Towards Finite Field Primitives in Network Switches

Daniel Gouveia da Costa Seara

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Fernando Manuel Valente Ramos
Prof. Muriel Médard

Examination Committee

Chairperson: Prof. António Paulo Teles de Menezes Correia Leitão
Supervisor: Prof. Fernando Manuel Valente Ramos
Member of the Committee: Prof. João Luís Da Costa Campos Gonçalves Sobrinho

November 2022

Acknowledgments

First and foremost, I want to thank my supervisors, Professor Fernando Ramos, and Professor Muriel Médard. The guidance, support, and availability to accompany me in this work were outstanding and, above all, an inspiration.

To the leaders of my scouts group, Grupo 23 de Queluz, a special thank you for the many memories, adventures, and moments we had together. Working with you in order to give so many young people the tools to be better in this world is a real pleasure and a privilege. Your strength and help were paramount for this work to reach a happy ending. In particular to Cláudia and Marco, for, alongside myself, closely accompanying thirty young people and making them grow and, most importantly, dream.

To Catarina and André, my closest high school friends, a thank you for the conversations, the laughs, and the support you gave me, even when I didn't ask for it.

I would also like to thank my friends who accompanied me through this journey. Special thanks to Afonso and Marcelo, my two most loyal partners in the university's projects since we first met each other at our courses boot camp way back in 2017. Also, a special appreciation for Margarida, who motivated and listened to me during this time.

Finally, and most importantly, to my parents, Artur João Seara and Maria Paula Gouveia. Your continuous dedication and support are my biggest motivation. Also, thank you for the sacrifices you made in order for me to get the best education and allow me to discover myself, grow and, most importantly, learn. Without you, this thesis would not happen.

Abstract

Finite Field arithmetic is the building block of many networking use cases, from Cryptography to Network Coding and Forward Error Correction. The flurry of innovation triggered by the ability to reprogram the network data plane has recently enabled solutions that run simplified versions of these complex use cases in the data plane. These experiences have shed light on the difficult trade-offs involved in the design and implementation of these operations under the computational constraints of high-speed switches. We, thus, find a challenge: can Finite Field operations be implemented and run at line rate in current high-speed network switches and are generic enough to fulfill the most common requirements, or do the characteristics of existing architectures fundamentally preclude any useful implementations of these operations?

As a first step towards a solution for this challenge, the work of this thesis examines common approaches for the design of Finite Field primitives and discusses in-network implementations of these operations for current high-speed, production-level, programmable switches, as well as a prototype for a new switch architecture. Our findings showcase that the current hardware can only perform Finite Field operations on Field sizes with at most eight bits, and cannot perform enough of these operations in parallel to counter that fact. The more recent prototype presents better results, achieving operations over 56-bit Fields for the multiplication operation. These results show that, although the prototype is a step in the right direction, it either needs to be refined, or a new architecture needs to be created if we want to be able to implement to real-life use cases.

Keywords

Finite Field Arithmetic; Programmable Switches; Data plane architectures

Resumo

A aritmética de Corpos Finitos é uma das bases de muitos casos de uso em redes de computadores, desde Criptografia até *Network Coding* e *Forward Error Correction*. A grande quantidade de inovação despoletada pela capacidade de reprogramar o plano de dados fez com que aparecessem soluções capazes de executar estes casos de uso complexos no plano de dados, embora apenas versões simplificadas. Estas experiências demonstraram os compromissos envolvidos no desenho e implementação destas operações sobre os constrangimentos computacionais dos comutadores de rede que operam a altas velocidades. Devido a isto, encontramos um desafio: será que as operações sobre Corpos Finitos podem ser implementadas e executadas a taxa de linha nos comutadores de redes atuais, e será que essas operações são genéricas o suficiente para cobrir os requisitos mais comuns, ou será que as características das arquiteturas disponíveis fundamentalmente precedem qualquer implementação útil? Como primeiro passo para encontrar uma solução, o trabalho desta tese examina as vias mais comuns para desenhar primitivas de aritmética em Corpos Finitos, e discute implementações destas operações nos dispositivos de rede atuais, mas também numa nova arquitetura prototipada. As nossas descobertas mostram como o *hardware* atual apenas consegue operar sobre Corpos Finitos com tamanho até oito *bits*, e não consegue realizar operações suficientes em paralelo para contrariar esse facto. O protótipo, mais recente, apresenta melhores resultados, conseguindo operar sobre 56 *bits* para a multiplicação. Estes resultados mostram que, embora o protótipo seja um passo na direção correta, este precisa de ser remodelado, ou uma arquitetura nova necessita de ser criada se queremos implementar casos de uso reais.

Palavras Chave

Aritmética de Corpos Finitos; Comutador programável; Arquiteturas de plano de dados

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AES	Advanced Encryption Standard
AES-GCM	AES-Galois Counter Mode
AES-ECB	AES-Eletronic Codebook
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CGRA	Coarse-Grained Reconfigurable Array
CPU	Central Processing Unit
CU	Compute Unit
DSL	Domain Specific Language
DPDK	Data Plane Development Kit
FEC	Forward Error Correction
FF	Finite Field
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
FU	Functional Unit
GCD	Greatest Common Divider
GF	Galois Field
IoT	Internet of Things
IPsec	Internet Protocol Security
MAT	Match-Action Table
MIPS	Microprocessor without Interlocking Pipeline Stages

ML	Machine Learning
MU	Memory Unit
NC	Network Coding
P2P	Peer-to-Peer
P4	Programming Protocol-independent Packet Processors
P4-SDE	P4-Software Development Environment
P4i	P4 Insight
PHV	Packet Header Vector
PISA	Protocol Independent Switch Architecture
PR	Pipeline Register
RLNC	Random Linear Network Coding
RPA	Russian Peasant Algorithm
SDN	Software-Defined Networks
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
TCAM	Ternary Content-Addressable Memory
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time-To-Live
VLIW	Very Long Instruction Word

1 Introduction

Contents

From the vast pool of networking applications present nowadays, a considerable number require some form of Finite Field (FF) or Galois Field (GF) arithmetic. Cryptography, Network Coding, and Forward Error Correction are some concrete examples. To further attest to the importance of said operations, modern Central Processing Units (CPUs) already contain dedicated, built-in instructions to efficiently perform Finite Field arithmetic operations. However, the growing performance, scalability, and security requirements of modern distributed applications are pushing many functions to be network-accelerated, running directly in the data plane of network switches.

Most network-based applications and protocols that include Galois Field computations run in software. While for end-host based protocols such as Transport Layer Security (TLS) [6], this is not an issue, for new cryptographic algorithms (e.g., the EPIC protocols [7] of the SCION network architecture) or for Network Coding, software based solutions (e.g., Kodo [8]) do not have the required performance. Our goal in this thesis is to run these operations directly in network switches, in order to improve packet processing from a few Gbps as in high-performance software implementations, to Tbps scales, by running them directly in the switch data plane.

The main challenge is that, in order to process packets at line rate and at Tbps scales, the modern switch hardware architectures are restricted to a limited number of very simple operations. As a natural consequence, the current specification of P4, a language to program these devices, does not allow for the execution of common operations in CPUs, like multiplication and division. Furthermore, memory is also a scarce resource, meaning that stateful operations are also very limited.

The question we thus ask in this thesis is as follows. Can we implement Finite Field arithmetic efficiently and at line rate in current network switches? Or do we fundamentally need to redesign the data plane architecture? We try to answer these questions in this thesis.

1.1 Main Contribution

With these questions in mind, our main contribution is to give a concrete look into the design and implementation of Finite Field arithmetic in state-of-the-art programmable switches. We explore different avenues to perform these operations, looking at the two main approaches, memory-based or computational-based. We implement algorithms of each type and discuss their benefits and limitations. As a last result, we make the case that current switch architectures are insufficient to perform Finite Field arithmetic even for the most basic use cases.

We also take a step forward by implementing these operations in the prototype of a newly proposed data plane switch architecture. We show that this architecture improves over the state-of-the-art, and by a significant amount in one of the operations. We also discuss how these improvements are not enough and some refinements or entirely new architectures are necessary.

1.2 Organization

The remainder of this dissertation is the following.

The next Chapter presents background and related work. It starts with a primer on Finite Fields and their use cases and implementations. Afterward, it provides some background on Programmable Networks and Programmable Switches, detailing the data plane architectures we will be exploring for this work.

In Chapter 3, we explain how to perform the most basic operations over Finite Fields: addition, subtraction, multiplication, and division. We present different approaches and algorithms for performing them, as well as a discussion of their benefits and drawbacks.

Chapter 4 showcases the solutions we designed and implemented to perform GF operations in both current hardware as well as new data plane architectures.

In Chapter 5 we evaluate all of our solutions.

Finally, we conclude the thesis in Chapter 6 and we present a discussion about the different solutions.

2

Related Work

Contents

In this chapter, we present the related work of this thesis. We start with a primer on FFs, defining them and showcasing their properties. We then present some current use cases where Finite Fields are used, mainly in the field of networking. Next, we discuss the work done in the field of Programmable Networks, showcasing Software Defined Networks, and the P4 and Spatial languages. Afterward, we present the current state of Programmable Devices and their architectures. Finally, we show current implementations of Finite Field operations in several types of hardware, from common CPUs to switch Application-Specific Integrated Circuits (ASICs), and Field Programmable Gate Arrays (FPGAs).

2.1 Finite Fields

In order to get to Finite Fields, we first start with the definition of a Group [9]. A Group G is a set of elements and a rule called the law of composition. This rule associates each pair of elements $x, y \in G$ to a new element $x \times y \in G$ in the case of a *multiplicative Group* (a multiplication) or $x + y \in G$ in the case of a *additive Group* (an addition). We focus on the latter, however, these properties are the same for the former, just replacing the $+$ with a $\times$. In order for a set to be a Group, three properties need to be upheld.

1. For all $x, y, z \in G$, there exists associativity. In other words, $(x + y) + z = x + (y + z)$.
2. There exists an element $e \in G$ such that $e + x = x + e = x$. For additive groups, it is normally called the 0 element ($x + 0 = 0 + x = x$). For multiplication, it is the 1 element.
3. If $x \in G$, then there exists an element $y \in G$ such that $x + y = y + x = e$. For additive groups $x + y = y + x = 0$. This is called the inverse element and can be represented as $-x$ for addition and x^{-1} for multiplication.

There is one more property that is not mandatory for all Groups to uphold. However, if they do, they are called *abelian Groups*. This property is called commutation and states that for all $x, y \in G$, $x + y = y + x$. This is an important property for defining a Field.

A Field F [10] is also a set of elements but with more properties that need to be maintained, and for both addition and multiplication. These properties are:

1. F is an abelian Group with respect to addition.
2. $F^* = F \setminus \{0\}$ is a Group with respect to multiplication (Note that it does not need to be abelian).
3. Multiplication is distributive with respect to addition. In other words, we must have that for all $x, y, z \in F$, $x \times (y + z) = x \times y + x \times z$ and $(x + y) \times z = x \times z + y \times z$.

As one can see, a Field is a specialization of a Group. Every Field is a Group, but not all Groups are Fields. Every Field is also a Ring [10], which is a similar construct but missing one important property. A Ring does not need to be a Group for multiplication. Its multiplication must only be associative. What does this mean? It means that a Field has an element 1 such that for all $x \in F$, $x \times 1 = 1 \times x = x$ and, most importantly, for all $x \in F^*$, there is an *inverse element* with respect to multiplication $x^{-1} \in F$ such that $x \times x^{-1} = x^{-1} \times x = 1$. A Ring does not have this element. As such, one can also define a Field as a unitary Ring that admits an inverse for all of its non-zero elements [10].

Some of the most common sets of numbers are examples of Fields. The set of rational numbers $\mathbb{Q}$ is a Field, as it respects all its properties. There is associativity (and even commutation) for both addition and multiplication, there exists a 0 element and a 1 element, there exists an inverse for all non-zero elements, and the distributive property is assured. The same can be said for the Field of complex numbers $\mathbb{C}$ and the Field of real numbers $\mathbb{R}$ [10]. However, as counter-examples, the sets of both natural numbers $\mathbb{N}$ and integer numbers $\mathbb{Z}$ are not Fields. The former is because it does not comply with the third property of a Group with respect to addition. There are no two elements that produce zero as a result when added – the latter since it does not have an inverse for multiplication [10].

