
  flag = 1;

 if hex(packet[Ether].type) == "0x2020" or hex(packet[Ether].
    type) == "0x8000":

  #print('%s %.30f' % (hex(packet[Ether].type),packet.time))
  if hex(packet[Ether].type) == "0x8000":
   tx_times.append(packet.time)
  if hex(packet[Ether].type) == "0x2020":
   rx_times.append(packet.time)

print(len(tx_times))
print(len(rx_times))

#print(rxtx_deltas)
if len(tx_times) > len(rx_times):
 #Remove last elements as packets dropped/lost
```

```
 tx_times = tx_times[:len(rx_times)]

rx_times = rx_times[:len(rx_times)]

#Throughput, based on a single generation, and not over the
    entire test.. but averaged over the test.
#Input rate: Total bytes / From first packet to last Tx packet
    time
throughputs_t = np.array(tx_times[6::7]) - np.array(tx_times
    [0::7])
print("Tx input only Throughput mean: Mbps", ((448*8)/np.mean(np
    .array(throughputs_t)))/(1000000) )

#Ouput rate: Total bytes / From first packet to last Rx packet
    time
throughputs_t = np.array(rx_times[6::7]) - np.array(rx_times
    [0::7])
print("Rx out only Throughput mean: Mbps", ((448*8)/np.mean(np.
    array(throughputs_t)))/(1000000) )

#Thoughput time between last Tx and first Rx:
throughputs_t = np.array(rx_times[0::7]) - np.array(tx_times
    [6::7])
print("Between last Tx and first Rx Throughput mean: Mbps",
    ((448*8)/np.mean(np.array(throughputs_t)))/(1000000) )

#Last rx bit of last packet
throughputs_t = np.array(rx_times[6::7]) - np.array(tx_times
    [0::7])
print("Throughput (of last Rx packet) mean: Mbps", ((448*8)/np.
    mean(np.array(throughputs_t)))/(1000000) )

#First rx bit of first packet
throughputs_t = np.array(rx_times[0::7]) - np.array(tx_times
    [0::7])
print("Throughput (of first Rx packet) mean: Mbps", ((448*8)/np.
    mean(np.array(throughputs_t)))/(1000000) )

#Latency
rxtx_deltas = np.array(rx_times) - np.array(tx_times)
rxtx_deltas_split = np.array_split(rxtx_deltas,len(rxtx_deltas)
    /7)
rxtx_deltas_split_avg = [np.mean(arr) for arr in
    rxtx_deltas_split]
```

```
print("Average Latency:",np.mean(rxtx_deltas_split_avg))

#Jitter
print("Jitter:",np.mean(np.abs( np.diff(rxtx_deltas) )))
```