There is a crucial difference between these Fields we mentioned and the ones relevant to our work. They all have an infinite number of elements. The Fields we are interested in have a *finite* number of elements. These kinds of Fields are commonly known as Finite Fields or Galois Fields [9]. The most common GFs come from the set of integer numbers $\mathbb{Z}$, but only considering the elements up to, but not containing, a certain value p (recall the full set $\mathbb{Z}$ is not a Field). In this case, the operations are done modulo p. As a concrete example, we can look at the GF $\mathbb{Z}_5 = \{0, 1, 2, 3, 4\}$ [10]. All the operations are done modulo 5, so, for example, $2 + 3 \equiv 5 \mod 5 \equiv 0 \mod 5$. Table 2.1a shows the addition table and Table 2.1b show the multiplication table. By looking at all the possible combinations of elements, it is easy to see that all the properties of a Field are upheld. There is a 0 element and a 1 element, and we can quickly see that there is distribution. As a quick example, $4 \times (2 + 3) \equiv 4 \times 0 \mod 5 \equiv 0 \mod 5$ and $4 \times 2 + 4 \times 3 \equiv 3 + 2 \mod 5 \equiv 0 \mod 5$.

+	**0**	**1**	**2**	**3**	**4**
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

(a) Addition Table

$\times$	**0**	**1**	**2**	**3**	**4**
0	0	0	0	0	0
1	0	1	2	3	4
2	0	2	4	1	3
3	0	3	1	4	2
4	0	4	3	2	1

(b) Multiplication Table

Table 2.1: Addition and Multiplication Tables for Z_5

Not all integer values can be used as p, however. As an example, $\mathbb{Z}_4 = \{0, 1, 2, 3\}$ is not a Field.

That is because the set $\mathbb{Z}_4^*$ is not a Group concerning multiplication. $2 \times 2 \equiv 0 \mod 4$ and $0 \notin \mathbb{Z}_4^*$ and $2 \times 1 \equiv 2 \times 3 \mod 4$ meaning that if the element 2 had an inverse, we would have $1 = 3$, which is, of course, wrong [10]. In general, in order for a set $\mathbb{Z}_p$ to be a Galois Field, the value of p must be a prime number greater or equal to 2. The proof for this can be seen in [11].

Fortunately, we are not bound only to use positive integers modulo p to create Galois Fields. We can, for example, work in the polynomial space, where each element of the Group, Ring, or Field is a polynomial. As [10] proves, Galois Fields in the polynomial space will always be defined by a *prime power* p^m, where m is an integer greater or equal to 1. We represent these fields by $GF(p^m)$, and they are composed of all the polynomials with degree less than m (x^{m-1}) and coefficients that belong to Z_p (we can also represent Z_p as $GF(p)$). These fields will have p^m elements. We call this the size or cardinality of a Field. As a concrete example, the field $GF(3^2)$ will be composed of polynomials with degree up to 1 and coefficients in $\{0, 1, 2\} : \{0, 1, 2, x, x + 1, x + 2, 2x, 2x + 1, 2x + 2\}$.

Remember that with Fields $\mathbb{Z}_p$, the construction would be all of the values modulo p. The construction is the same with the Galois Fields $GF(p^m)$, but it must be made modulo a polynomial. This polynomial must be *irreducible* over the Field, meaning it can't be the product of two other elements of the Field. The polynomial must also be *monic*, meaning all of its coefficients must belong to the $GF(p)$ Field, and it must have degree m, with the coefficient of x^m equal to 1. If we have such a polynomial (both monic and irreducible), we have found a *prime polynomial*, and we can use it to construct the Field by using the rules of the quotient rings [10].

But there exists even another way to construct these Fields. If we have a prime polynomial P, there might be a case where one of its roots is a *primitive* of the Field. A primitive α of a field is an element that does not belong to $GF(p)$ but does belong to $GF(p^m)$, and all values of the $GF(p^m)$ Field can be expressed as a power of α with exponents ranging from 1 to $p^m - 1$. Additionally, α^{p^m-1} must be 1 [10]. The prime polynomial that contains such a root is called the *primitive polynomial*. Another name given to the primitive element α in the literature is the *generator*. We will use both terms interchangeably. One important thing to note is the existence property. No matter the GF, there will always be at least one primitive polynomial [10]. As a quick example, one primitive polynomial of $GF(2^4)$ is $x^4 + x + 1$ with primitive element $x + 1$.

2.1.1 Finite Fields with p = 2

In computer science, the class of Fields which are more relevant to the use cases we will explore later is $GF(2^m)$. These fields are composed of polynomials with degree up to m, but crucially the coefficients of the polynomial are in $GF(2)$, meaning they can only be 0 or 1. What this means is that each coefficient will fit in exactly one bit. We can then represent each polynomial $c_0 + c_1x + c_2x^2 + ... + c_{m-1}x^{m-1}$ as a vector $[c_0, c_1, c_2, ..., c_{m-1}]$, or a number with at most m bits.

This is the fundamental intuition behind these Fields since now we can think of any Finite Field $GF(2^m)$ as the set of all numbers that fit in at most m bits. As an example, the field $GF(2^4)$ will contain all of the values that fit in exactly four bits, from $0 = 0000b = 0 + 0x + 0x^2 + 0x^3$ to $15 = 1111b = 1 + 1x + 1x^2 + 1x^3$. Throughout the rest of this thesis, we will focus exclusively on this class of Finite Fields.

To conclude this section, we remark that one of the most important properties of Finite Fields is *closure*. Remember that all the operations performed over a Finite Field need to result in an element that is also part of said Field. As such, if we work with a Finite Field $GF(2^m)$, we use all the values that fit in exactly m bits. Because of this, all operations we do will result in a value with at most m bits as well, contrasting with standard computational arithmetic where multiplication, for example, can result in a value with double the number of bits as its operands.

2.2 Finite Field Use cases

In this section, we motivate the usage of Finite Fields. Since the focus of this thesis is to run operations over Finite Fields in network devices, the presented use cases are all related to networking. We chose three – In-network Cryptography, Network Coding, and Forward Error Correction – for three main reasons. First, they represent techniques that are widely used in a variety of contexts. Second, the possibility of performing these computations entirely in the data plane of high-speed programmable switches and routers can open the door to an array of new applications for various types of networks, including data centers, service provider networks, and even the global Internet. Third, they often require very large Galois Fields (e.g., $GF(2^{128})$ and above), making them very challenging to implement in modern high-speed network switches.

2.2.1 Cryptography

Implementing security solutions in the switch data plane is an appealing idea that can drastically change the network security landscape. For instance, it would enable implementing security checks *on all packets* as they traverse the switch, instead of on only a small subset of the traffic – or ignore security altogether, as is common. One can anticipate performing per-packet authentication [7] or checks to proof-carrying network codes [12] directly in the network switch. Other benefits of in-network security include eliminating the need for offloading security tasks to a centralized controller [13–16] or for mirroring traffic to powerful middlebox hardware [17]. This can not only avoid bottlenecks but also result in significant reductions of bandwidth and compute overheads, with a positive effect on network latency as well. Furthermore, it can provide an increased level of security and privacy within the network (e.g., by obviating attacks against single points of failure).

Significant efforts have been devoted to developing cryptographic in-network security solutions. For example, in the context of the SCION Internet architecture [18], Legner et al. [7] proposed a family of data-plane algorithms, EPIC, with increasing security properties in an effort towards building a secure path-aware network architecture. This solution requires Advanced Encryption Standard (AES) [19] to be performed directly in the network devices (EPIC [7] was implemented on a server with Intel Data Plane Development Kit (DPDK)).

Recently, major network operators, such as Amazon and Microsoft, have started adopting cryptographic protocols like MACsec (which, at its core, also uses AES) to encrypt their traffic not only at the application layer but also at the link layer [20–22]. Notably, the level of security provided by many cryptographic algorithms is directly connected to the size of the Galois Field over which their operations are performed. For instance, while AES-Eletronic Codebook (AES-ECB) mode uses a relatively small Field ($GF(2^8)$), this mode is considered to be insecure and should never be used in practice [23]. Secure AES modes widely used in practice, such as AES-Galois Counter Mode (AES-GCM) (e.g., as used in TLS and Internet Protocol Security (IPsec) [6]), requires GF operations on a Field with (at least) 128 bits ($GF(2^{128})$) [24].

2.2.2 Network Coding

IP networks commonly use a *store-and-forward* mechanism for packet forwarding. Network Coding (NC) [1] proposes an alternative: *store-code-forward*. Instead of just forwarding packets, network devices are now responsible for encoding multiple packets together into a *linear combination* that is then output to the next hop. This mechanism has been shown to bring advantages in terms of throughput, robustness to packet loss, and increased security [1].

The most simple and common example of Network Coding is presented in Figure 2.1. Consider the two nodes *A* and *B* that want to exchange packets in a wireless network, using access point *S*. *A* sends *a*, and *B* sends *b*. With a conventional access point, *S* would be forced to broadcast *a* and then *b*. With Network Coding, *S* can combine *a* and *b* using the `XOR` operation and broadcasts this instead. Due to the properties of `XOR`, *A* and *B* can trivially recover the wanted packet, and the number of transmissions is reduced from four to three.

The NC technique has been widely used in practice and in many contexts. One of the first known applications was Avalanche [25], an NC-based Peer-to-Peer (P2P) content distribution system used for Microsoft Secure Content Distribution. Companies like Veniam [26] are also using NC techniques to improve throughput in WiFi in the Internet of Things (IoT) space. Recently, Network Coding has also been proposed to improve the throughput of inter-datacenter bulk transfers [27].

Network coding uses GF arithmetic at its core. The encoding operation is defined by $X = \sum_{i=0}^{n} g_i M_i$, where each M_i is an original packet, and each g_i a coefficient that can be either defined deterministically

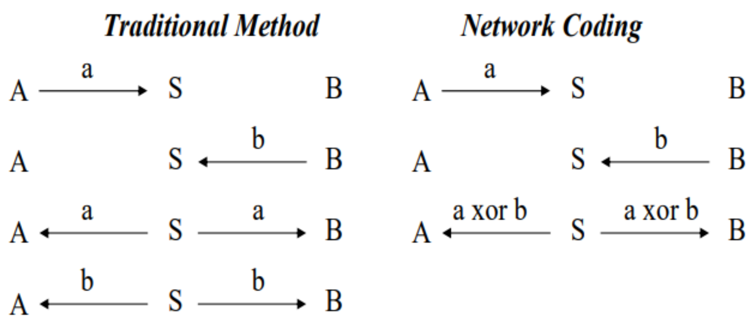


Figure 2.1: Example of Network Coding in a wireless setting, taken from [1]

(e.g., by some centralized entity) or, more commonly, can be randomly selected (as in Random Linear Network Coding (RLNC) [28]). Each packet is multiplied by a coefficient, and the results are summed together to form a new packet X. Both multiplication and summation operations are performed using Finite Field arithmetic to ensure the resulting packet is the same size as the original packets.

The devices in the path can also recode the combinations they receive into new combinations (to improve the chances of decoding success, as new independent linear combinations are being created). This recoding process is at all equal to the original encoding, but instead of working with the original packets M, the node operates on several encoded packets X_i to create a new one, X'. There is, however, one extra step that needs to be taken. The coefficients used by the node that recodes the packets and creates X' need to be multiplied by the coefficients that were used to create the X_i packets so that they relate to the original data M and can be used to retrieve the information.

The egress is, finally, responsible for decoding the coded packets to retrieve the original information. For decoding to succeed, it is necessary to have enough linearly independent combinations. At least equal to the number of original packets. When this happens, we have a linear system of equations, which means we can leverage Gaussian Elimination to solve the system and recover the original information. All the operations performed while executing Gaussian Elimination follow the rules of Finite Field arithmetic.

In theory, Network Coding does not require very large Fields, with some solutions using simple bitwise XOR ($GF(2^1)$), like the presented example, or per-byte ($GF(2^8)$) operations. However, larger Fields have the advantage of increasing the probability of coding success in the most common randomized settings [29]: there is a higher probability of linear independence between combinations on a larger Field. There is also a practical challenge in coding packet payloads using small Fields, which we illustrate with an example. It is not trivial to code, for instance, a 1000-byte payload using a $GF(2^8)$ field, as it requires chopping the payload into one thousand “cells” [30], and perform one thousand multiplications in parallel. Switch primitives for Finite Field operations *over large Fields* can thus help enable new NC use cases.

2.2.3 Forward Error Correction

Finally, Forward Error Correction (FEC) codes [31] are an important mechanism for reliable network communication. The message's sender will send the packets with the data through the network, but to account for losses of information, it also appends some redundant information to the packets. This way, the receiver of the message can reconstruct packets that have been lost. The main difference between this approach and a protocol like Transmission Control Protocol (TCP) is that FEC codes can restore a certain number of the lost packets of the communication, regardless of the exact packets that were not received. TCP, in contrast, resends exactly the packets that are thought to be lost when a specific timeout expires. The most used type of FEC codes is Reed-Solomon codes [32].

The main advantages of this approach are twofold. First, the time to restore missing information is reduced, compared to feedback approaches (like TCP). Second, in multicast settings, the sender is not required to resend packets that were not received by, for example, only a small subset of the total receivers of the message, improving the overall achievable throughput [31].

The encoding of data to create the FEC codes and the decoding process are similar to that of Network Coding, which was previously discussed. The packets are seen as values over a Finite Field $GF(2^m)$, and the sender multiplies each one of them by some coefficient before summing it all together in a new packet. The decoder also uses Gaussian Elimination to recover the lost packets. The main difference compared to NC is that, in this case, original packets are circulating through the network, not only encoded ones. As such, the receiver, upon obtaining a new original packet, can directly plug said packet into the linear system of equations and expedite the process of retrieving encoded information.

FEC codes have seen a lot of usage in networking systems like low latency 5G networks [33], media streaming over wireless networks [34], and multiple description source coding [35]. Recent systems [36] have proposed improving network reliability by performing in-network FEC. However, the proposed solution does not run entirely in a programmable switch, as the computations required for FEC (Finite Field arithmetic) exceed the devices' capabilities. It offloads the computations to an external FPGA or a standard CPU, significantly lowering the achievable throughput.

2.3 Programmable Networks

If we want to perform Finite Field operations in the devices, we cannot rely on traditional IP networks. Due to their complexity and heterogeneity, these networks are generally difficult to manage and maintain. There are many different devices, like switches, routers, and firewalls, among others, from different vendors and various models. Every one of them needs to be configured by the network administrator, often manually and using different low-level interfaces per manufacturer. Traditionally, each device contains both the mechanism that decides how to handle the traffic (known as the *control plane*) and the mechanism that forwards the traffic flows based on the rules defined (known as the *data plane*). If a new protocol is to be deployed, the process is slow and complex due to this coupling between planes.

2.3.1 Software Defined Networks

Software-Defined Networks (SDN) [2, 37] has emerged as a new network paradigm that aims to remove the barriers and problems described above. The main idea behind it is to decouple the control plane from the data plane, which gives greater flexibility to the network operator and an easier way to test and deploy new protocols. This separation means that the devices are now only forwarding elements of the network, and the decision process is *logically* centralized in the SDN controller. The programmed applications are executed in the controller, which abstracts the interaction with the network devices.

The architecture of an SDN is depicted in Figure 2.2. The topmost layer is the *Network Applications*. The applications define the network behavior. As examples, there are applications for SDN in routing [38], network visualization [39], and even as the entire backbone of Google's network [40]. The SDN controller then implements this behavior in the network devices which execute it. The *Controller Platform* layer is where the SDN controller resides. This controller abstracts the lower-level layer and considers the network applications and other policies defined in its interaction with the *Network Infrastructure* layer. The latter is where the networking devices actually reside and forward the traffic according to the rules from the controller. Since we can have multiple devices from multiple different vendors, each one with its own implementation details, the abstraction provided by the controller is crucial.

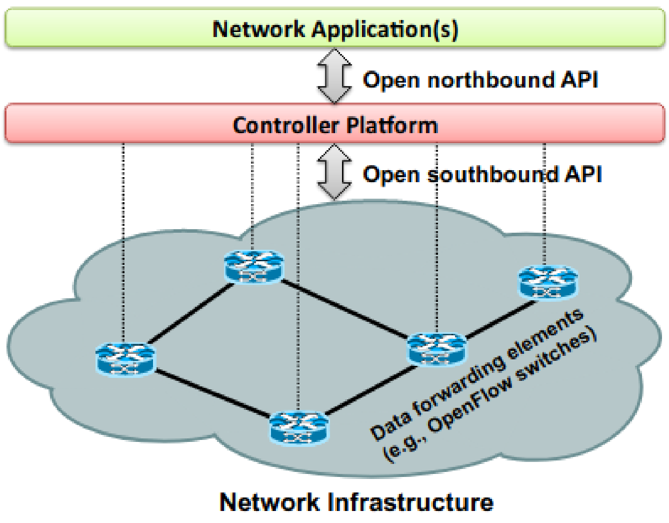


Figure 2.2: Software Defined Networks overview, from [2]

An SDN controller is interpreted as a single entity, however, it does not need to be *physically* centralized in a single device. This would be a significant problem for larger-scale implementations of SDNs. Instead, the controller can be *logically* centralized and physically distributed, abstracting to the user a centralized view of the network. The most commonly used controller implementation is ONOS [41], an open-source project that contains high-level abstractions and run-time extensible modules that the network applications can leverage.

The communication between the applications and the controller is enabled by the *Northbound API*. This Application Programming Interface (API) is not standardized but is usually REST-based. As for the communication between the controller and devices, the *Southbound API* is the communication channel that enables the forwarding rules to be sent from the former to the latter. OpenFlow [42] is the standard protocol for this interface.

2.3.2 P4 language

The OpenFlow protocol has many advantages, like being open source and having a vendor-agnostic interface. However, it also presents some issues. The main problem is its inflexibility and difficulty in evolving. As operators want switches to expose more capabilities to the SDN controller in the form of new data plane protocols, more header fields and table types need to be supported. As an example, OpenFlow *v1.0* could match packets on 12 header fields, while OpenFlow *v1.4* already supports 41. As there was no sign that this increase would stop [43], a new future-proof design was needed.

Programming Protocol-independent Packet Processors (P4) [43] was developed as a high-level programming language that could program any compliant device in a unified manner, serving as an ”OpenFlow 2.0”. P4 was designed to achieve three main goals. Firstly, *target independence*. P4 is unaware of the specific platform it is configuring. Only the compilers are made specifically for the target device, translating the P4 code to the machine code the target can understand. Secondly, *protocol independence*, since P4 is generic enough to specify a great variety of network behaviors without being subject to the characteristics of a certain protocol. Thirdly, *reconfigurability*, because the logic for processing the packets can be redefined by the programmer, even after it is deployed to the target devices. Note that P4 is used for the functionality of the data plane, not the control plane's logic. That is still left to the controller. The typical Southbound API is now the P4Runtime [44].

A P4 program has several key components.

(i) *Headers* specify some fields' constraints, like their width and possible values.

(ii) *Parsers* specify how the headers are identified and the valid header sequences within a packet.

(iii) *Tables*, which contain match-action rules that match on a specific field and indicate which action to perform.

(iv) *Actions*, describing manipulations that can be performed to the packet fields.

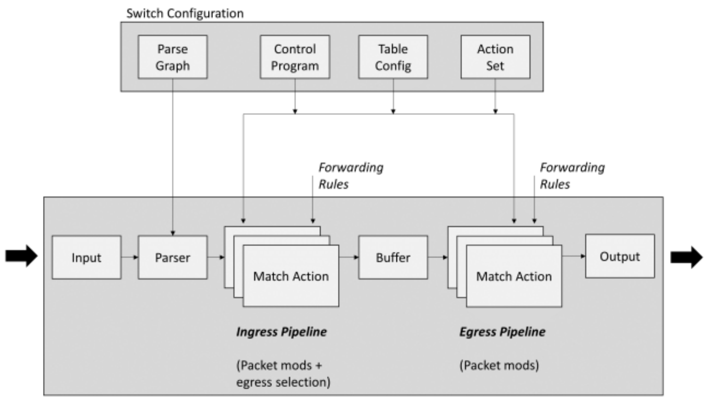


Figure 2.3: P4 Forwarding model

(v) *Control Programs* specify which, and by what order, are Match-Action Tables (MATs) applied to the packets.

The forwarding model is illustrated in Figure 2.3. It requires a programmable parser and multiple match-action rules. The control portion has two responsibilities, configuring and populating. The first defines the match-action rules, while the second fills the tables according to the defined rules.

The sequence of execution is as follows. Once a packet arrives, it first goes to the parser. This parser will extract header fields from the packet according to the specifications in the *Headers* and *Parsers* components. Note that there are no assumptions about the protocol the packet is using. The fields are then passed on to the MATs, which perform the defined operations (*Actions*) on the packet, like setting the egress port, setting the destination port, and replicating the packet if needed, for multicasting operations.

2.3.3 Spatial Language

The main goal of P4 was to program Packet Processors, but other types of reconfigurable hardware exist, like FPGAs and Coarse-Grained Reconfigurable Arrays (CGRAs). One of the most recently proposed ways to program these devices is the Spatial Domain Specific Language (DSL) [45], based on the well-known Scala programming language.

Spatial's main purpose is to simplify the programming of this type of hardware, enabling easy development, testing and optimization of the programs. Spatial provides a set of control structures that can be used to express the wanted algorithm in a concise manner, but let the compiler identify and act upon parallelization opportunities. Among these structures, the most common are:

- *Finite State Machine*, similar to a *while* loop.
- *ForEach*, which is parallelizable *for* loop.

- *Reduce*, a scalar reduction loop, which is parallelized as a tree.

The compiler will try to schedule the operations inside these control structures in order to optimize their performance, but the user is free to modify this behaviour using specific directives. Some of them are:

- *Sequential*, that forces the loop instructions to run sequentially (no parallelization)
- *Pipe* that sets the loop instructions to be pipelined
- *Parallel*, allowing the instructions to run in parallel

One of the other key aspects of the language is the memory templates it provides. These templates let the user control the allocation data to the accelerator's memory, but in a more abstract way. The compiler is able to optimize each type of memory in order to extract the maximum performance in terms of memory access latency and resource utilization. Of course, the compiler only allocates resources which are present in the accelerator. Depending on the access patterns identified, memory can be duplicated, banked or buffered by the compiler, all while maintaining the behaviour programmed by the user. Among the supported memory types we draw special attention to Static Random Access Memory (SRAM), Registers and First-In First-Out (FIFO) queues.

Listing 2.1: Simple Spatial Example

```
1 val s = SRAM[Int](16,32)
2 val r = Reg[Int]
3
4 ForEach(16 by 1, 32 by 1){(i, j) =>
5     s(i, j) = i + j
6 }
7
8 r := s(x,x)
```

We showcase a simple Spatial program in Listing 2.1. The program iterates from 0 to 16 with the first variable i, and 0 to 32 with second one, j. It then stores the result of $i + j$ in the correct location of the SRAM. Finally, in line 8, we load to a register the value stored in the SRAM location x, x where x is a supplied input argument. As is obvious, the operations inside the *ForEach* can easily be parallelized, since there are no dependencies. Fortunately, that task is left to the compiler and the programmer does not have to add anything else.

2.4 Programmable Network Devices

The reason behind the creation of P4 was the appearance of a new switch chip architecture. Traditionally, switch chips could only match in a fixed set of packet header fields and had a fixed number of tables, each with a fixed size. If a programmer wanted to add a new table, change the size of a table, or match to different header fields, that was not possible unless new hardware was acquired.

2.4.1 PISA architecture

A reconfigurable match tables switch architecture was initially proposed by [46]. This architecture allowed the forwarding plane of the switch chip, the Match-Action Tables, to be changed without replacing the underlying hardware. This type of architecture is now called *Protocol Independent Switch Architecture (PISA)* [3].

PISA is a *pipelined* architecture, meaning the tables are arranged in a pipeline fashion. When a table has finished processing a packet, that packet goes to the following table in the pipeline, and the first one starts already processing a new packet. As such, the tables always process packets, and the throughput is maximized. This pipeline is divided in stages, and the processing of a stage can be made to depend on the processing of a previous stage, if some part of the packet is modified by the tables that occupy that stage, for example, decrementing the Time-To-Live (TTL) field before deciding if the packet is to be processed further or not.

Switches based on PISA architectures are composed of 3 main parts:

1. A programmable parser
2. The Match-Action Tables
3. A programmable deparser and scheduler

The parser is responsible for receiving a packet and extracting the header values according to the protocols it supports. For example, a parser that supports an IP header must be able to extract the header values it needs, like the source address, the destination address, the TTL, and the flags, for example. The parser's output is the Packet Header Vector (PHV), a set of the extracted fields, as well as some metadata, for example, the port through which the packet entered the switch. Since it is programmable, the parser's hardware will not be optimized for specific protocols but, instead, be flexible for any protocol the programmer wants to implement. In practice, the parser is implemented by using Ternary Content-Addressable Memory (TCAM) tables that match the bits of the packet and follow the rules defined by a state machine the programmer creates [46].

The MATs are then responsible for matching the values of the PHV in their tables and appropriately selecting an action. Each stage comprises an Arithmetic Logic Unit (ALU) for standard boolean and

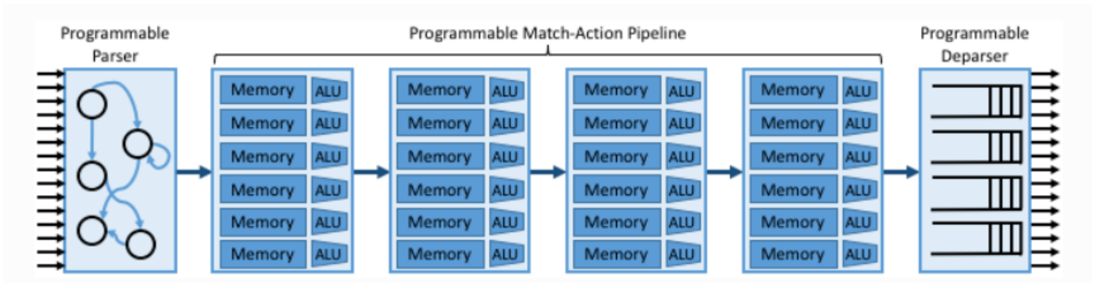


Figure 2.4: PISA architecture, from [3]

arithmetic operations or header modifications, among others. Each *logical* stage in the switch will match the PHV values to a certain number of tables and then execute the appropriate actions using Very Long Instruction Words (VLIWs) [46]. These logical stages are mapped to the *physical* stages in a way that best maximizes the available resources. One important thing to note is that the matches do not need to be exact. This architecture supports ternary matching so that one can use longest-prefix matching, for example. As a concrete example of a Match-Action table, we can define a table that matches the destination IP address of a packet. If there is a hit, the output port is defined so that the packet exits by the appropriate interface.

Finally, the deparser and scheduler are responsible for re-serializing the information of the packet for over-the-wire communication and scheduling its egress. Figure 2.4 illustrates this architecture.

There is already commercially available hardware that implements this architecture. Intel's Tofino [47] is an example of this. The work on this thesis was developed in this platform (more details in Chapter 5).

2.4.2 Taurus

Although PISA is the main data plane architecture used in switches nowadays, a few others have been proposed recently. One of them is Taurus [4], an architecture for performing per-packet Machine Learning (ML). Taurus takes a standard PISA switch as its basis, but it adds custom hardware based on a MapReduce abstraction to the switch pipeline. The efficiency it achieves for ML inference arises from its use of pipelined Single Instruction Multiple Data (SIMD) parallelism, in contrast to the purely VLIW-based architectures of the current network switches we have described previously [46]. Most importantly, this architecture can perform these operations while maintaining the line rate. At a 1GHz clock, this architecture ensures nano-second level latencies [4].

Therefore, the main contribution of Taurus is the MapReduce block that sits between some pre-processing and some post-processing MATs. The diagram of Figure 2.5 illustrates the architecture. As one can see, a parser, deparser, and scheduler still need to exist, which are responsible for the same functionality as in the PISA architecture. The same can be said for the pre-processing and post-processing MATs, which are used for some simple operations (add some information to the metadata of

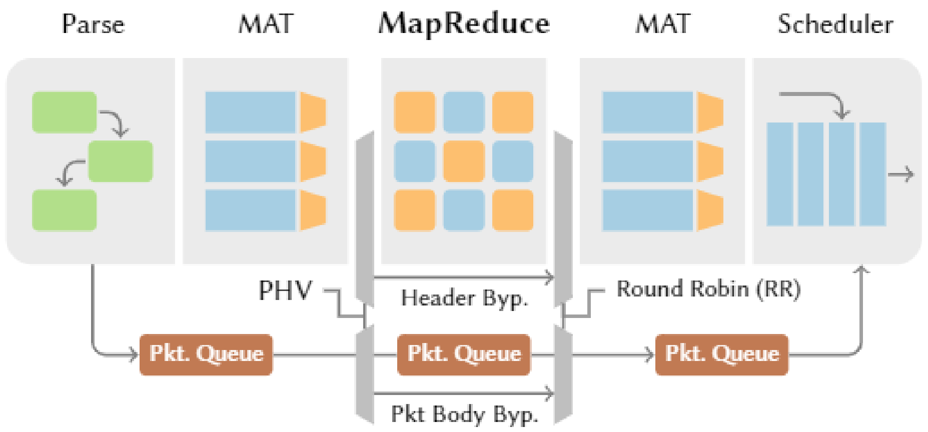


Figure 2.5: Taurus architecture, from [4]

the packet, interpret the ML decisions, forward packets, etc.) before and after the packet goes through the MapReduce block. This block is, thus, the one responsible for SIMD parallelism as it can perform operations in each element of a vector (*Map*) and also reduce the elements to a single scalar (*Reduce*). The lower part of the diagram is a bypass added to the architecture so that packets that do not need to go through the MapReduce block, for example, control packets, can more quickly go through the device.

Looking more deeply into this block, we see it is composed of two units. Compute Units (CUs) and Memory Units (MUs). CUs are the ones that perform the arithmetic operations. They are composed of Functional Units (FUs), which perform one Map, one Reduce or one MapReduce operation, and Pipeline Registers (PRs), which enable the pipeline architecture, as the value from a FU is written to a PR, and then the next FU reads from that PR. Inside the CU, we can have multiple stages and multiple lanes. Each lane contains values from the vector (*multiple data*), which we want to execute operations on, and the FUs in each stage execute the same operation (*single instruction*). Each CU can perform several operations with its multiple stages but always following the SIMD approach. The number of lanes and stages is configurable, paying a cost in the area occupied by each CU and the power consumed [4]. Figure 2.6 showcases a CU with three stages and four lanes, with its FUs and PRs.

MUs act like coarse-grain pipeline registers and are interspersed with the CUs, realizing the pipelining architecture needed for line rate operations. At a 1GHz clock, it is these MUs that enable the nano-second level latencies mentioned previously. Interestingly, these MUs also have some logic incorporated that lets them perform some straightforward computations, like an `XOR` or a `SHIFT`. A full Taurus implementation will have several of these CUs and MUs in a mesh (like in Figure 2.7).

There is currently no actual hardware implementation of this architecture. In order to prove the advantages of Taurus, software simulations based on realistic hardware were conducted. A testbed was also designed to evaluate the end-to-end performance of the architecture. In this case, the MapReduce

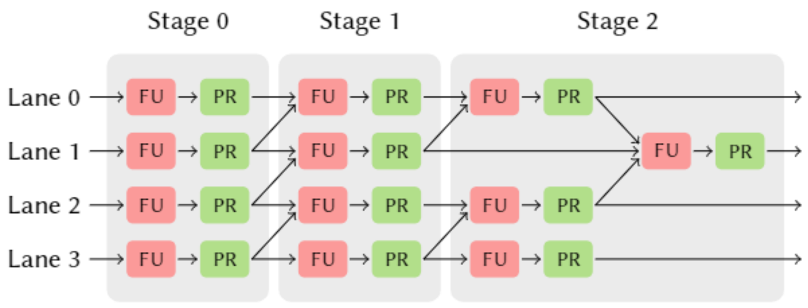


Figure 2.6: An example of a CU with 3 stages and 4 lanes, from [4]

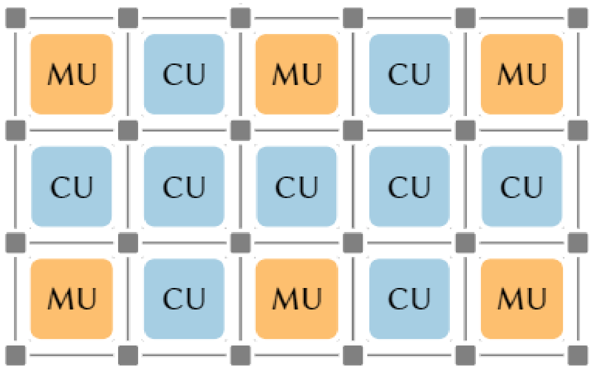


Figure 2.7: Taurus Mesh of CUs and MUs, from [4]

block was implemented on an FPGA [4].

2.5 Implementations of Finite Field Operations

Finite Field operations have been implemented directly in the hardware of various computing architectures, from the common CPUs we see in personal devices to FPGAs and networking switches ASICs. There is, unsurprisingly, a plethora of software implementations of said operations. Chapter 3 will discuss in more detail how to actually perform the operations and the algorithms that are commonly used, but the primary goal of this section is to expose where they have been implemented before.

Software implementations of Finite Field arithmetic are plentiful, using common programming languages. Kodo [8] is a library that performs the aforementioned Network Coding operations, written in C++, and performs the required Finite Field arithmetic using tables of values (more details in Chapter 3). Other languages with libraries for Finite Field operations include: Rust [48], Perl [49] and Python [50].

As for hardware implementations, which aligns more with the focus of this thesis, several modern processors already ship with built-in instructions to perform Finite Field arithmetic. For example, the *x86_64* architecture of common CPUs supports the `PCLMULQDQ` instruction that performs carryless multiplication over a Finite Field [51]. This instruction takes two 64-bit values and computes the 128-bit carryless product. Other instructions are then performed in order to reduce the result to the appropriate defined Finite Field. Other work has proposed extensions to the Microprocessor without Interlocking Pipeline Stages (MIPS) architecture to perform operations over Finite Fields efficiently [52]. For the constraints present in IoT environments, a processor architecture capable of performing such operations has already been proposed as well [53].

Operations over Finite Fields have also been successfully implemented in FPGAs. FPGAs are integrated circuit platforms that can be rearranged by a programmer, using languages like Spatial (Section 2.3.3), in order to perform some desired specific task. Due to the hardware being explicitly rearranged for the task at hand, these boards perform at much higher speeds than the common CPUs. [54–57] are examples of the implementation of Finite Field operations in FPGAs. All the examples focus solely on multiplication since it is the most prominent for the considered use cases. However, they do not regard Finite Field division, which is an operation we consider in this work.

A greater challenge is performing Finite Field arithmetic on a programmable switch ASIC. Recent work has investigated the feasibility of implementing cryptographic algorithms like AES on these devices for in-network cryptography-related use cases. Due to the constraints of the switches' programming language (which we will present in Section 4.1) and of the target devices, several researchers have implemented cryptographic algorithms as extern functions running in software switches (e.g., [58, 59]). As a result, these solutions cannot perform the required operations at Tbps switch line rates. Chen presented a new technique that leverages the table matching capability (which we discuss in Section 2.4) available on programmable switches to implement AES [60]. However, as we will show in Chapter 3, this table-based method does not scale for Finite Fields with a larger number of elements. In addition, it requires all packets to be recirculated multiple times in the switch (one recirculation per AES round), which affects latency and, most importantly, severely reduces the maximum achievable throughput. As for Network Coding, we note the work created by Gonçalves et al. [61] that has recently proposed an NC solution for network devices. However, their solution targeted a software switch and was prototyped only for Finite Fields of small size. They also only considered the coding and recoding operations, and not decoding. Table 2.2 compares all the work we have mentioned against our own.

2.6 Summary

In this chapter, we gave a primer on Finite Fields, defining their properties and focusing on the $GF(2^m)$ fields. We also presented some use cases of this construct in the areas of In-network Cryptography, Network Coding, and Forward Error Correction.

Afterward, we presented the concept of Programmable Networks, showcasing Software Defined Networks and the P4 language for programming network data plane devices. We also discussed the

	Hardware Implementation	In Network	All GF operations	10/100s Gbps	Tbps
Software implementations [8, 48–50]	X	X	✓	X	X
CPU implementations [51–53]	✓	X	✓	X	X
FPGA implementations [54–57]	✓	X	X	✓	X
AES in Switch [60]	✓	✓	X	✓	X
NC-Switch [61]	X	✓	X	✓	X
This work	✓	✓	✓	✓	✓

Table 2.2: Comparison of work in implementing Finite Field operations

current programmable network devices and their hardware architecture, mentioning not only the current state of the art, the PISA architecture, but also a new alternative proposed, Taurus.

Finally, we discussed current implementations of operations over Finite Fields in several types of hardware, like common CPUs, FPGAs, and switch ASICs.

3

Finite Field Operations

Contents

This thesis goal is to implement operations over Finite Fields in programmable network devices and evaluate their feasibility and usefulness. In this chapter, we look into the four most common operations executed over Finite Fields, addition, subtraction, multiplication, and division. We first start with addition and subtraction, which are trivial to perform. Afterward, we explore the multiplication operation and present the two main approaches used in practice. We then move to the division operation, also showcasing the two main approaches, which are similar to the ones for multiplication, and introducing a new operation, Finite Field inversion. Finally, we analyze both methods, and discuss their merits and their drawbacks.

3.1 Finite Field Addition and Subtraction

For the Finite Fields defined as $\mathbb{Z}_p$ with p being a prime value (explored in Section 2.1), addition and subtraction are similar to the basic arithmetic one is accustomed to. First, the rules defined for Fields only mention the addition operation and never explicitly mention subtraction. However, since every Field F is a Group and every Group element must have an inverse element with respect to addition that is also in the Field, we can define $a - b$ for any $a, b \in F$ as $a + (-b)$ where $-b$ is the inverse [11].

The only added complexity to common addition and subtraction operations is that, to get the actual result, one needs to apply the *modulo* operand with value p. This ensures that the resulting value is a member of the Finite Field and, most importantly, complies with the rules that define a Field. As an example, in the Field $\mathbb{Z}_7 = \{0, 1, 2, 3, 4, 5, 6\}$, $2 - 5 \equiv -3 \mod 7 \equiv 4 \mod 7$ and $6 + 5 \equiv 11 \mod 7 \equiv 4 \mod 7$.

When we exit the Fields with natural elements and focus on Finite Fields defined by polynomials ($GF(p^m)$), addition and subtraction are the common operations over polynomials, meaning that we add, or subtract, the coefficients of the polynomials that share the same degree. Recall, however, that the coefficients still need to remain in $GF(p)$, so the modulo operand must be applied. As a quick example over the Field $GF(7^5)$, $(5x^4 + 2x^3 + x + 5) + (3x^4 + 2x^2 + 5) = 8x^4 + 2x^3 + 2x^2 + x + 10$. Since 8 and 10 do not belong to $GF(7)$, we must apply the modulo 7. $8 \mod 7 \equiv 1 \mod 7$ and $10 \mod 7 \equiv 3 \mod 7$, so the final result would be $x^4 + 2x^3 + 2x^2 + x + 3$.

For the Finite Fields that are important for our work, $GF(2^m)$, the coefficients can only be 0 or 1. If we look at all possible combinations of adding and subtracting 0 and 1, modulo 2, the results are the same as an `XOR` operation. Table 3.1 showcases this operation. As such, adding or subtracting values in $GF(2^m)$ is straightforward. It is a simple bit-wise `XOR` [5] over the operands. As a quick example, in $GF(2^5)$, $(x^4 + x + 1) + (x^3 + x^2 + 1) = x^4 + x^3 + x^2 + x$.

The real challenge regarding operations over Finite Fields is thus the other operations, multiplication and division. In the following sections, we explore the two main approaches used in the literature to

$\oplus$	**0**	**1**
0	0	1
1	1	0

Table 3.1: XOR table

perform them.

3.2 Finite Field Multiplication

In order to perform Finite Field multiplication, one can choose to go in one of two ways, depending on the size of the Finite Field and the computational capabilities of the device where it will be implemented, among others. We first start with an approach that is more memory-intensive in Section 3.2.1, and then move on to the computationally-intensive approach in Section 3.2.2.

3.2.1 Memory Intensive Approach

´

The multiplication of two values a and b can be a complex operation to perform, even without considering Finite Fields, if, for example, one needs to multiply two extremely large values. However, there is a way to convert a multiplication operation into an addition if we recall the properties of the *logarithms*. For any base of a logarithm g, the sum of the logarithms of two values is the same as the logarithm of the product of those values. In other words, $\log_g a + \log_g b = \log_g ab$. If we then exponentiate g to this value, we get $a \times b$. For Finite Fields, we can also follow this idea, but the basis g of the logarithm cannot be any value, but rather the *generator* of said Finite Field (see Section 2.1). To summarize, we can get the product of any two elements of a Finite Field F with generator g by following the rule $a \times b = g^{\log_g a + \log_g b}$ [5].

At first glance, this seems to add complexity to the operation and does not help to simplify it. We need to compute the logarithm of a, the logarithm of b, and the value of g to the power of the result of the sum. However, since we are working on a Finite Field, and therefore we have a finite number of elements, all the possible values of the logarithm, and also of the exponentiation, which we call anti-logarithm, can be pre-computed beforehand and stored in the memory. This is the main advantage of this approach, as we are effectively trading one multiplication, for one addition and some accesses to a table stored in memory.

As a quick example, Figure 3.1 shows all the possible values of the logarithm (Figure 3.1a) and anti-logarithm (Figure 3.1b) pre-computed for the field $GF(2^8)$ using the generator $x + 1$. For calculating 10×25, we first look at the logarithm table (note that in the table, the values are in hexadecimal) and

Table of “Logarithm” Values																	
L(rs)		s															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	–	00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
r	7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

(a) Logarithm values

Table of “Exponential” Values																	
E(rs)		s															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
	1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
	2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
	3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
	4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
	5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
	6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
r	7	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
	8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
	9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
	a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
	b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
	c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
	d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
	e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
	f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

(b) Antilogarithm values

Figure 3.1: Multiplicative tables for field $GF(2^8)$, from [5]

extract the values, $0x1b$ and $0x71$, respectively. We add them together, resulting in $0x8c$, and finally, we get the result from the anti-logarithm table, $0xfa$, which is 250. As an important remark, the addition of the logarithm values is *not* a Finite Field addition (XOR), but rather a common arithmetic addition, modulo the last value of the Field. In this case, 255. This works because the powers of a generator of a Finite Field repeat after $2^n - 1$ iterations [5].

This is, therefore, a “memory-intensive” approach, as we can reuse the values stored in these tables for different computations. By decomposing the multiplication operation this way, we can perform it with a small number of table accesses (usually three). This solution can be easily implemented in all sorts of devices, including programmable switches using the P4 language (see Chapter 4 for more details), and its high performance is guaranteed: looking up values in tables is quick, especially for these types of devices, and can be done in parallel. To no surprise, some state-of-the-art solutions follow this approach, either in software for common CPUs [8] or in hardware [61, 62].

3.2.2 Computationally-Intensive Approach

There is another solution for multiplying two values in a Finite Field that does not involve using tables to store helpful values. Instead, it relies upon using number decomposition and manipulating the operands and is, therefore, a computationally-intensive approach. The most common solution that follows the number decomposition approach is the Russian Peasant Algorithm (RPA) algorithm [63].

RPA uses a halving and doubling method to multiply whole numbers, in our case, a and b. Doing this transforms the problem of multiplying two whole numbers into a much simpler one based on multiplication and division by 2. To illustrate how it works, assume we have two columns – one for the doubled numbers and the other for the halved numbers – and multiple rows – one for each iteration of the algorithm (where the first row denotes the multiplication problem we aim to solve). In the halving column, one needs to put the quotient of the division operation, disregarding the remainder (if any). In each iteration,

a is multiplied by 2, and b is divided by 2; this process is then repeated multiple times until b is equal to 1. As a final step, one must cross out all rows with an even number in the halving column. The result of the multiplication can be found by adding the remaining numbers in the column where the doubled numbers are kept.

Algorithm 3.1 showcases the pseudo-code of the RPA algorithm for any Finite Field $GF(2^n)$. For all the algorithms throughout this thesis, a_x represents the xth bit of the operand a, with 0 being the least significant bit. Since we are working with Finite Fields, there are two essential things to note. First, we know that at most after n iterations, the value of b will be 1, and so we can execute the algorithm using a *for* loop instead of checking for a condition at the beginning of a new iteration. This will significantly help the algorithm's implementation in pipelined architectures, as we will discuss in Chapter 4. Second, since we are multiplying the value of a by 2 at each iteration, and we will sum these values to get the result, there can be a case where $a \times 2$ is a number that does not belong to the Finite Field. As such, we need to perform one more operation on a that ensures the result will belong to said Field. This operation is an `XOR` with the irreducible polynomial P since, in this particular case, it is equivalent to executing the modulo operation [63].

Analyzing the algorithm, we start with a variable *product* which will store the result. In line 3, we check if the value of b is odd (the last bit is 1). If it is, we know the current value of a needs to be added to the final value, and we do it in line 4. Recall we are working with Finite Fields, so the addition is performed using an `XOR`. In line 5, we check if the most significant bit of a is 1. If it is, we know that once a is multiplied by 2, its value will no longer be part of the Field, and we need to use the irreducible polynomial (line 6). If it isn't, nothing more (except multiplying by 2) needs to be done (line 8). Note that multiplication by 2 is a simple `SHIFT` left operation by one bit. Finally, in line 9 we divide b by 2, a simple `SHIFT` right of one bit, and the iteration is complete. We go back to line 3 and repeat the process. After completing all the iterations, the variable *product* will have the result.

Algorithm 3.1: Russian Peasant Algorithm

```
   Data: $a, b \in GF(2^n),\ P$ as the irreducible polynomial
   Result: $product = a \times b$
1  $product \leftarrow 0$
2  for $i \leftarrow 0$ to $n - 1$ do
3      if $b_0 = 1$ then
4          $product \leftarrow product \oplus a$
5      if $a_{n-1} = 1$ then
6          $a \leftarrow (a << 1) \oplus P$
7      else
8          $a \leftarrow a << 1$
9      $b \leftarrow b >> 1$
```

	a	b	product
Iteration 1	00001010	00011001	00000000
Iteration 2	00010100	00001100	00001010
Iteration 3	00101000	00000110	00001010
Iteration 4	01010000	00000011	00001010
Iteration 5	10100000	00000001	01011010
Iteration 6	01011011	00000000	11111010
Iteration 7	10110110	00000000	11111010
Iteration 8	01110111	00000000	11111010
Final	11101110	00000000	11111010

Table 3.2: Multiplication of 10 by 25 using RPA over the $GF(2^8)$ Field

To more clearly illustrate this algorithm, Table 3.2 showcases the multiplication of 10 by 25 in the $GF(2^8)$ Field. The columns represent the values of the variables *before* the iteration begins. As such, in the first row, we have the original values of a ($10 = 0b1010$) and b ($25 = 0b11001$) and the value of the accumulator still at 0. The last row contains the values after the conclusion of the algorithm. In this case the result is $250 = 0b11111010$. We also point out the change in variable a from **Iteration 5** to **Iteration 6**, a result of the fact that we have `XOR` a with the irreducible polynomial.

3.3 Finite Field Division

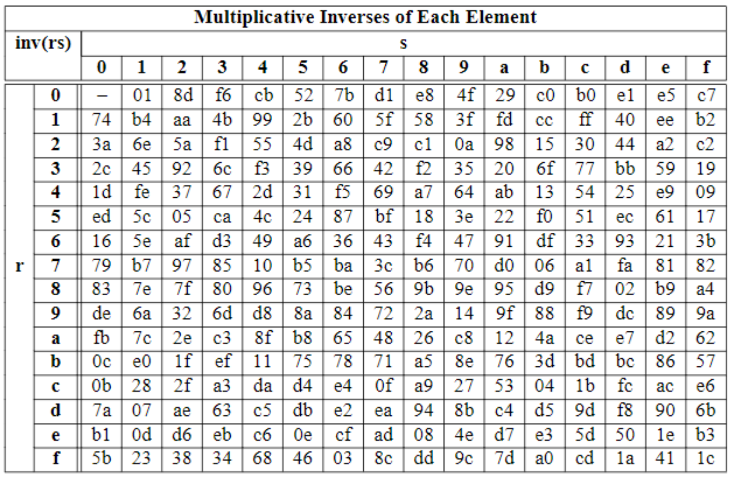
Similarly to the multiplication operation, there are several ways to perform Finite Field division, depending on the device's capabilities where the operation is to be implemented. We will start with the approach that relies on leveraging the device's memory in Section 3.3.1 before moving to the more computationally heavy approach in Section 3.3.2. In the latter, we present two options: a direct algorithm for computing the division and an algorithm for calculating the inverse of a value so that the multiplication algorithm can be applied afterward.

3.3.1 Memory Intensive Approach

In order to calculate the division of any two values, one can take advantage of the mathematical property of the inverse. If a value b has an inverse, than $\frac{a}{b} = a \times b^{-1}$ where b^{-1} represents said inverse. With Finite Fields, recall from Section 2.1, that all elements of a Finite Field F, except the 0 element (the F^* Field), must have an inverse, which is also an element of the Field.

With this property, we can quickly compute a division $\frac{a}{b}$ by leveraging the multiplication tables presented before. We need to look up the value for a and the value for b^{-1}. Fortunately, we can also pre-compute all the inverse values for all Field elements and store them in a new table. Figure 3.2 showcases the values of the inverses of all elements of the Field $GF(2^8)$, the same Field for which we have shown the multiplication tables in Figure 3.1.

As such, computing a division operation takes only one more table access than multiplication. For example, if one wants to calculate $\frac{a}{b}$, they lookup the inverse of b in the table and then lookup the logarithms of a and b^{-1}. Finally, they add them together and look up the result in the anti-logarithm table. This approach is also easily implemented in all types of devices, including programmable switches.



Multiplicative Inverses of Each Element																	
inv(rs)		s															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
r	0	–	01	8d	f6	cb	52	7b	d1	e8	4f	29	c0	b0	e1	e5	c7
	1	74	b4	aa	4b	99	2b	60	5f	58	3f	fd	cc	ff	40	ee	b2
	2	3a	6e	5a	f1	55	4d	a8	c9	c1	0a	98	15	30	44	a2	c2
	3	2c	45	92	6c	f3	39	66	42	f2	35	20	6f	77	bb	59	19
	4	1d	fe	37	67	2d	31	f5	69	a7	64	ab	13	54	25	e9	09
	5	ed	5c	05	ca	4c	24	87	bf	18	3e	22	f0	51	ec	61	17
	6	16	5e	af	d3	49	a6	36	43	f4	47	91	df	33	93	21	3b
	7	79	b7	97	85	10	b5	ba	3c	b6	70	d0	06	a1	fa	81	82
	8	83	7e	7f	80	96	73	be	56	9b	9e	95	d9	f7	02	b9	a4
	9	de	6a	32	6d	d8	8a	84	72	2a	14	9f	88	f9	dc	89	9a
	a	fb	7c	2e	c3	8f	b8	65	48	26	c8	12	4a	ce	e7	d2	62
	b	0c	e0	1f	ef	11	75	78	71	a5	8e	76	3d	bd	bc	86	57
	c	0b	28	2f	a3	da	d4	e4	0f	a9	27	53	04	1b	fc	ac	e6
	d	7a	07	ae	63	c5	db	e2	ea	94	8b	c4	d5	9d	f8	90	6b
	e	b1	0d	d6	eb	c6	0e	cf	ad	08	4e	d7	e3	5d	50	1e	b3
	f	5b	23	38	34	68	46	03	8c	dd	9c	7d	a0	cd	1a	41	1c

Figure 3.2: Inverse values for Field $GF(2^8)$, from [5]

3.3.2 Computationally Intensive Approach

The division operation using more computationally intensive methods can be executed in two different ways. One can leverage an algorithm that directly computes the result of the division of the two operands, or we can use an algorithm that first computes the inverse of the second operand so that we can apply RPA afterward.

A – EBD For the first option, we rely on the EBd algorithm [64], which is a derivate of Stein's algorithm to find the Greatest Common Divider (GCD) between two numbers (i.e., a and b). [65]. The Stein algorithm finds the GCD of a and b by iteratively performing some `SHIFT`s, in order to multiply and divide by 2, and `XOR`s, in order to perform addition and subtraction, according to the following rules: (1) if a and b are even, then $gcd(a, b) = 2 \times gcd(a/2, b/2)$; (2) if one of the operands is odd, then $gcd(a, b)$ is equal to $gcd(a/2, b)$ if b is the odd one, or $gcd(a, b/2)$ otherwise; and finally (3) if both are odd, then $gcd(a, b) = gcd(|a - b|/2, min(a, b))$. This algorithm can then easily be extended to perform Finite Field division.

The EBd algorithm for Finite Field division contains two additional helper variables *v* and *s*. The former is responsible for storing the result of the division and will also manipulate the a operand. In contrast, the latter is used to store the value of the irreducible polynomial *P* and will be used in the iterations of the algorithm to modify the b operand accordingly. There is also a variable δ, tracking the difference between the degrees of the two polynomials we are dividing. This is an important value to keep track of during the algorithm's execution. This algorithm finds a result after at most $2n-1$ iterations, where n is the number of bits of the Finite Field.

In each iteration, the EBd algorithm checks if b is odd and, if so, computes $b = b + s$ and $a = a + v$. When the degree of polynomial a is higher than that of b, the value of the variable δ will be less than 0. When that is the case, the algorithm also sets $s = b$ and $v = a$. This operation will always result in the difference of the degrees of the polynomials being symmetrical, so δ needs to switch signs.

At the end of each iteration, the b operand will always be divided by two (`SHIFT` right of one bit), as well as the a operand. However, the latter has an important detail we will see shortly. This instruction will also cause the degree of b to reduce by one and the one of a to stay the same, so the δ variable needs to be decremented.

Algorithm 3.2: EBd Algorithm

```
   Data: $a, b \in GF(2^n)$
   Result: $v = \frac{a}{b}$
1  $s \leftarrow P$                                  /* $P$ is the irreducible polynomial */
2  $v \leftarrow 0$
3  $\delta \leftarrow -1$
4  for $i \leftarrow 0$ to $2n-2$ do
5      if $b_0 = 1$ then
6          if $\delta < 0$ then
7              $(b, s) \leftarrow (b \oplus s, b)$
8              $(a, v) \leftarrow (a \oplus v, a)$
9              $\delta \leftarrow -\delta$
10         else
11             $b \leftarrow b \oplus s$
12             $a \leftarrow a \oplus v$
13     $b \leftarrow b >> 1$
14     $\delta \leftarrow \delta - 1$
15     $a \leftarrow (\frac{a}{2})_P$
```

Algorithm 3.2 presents the pseudo-code of the EBd algorithm we have just described. We draw special attention to lines 7 and 8 where the two assignments in each line must be performed "at the same time" to ensure the algorithm's correctness. In other words, in line 7, for example, b gets the value of $b \oplus s$, where the value of s comes from the previous iteration, but s must also get the value of b from before the assignment. One way to perform this in a common programming language is by using a

	a	b	delta	s	v
Iteration 1	11011111	00000111	-1	100011011	00000000
Iteration 2	11101110	10001110	0	00000111	11011111
Iteration 3	01111011	01000111	-1	00000111	11011111
...	—	—	—	—	—
Iteration 15	01110111	00000001	-1	00000001	10110100
Final	11100000	00000000	0	00000001	01110111

Table 3.3: Division of 223 by 7 using EBd over the $GF(2^8)$ Field

temporary variable that stores the first value, assigns the second value to the first variable, and then assigns the temporary value to the second variable.

We also point out the operation of $(a/2)_P$ in line 15, which is the division of a by 2, but taken modulo P, which does not seem trivial at first but can be performed using simple `SHIFT`s and `XOR`s following these rules [64]:

- $a_{n-1} \leftarrow a_0$.
- $a_k \leftarrow a_{k+1} \oplus (a_0 \cdot P_{k+1})$ for $0 \leq k \leq n-2$.

In other words, this operation performs a `RotR`, but then a bit in position k needs to be `XOR`ed with the least significant bit of a (a_0) if the irreducible polynomial P has the bit in the position $k+1$ set to 1. For example, in $GF(2^8)$ using the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$, if we want to execute this operation, we need to perform this `XOR` for bits a_0, a_2 and a_3.

We illustrate the algorithm via an example in Table 3.3, dividing 223 by 7 over the $GF(2^8)$ Field. We present the first three iterations of the algorithm, the last one and the final result. The result of the division is stored in the v variable, in this case $119 = 0b1110111$. Recall that the rows present the values of the variables before the iteration begins. For a concrete example of the instruction of line 15 mentioned previsously ($(a/2)_P$), one can look at the change of the a variable from **Iteration 2** to **Iteration 3**.

B – Inversion The second option for Finite Field division relies on an algorithm that finds the inverse of the b operand [66], represented by the x variable, which is heavily inspired by the extended Euclid's algorithm. Remember, however, that if we want to perform Finite Field division, we still need to execute the RPA after this algorithm. The original Euclidian algorithm computes $gcd(x, y)$, leveraging the fact that the GCD between two values does not change if the larger value is replaced by the remainder of the division between itself and the smaller value. By applying this concept until $gcd(z, z)$ is obtained, we have $z = gcd(x, y)$.

The extended Euclid's algorithm is also capable of outputting the values u and v that satisfy $gcd(x, y) = u \times x + v \times y$, as in each iteration i, it finds the quotient q_i and the remainder r_i of the division between x and y , and performs the following operations $u_i = u_{i-2} - q_{i-1}u_{i-1}$ and $v_i = v_{i-2} - q_{i-1}v_{i-1}$. In order to apply this to calculating the inverse of an element in a Finite Field, we note that the GCD between any polynomial x and the irreducible polynomial P of the Field is one. As such, if we replace y with P, we obtain $1 = u \times x + v \times P \iff 1 \equiv u \times x \; mod \; P \iff x^{-1} \equiv u \; mod \; P$.

The algorithm of [66] also relies on multiple iterations over the operands, using only simple `SHIFT`s and `XOR`s. This algorithm finds the inverse of a value after, at most, $2n$ iterations, with n being the number of bits of the Finite Field. Besides the extra values u and v mentioned previously, this algorithm also uses an additional helper variable s, that stores the irreducible polynomial P and manipulates the operand x, as well as a variable δ that tracks the degree of the polynomial u.

Algorithm 3.3: Inverse Algorithm

```
Data: $x \in GF(2^n)$
Result: $u = x^{-1}$
1  $s \leftarrow P$                                  /* $P$ is the irreducible polynomial */
2  $v \leftarrow 0$
3  $u \leftarrow 1$
4  $\delta \leftarrow 0$
5  for $i \leftarrow 0$ to $2n - 1$ do
6      if $x_n = 0$ then
7          $x \leftarrow x << 1$
8          $u \leftarrow u << 1$
9          $\delta \leftarrow \delta + 1$
10     else
11         if $s_n = 1$ then
12             $s \leftarrow s \oplus x$
13             $v \leftarrow v \oplus u$
14         $s \leftarrow s << 1$
15         if $\delta = 0$ then
16             $(x, s) \leftarrow (s, x)$
17             $(u, v) \leftarrow (v << 1, u)$
18             $\delta \leftarrow 1$
19         else
20             $u \leftarrow u >> 1$
21             $\delta \leftarrow \delta - 1$
```

Algorithm 3.3 presents the pseudo-code. Just like with Algorithm 3.2, in lines 16 and 17, the two assignments of each line need to be performed “at the same time” to assure the correctness of the algorithm. Note also the if condition in line 6, that assures that the value of x is only multiplied by 2 until its value remains within the bonds of the Field. When that does not happen, and $x_n = 1$, we see if s_n is also 1. If it is, we `XOR` the two (line 12), before multiplying s by 2 and manipulating the variables according to the $delta$ variable in the next lines.

To conclude this section, we showcase in Table 3.4 the inversion of 223 over the $GF(2^8)$ Field. Like

	x	s	v	delta	u
Iteration 1	011011111	100011011	00000000	0	00000001
Iteration 2	110111110	100011011	00000000	1	00000010
Iteration 3	110111110	101001010	00000010	0	00000001
...	—	—	—	—	—
Iteration 15	100000000	110000000	110100110	1	11010110
Final	100000000	100000000	101110000	0	01101011

Table 3.4: Inversion of 223 over the $GF(2^8)$ Field

for the EBd algorithm, we present the first three and the last iterations, as well as the final result of all the variables. The result of the inversion is stored in the u variable, which, in this case, is $107 = 0b1101011$.

3.4 Analysis of Both Approaches

In this section, we briefly analyze the approaches we have described so far. Then, we explore the characteristics, trade-offs, and where should each one of them be employed. This analysis will also help to understand some of the challenges we faced when implementing them in a programmable switch, which will be described in Chapter 4.

3.4.1 Takeaways from the Memory Intensive approach

The simplicity of this approach is its main advantage. All the calculations can be done beforehand in order to populate the logarithm, anti-logarithm, and inverse tables. As such, when an operation is to be performed, it is just a case of looking up values in these tables. This sort of table lookup based approach is where a programmable switch typically excels. This approach can be parallelized with many packets looking up the values in the tables per stage. However, dependencies have to be maintained. If one multiplication is to be performed, for instance, you can only look up the anti-log value after completing the addition of the values taken from the logarithm table. So, in a switch these operations have to occur in multiple stages. For a division, you have to look up the inverse in the respective table beforehand.

However, this technique has a significant drawback in terms of scalability. The size of the tables that store all the possible values grows *exponentially* with the size of the Field. As an example, if we perform computations in the Field $GF(2^8)$, we have 2^8 possible values, and each value has one byte; so, this solution requires only 256B of stateful memory per table, a residual amount considering the memory capacity of modern devices. However, operations are performed on a per-byte basis. If, for example, operations are performed on the larger $GF(2^{16})$ Finite Field, we now have $2^{16} = 65536$ possible values and two bytes per value; so, the table size increases to 128KB, which is still feasible. With this Field size, we can perform two-byte operations at a time. If we want to perform computations with 128-bit words, or 16 bytes, it would require an astronomical 10^{39} bytes of memory. And operations over such Finite Fields are required, as we have described in Section 2.2.

To address this scalability issue, the standard solution is to operate on smaller fields ($GF(2^8)$ or $GF(2^{16})$), trading off the required number of table lookups. This, however, is rarely enough. As a concrete example, coding a 1500-byte payload for Network Coding using $GF(2^8)$ requires $1500 * 3 = 4500$ table lookups, which is a great number but still feasible for a typical CPU. Unfortunately, it is not feasible for a programmable switch, as we will see in Chapter 5. Worse still, in cryptography, operating in smaller Fields is impossible without considerably reducing or even breaking the security properties of the underlying cryptographic algorithms [67,68]. In summary, this approach is only feasible if we operate on very small packets/payloads using small Finite Field sizes.

3.4.2 Takeaways from the Computationally-Intensive approach

The algorithms presented for performing multiplication, division, and inversion all follow the same pattern. A certain number of iterations is performed, and each iteration manipulates the operands using simple instructions, like `SHIFT`s and `XOR`s, which any device can perform. Recall that a network switch using P4 cannot multiply by any number larger than 2, for example. The amount of iterations to execute is a function of the number of bits of the Finite Field. For any field $GF(2^n)$:

1. RPA needs n iterations;
2. EBd needs $2n - 1$ iterations;
3. The inverse algorithm needs $2n$ iterations.

The main advantages of this algorithm are the simplicity of the instructions performed in each iteration and the scalability. In order to work with larger Field sizes, one only needs to do more iterations. Better yet, the increase in the number of iterations to be performed is linear to the number of bits. For a standard CPU, this is perfect, as it can perform cycles efficiently, and since the instructions are simple, each iteration is quick to execute.

However, as the pipeline architecture of network switches includes a limited number of stages and packet recirculation affects throughput, performing iterations is not so trivial for these devices. P4, for instance, does not have any loop primitives, recognizing that practical challenge. The only option is to physically unroll the iterations in order to be able to implement all of these algorithms in a P4-programmable data plane (we explore this further in Chapter 4). Another drawback is that this approach does not require the use of tables, so it does not take advantage of the efficiency of the device's MATs.

Parallelization of these algorithms is also a drawback. The manipulations performed in each iteration mean that there are data dependencies, making parallelizing the iterations practically impossible. The only solution typically comes from selectively performing some instructions for the next iteration while

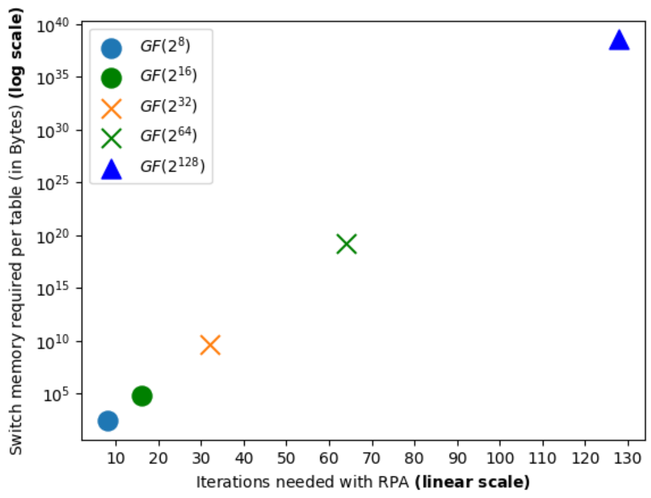


Figure 3.3: Memory cost vs iteration cost for various finite fields

the previous iteration is still concluding. As an example, in Algorithm 3.2, one can execute the *ifs* in lines 5 and 6, as well as the instructions in line 7 of iteration $i + 1$ while the operation of line 15 is still being performed for iteration i, and if the instructions of lines 13 and 14 are already completed.

3.5 Summary

In summary, table-based approaches leverage the properties of the logarithms and the fact that all the values can be pre-computed beforehand. This property, of course, means that this approach has a memory cost that can quickly become insurmountable once the size of the Finite Field starts to grow. However, this approach is still the most used in state-of-the-art use cases that do not need operations in large Finite Fields and that can work with small packets or payloads.

Computational-heavy approaches leverage number decomposition and perform several iterations to get the final result. Each iteration executes some simple instructions that manipulate the operands. The cost of this approach is the number of required computational elements, cycles, and iterations. But the scalability is better. The cost grows *linearly* with the size of the field, whereas the memory multiplication tables *scale exponentially*.

We close this chapter by illustrating this point in Figure 3.3. In the x-axis, we plot the number of required iterations for executing RPA on various Finite Field sizes (Note: linear scale). The y-axis represents the memory needed in bytes for those same Fields (Note: *logarithmic* scale). As one can see, the scalability is much poorer for the memory-based approach. For the largest field, $GF(2^{128})$, only 128 iterations are needed in RPA, but 10^{39} bytes per table are required to store all possible values.

4 Implementation

Contents

In this chapter, we present the challenges of working with a switch ASIC. Afterward, we discuss and present the implementation of the various Finite Field operations algorithms we introduced in the previous chapter. We do not implement addition and subtraction because it is a simple `XOR` operation that is natively present in most switches. We start by implementing both approaches for multiplication and division in a programmable switch, Intel Tofino, using the P4 language. Next, we implement the number decomposition-based algorithms in Taurus, using an FPGA programming language, Spatial. Finally, we discuss the challenges we had to overcome during this process, supported by the actual code we developed.

4.1 Challenges of programming a switch ASIC

PISA architectures have high speed requirements for processing packets (nowadays around the scale of several Tbps), which translates in several limitations present in their programming language:

1. There is a fixed number of physical stages, and each stage has a fixed set of resources.

2. The PHV has a size limit, generally around 512 bytes. Headers, including metadata, cannot exceed this size, and you cannot work on the actual payload of the packet.

3. The number and complexity of the actions that can be executed on a single stage are severely limited. Usually, each stage can only perform one simple operation, like a `SHIFT`, an addition, or an `XOR`. Multiplication or floating point operations are not possible, for example.

4. Since this is a pipelined architecture, no standard loops are allowed, and the iterations must be unrolled so that each stage performs one or more of them. If one wants a lot of iterations or to repeat certain operations, the packets can be recirculated inside the switch, but the throughput suffers and decreases.

5. These ASICs are equipped with limited amounts of high-speed memory. Specifically, TCAM, suitable for longest-prefix matching in routing tables, and SRAM, used for exact matching and to persist state across packets (e.g., using stateful register arrays).

6. In addition to the limited memory available, its access is also very restricted. Registers in one stage cannot be accessed at different stages, and memory accesses are usually limited to a single read, write, or read-write operation per stage.

4.2 Finite Field Operations in a Programmable Switch

We start by implementing the aforementioned approaches in a P4 programmable switch, Intel Tofino [47]. This switch ASIC is, nowadays, common in modern high scale data centers, and its architecture is similar to other commercial network switches. As such, these implementations will showcase the capabilities of these devices for performing Finite Field operations in-network.

We defined a new header as illustrated in Figure 4.1. The first Field, op, serves as an identifier for the operation we want to perform (multiplication, division or inversion) and the approach we want to execute (memory intensive or computationally intensive). This Field could occupy less than eight bits; however, P4 demands that the total size of the header must be a multiple of eight bits. This has the advantage of enabling more operations to be added in the future. The following two fields store the operands a and b and, finally, the $result$ Field will store the result of the operation. As a note, for the inverse algorithm we do not need both operands, so we ignore one of them. In this particular case, we ignored the a operand.

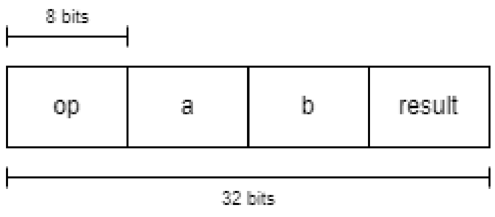


Figure 4.1: Header for Finite Field operations

4.2.1 Memory Intensive Approach

For the algorithms we described in Sections 3.2.1 and 3.3.1 we pre-compute all the necessary values (logarithm, anti-logarithm, and inverse) and load them in the MATs of the switch. One challenge we faced was that each logic MAT can only match with *one* operand. So, if we want to perform $a \times b$, for example, we need one MAT to find the logarithm of a, another for the logarithm of b and the final one for the anti-logarithm of the sum, each one with all the possible values of the Field. Of course, if the division is the required operation, another table is needed for finding the inverse of the b operand. For simplicity, we will describe the implementation of this approach for the Finite Field $GF(2^8)$, so all the values will have eight bits.

Listing 4.1 showcases the *apply* block in the Ingress pipeline, where the logic for calculating the desired result is applied. We start by matching the a operand and extracting its logarithm value using the *table_log_a* table. We then use the op header to decide whether we need to look up the value of the inverse of b (*table_inverse* table) for a division operation or not, before looking up its logarithm value, in the *table_log_b* table. After having both values, we sum them together in the *sum_vals* action, and we check if this value is greater or equal to 255. If it is, we subtract 255 using the *sub_max* action, before looking up the final value in the *table_antilog* table. The action of this table automatically loads the value to the *result* field.

Listing 4.1: P4 apply block for table based operations

```
#define MAX_SIZE 255
#define BIT_MASK 256

table_log_a.apply();
log_a = meta.tmp_log_value;
if (hdr.ff_calc.op & 1 != 0) {
    table_inverse.apply();
} else {
    meta.b = (bit<32>) hdr.ff_calc.b;
}
table_log_b.apply();
log_b = meta.tmp_log_value;
sum_vals(log_a, log_b);

if (meta.tmp_sum_value == MAX_SIZE ||
  meta.tmp_sum_value & BIT_MASK != 0) {
    sub_max(meta.tmp_sum_value);
}
table_antilog.apply();
```

This code can easily be modified to work for any Finite Field size whose tables fit inside the switch's memory. For $GF(2^{16})$, for example, one just needs to change the size of the header fields a, b, and $result$ to have 16 bits and repopulate the tables with all the 65536 possible values. One also also needs to change the values of the MAX_SIZE and BIT_MASK variables to 65535 and 65536, respectively.

4.2.2 Computationally Intensive Approach

Implementing the computationally intensive algorithms in a P4 programmable switch proved, as expected, more challenging than the memory-intensive approach we showcased previously. This challenge is mainly due to these devices' pipelined architecture. For instance, there are no available primitives in the language to implement loops, as this may lead to recirculations that hurt the achievable throughput. As such, we are required to unroll the iterations and explicitly write the source code for all of them.

For all the algorithms we first started with trying to implement operations over the Finite Field $GF(2^8)$, so as to have a direct comparison with the memory intensive approach. When, due to the switch's constraints, that was not possible, we reduced the size until we had a program that could run in the device. We will further explore this in Chapter 5.

Starting with the multiplication algorithm, RPA, since the Field has values with eight bits in our example, we know we need to unroll the cycle of the RPA algorithm (Algorithm 3.1) and repeat it eight times.

Listing 4.2: One iteration of RPA in P4

```
low_bit_f = meta.b & 0x1;
if (low_bit_f != 0)
    action_ff_mult(true);
else {
    action_ff_mult(false);
}
if (meta.high_bit_f != 0) {
    meta.a = meta.a ^ IRRED_POLY;
}
```

Listing 4.3: action_ff_mult action

```
#define HIGH_BIT_MASK 128

action action_ff_mult(bool low_bit) {
    if (low_bit) {
        meta.result = meta.result ^ meta.a;
    }
    meta.high_bit_f = meta.a & HIGH_BIT_MASK;
    meta.a = meta.a << 1;
    meta.b = meta.b >> 1;

}
```

Listings 4.2 and 4.3 showcase one iteration of RPA in the *apply* block of the P4 program and the action, or function, that we use to execute some of the instructions, respectively. The low_bit_f flag is set to 1 when the *b* operand is odd, in line 1. This value is then used in the $action\_ff\_mult$ action to decide whether or not to `XOR` *a* with the accumulator. This action is then also responsible for setting a flag, $high\_bit\_f$, in line 7, in case *a* needs to be `XOR`ed with the irreducible polynomial (line 5 of the algorithm), defined by the $IRRED\_POLY$ variable. This is done by looking at the most significant bit of *a*, which for this Finite Field, is the eighth bit. If the bit is 1, we know that once we multiply *a* by 2, it will be a value outside the Finite Field. This `XOR` (line 6 of the algorithm) happens in the final instruction of the loop, in lines 7 and 8. Afterwards, the action multiplies *a* by 2 and divides *b* by 2. These are simple `SHIFT` operations by one bit.

Just like with the previous algorithm, Listings 4.4 and 4.5 showcase the P4 code executed in a single iteration of the division algorithm presented in Algorithm 3.2. The former is the code in the *apply* block, and the latter the actions performed. For $GF(2^3)$, the maximum size we were able to work with, we had to repeat the code in the first listing five times. This code is straightforward up until line 19. We use the b_pair_flag to check if b is odd or not. Using the value from the $meta.delta$ variable, we decide which operations to execute. Either lines 4 to 9 (corresponding to lines 7 to 9 of the algorithm) or lines 11 and 12 (same lines in the algorithm). Finally, we execute lines 16 and 17, that decrement the $delta$ variable and divide b by 2.

We draw particular attention to the $action\_divide\_val\_a$ action that sets the a variable to the result of $a \oplus v$ and the v variable to the value of a and $action\_divide\_val\_b$ action that sets the b variable to the result of $b \oplus s$ and the s variable to the value of b (lines 7 and 8 of Algorithm 3.2). As we discussed in Section 3.3.2, they need to use an auxiliary variable in order to operate on the correct values.

From line 19 until the end of the iteration, we execute the final instruction of the algorithm, $(a/2)_P$. We described how to perform it in Section 3.3.2 and compared it to the Rotate Right (`ROR`) instruction. Since P4 does not have a `ROR` primitive, because the `SHIFT` primitive consumes all the computational resources of a single stage, there is no direct way to perform it. As such, we need to execute the rotate operation and change the bits accordingly and explicitly. In line 19, we save the a_0 bit, which is then placed in the most significant bit (a_2) in line 29. In line 20, we save the bits a_1a_0 to help us compute what the new value of a_0 should be. Recall from 3.3.2 that $a_0 \leftarrow a_1 \oplus (a_0 \cdot P_1)$. The irreducible polynomial P for $GF(2^3)$ is $x^3 + x + 1$ and so the bit P_1 is 1. As such, in the if section, we see if a_0 and a_1 have different values. If they do, the new bit will be 1, else it will be 0. Finally, we save the value of the a_2 bit in line 21, since that will be the new value of a_1 (the bit $P_2 = 0$, so the `XOR` is irrelevant here).

Listing 4.4: One iteration of the division algorithm in P4

```
1      b_pair_flag = meta.second_operand.b & 0x1;
2      if(b_pair_flag != 0) {
3          if (meta.delta < 0) {
```

```
            aux_a = meta.first_operand.a;
            aux_b = meta.second_operand.b;

            action_divide_val_a(meta.first_operand.a, meta.first_operand.v);
            action_divide_val_b(meta.second_operand.b, meta.second_operand.P);
            meta.delta = -meta.delta;
        } else {
            action_divide_xor_a(meta.first_operand.a, meta.first_operand.v);
            action_divide_xor_b(meta.second_operand.b, meta.second_operand.P);
        }
    }

    meta.delta = meta.delta - 1;
    meta.second_operand.b = meta.second_operand.b >> 1;

    a_low = meta.first_operand.a[0:0];
    a_aux = meta.first_operand.a[1:0];
    a_high = meta.first_operand.a[2:2];

    if(a_aux == 2 || a_aux == 1) {
        a_help = 1;
    } else {
        a_help = 0;
    }

    meta.first_operand.a[2:2] = a_low;
    meta.first_operand.a[0:0] = a_help;
    meta.first_operand.a[1:1] = a_high;
```

Listing 4.5: Actions for the division operation

```
     action action_divide_val_a(inout bit<8> a, inout bit<8> v) {
        a = a ^ v;
        v = aux_a;
    }

    action action_divide_val_b(inout bit<8> b, inout bit<8> P) {
        b = b ^ P;
        P = aux_b;
    }

    action action_divide_xor_a(inout bit<8> a, bit<8> v) {
        a = a ^ v;
    }

    action action_divide_xor_b(inout bit<8> b, bit<8> P) {
        b = b ^ P;
    }
```

Finally, we implemented the inverse algorithm (Algorithm 3.3) for the Finite Field $GF(2^4)$, as it was the maximum size Tofino could operate on. Although inversion alone can be used for some use cases, this operation is mostly used as a step toward computing the division. For this, after the inversion is calculated, an algorithm like RPA must be used.

Listings 4.6 and 4.7 present, just like the previous algorithms, the P4 code of a single iteration, more specifically, the *apply* block code and the actions, respectively. For the Field $GF(2^4)$, this code had to be repeated eight times.

The main challenge of this algorithm was working with the amount of auxiliary variables needed, as well as the number of manipulations per iteration. As one can see, we have four auxiliary variables ($meta.second.operand.s$, $meta.second.operand.v$, $meta.second.operand.u$ and $meta.delta$), which all had to be stored, and ten different possible manipulations of these values. This translated in a high number of actions and instructions. Once again, we note actions $action\_swap\_a\_s$ and actions $action\_swap\_shift\_u$ that need auxiliary variables in order to assign the correct values to s and v respectively.

Listing 4.6: One iteration of the inverse algorithm in P4

```
#define MSB 8

b_msb_flag = meta.second_operand.b & MSB;
s_msb_flag = meta.second_operand.s & MSB;
if(b_msb_flag == 0) {
    action_shift_b_u(meta.second_operand.b,
        meta.second_operand.u);
} else {
    if(s_msb_flag != 0) {
        action_xor_s_v(
                meta.second_operand.s,
                meta.second_operand.v,
                meta.second_operand.b,
                meta.second_operand.u);
        }
        meta.second_operand.s =
            meta.second_operand.s << 1;
        if (meta.delta == 0) {
            aux_b = meta.second_operand.b;
            aux_u = meta.second_operand.u;
            action_swap_b_s(
                meta.second_operand.b,
                meta.second_operand.s);
            action_swap_shift_u(
                meta.second_operand.u,
                meta.second_operand.v);
            meta.delta = 1;
        } else {
            meta.second_operand.u =
                meta.second_operand.u >> 1;
            meta.delta = meta.delta - 1;
        }
    }
```

Listing 4.7: Actions of the Inversion algorithm

```
    action action_shift_a_u(inout bit<8> a, inout bit<8> u ) {
        a = a << 1;
        u = u << 1;
        meta.delta = meta.delta + 1;
    }

    action action_xor_s_v(inout bit<8> s, inout bit<8> v, bit<8> a, bit<8> u) {
        s = s ^ a;
        v = v ^ u;
    }

    action action_swap_a_s(inout bit<8> a, inout bit<8> s) {
        a = s;
        s = aux_a;
    }

    action action_swap_shift_u(inout bit<8> u, inout bit<8> v) {
        u = v << 1;
        v = aux_u;
    }
```

4.3 Finite Field Operations in Taurus

In the next chapter we will demonstrate how the programatic restrictions of a switch ASIC significantly limit the Finite Field sizes that can run at line rate. This led us to explore a more recent switch data plane architecture, Taurus [4], with a more expressive programming model but also able to process packets at Tbps speeds, already described in Section 2.4.2. Since the memory constraints of both switches are similar, and because we know of the scalablity limitations of this approach, that precludes its use for larger Field sizes (Section 3.4.1), we opted to focus only on the number decomposition algorithms.

The MapReduce block of Taurus is responsible for performing the actual computations and is simulated with an FPGA. To program this FPGA, we leverage the Spatial language described in Section 2.3.3. Although Spatial provides the control structures we previously mentioned in that Section, we *cannot* use most of them, as they would preclude line rate processing at Taurus' clock speeds (1GHz). We leverage, however, a slightly modified version of the language used by the authors of the architecture [4].

Keeping the order from Section 4.2.2, we start with the code for RPA (Algorithm 3.1). Listing 4.8 showcases the implementation of one iteration of the algorithm.

The operands and variables are stored inside FIFO queues such that when an iteration begins, the program dequeues the values from the respective FIFOs, operates on them, and finally stores the new results in a new FIFO. The reason we do not use the same FIFO for all iterations is that Spatial, as of now, does not support multiple write operations to the same queue. As such, each iteration needs to write to a new FIFO. It is by using these FIFOs, as well, that we assure the correctness of the algorithm since the iterations cannot be executed in parallel, and we leverage the pipelining of the architecture.

The code follows the algorithm precisely. Lines 8 to 10 dequeue the results from the previous iteration, and in line 12, we use a mutex, which acts like an *if-else*. In this concrete case, if the least-significant bit of b is 1, then we enqueue the `XOR` between a and $result$; else, we only need to enqueue the current value of $result$. These operations correspond to lines 3 and 4 of Algorithm 3.1. Lines 15 and 16 check whether or not the most significant bit of a is 0 or 1 since when it is 1, we need to `XOR` a with the irreducible polynomial after multiplying it by 2 (lines 5 to 8 of the algorithm). This is also accomplished using the mux construct. Finally, line 17 enqueues the new value of b, which is a simple division by 2.

Considering again a Finite Field $GF(2^8)$, this code block has to be copied eight times, and we need another eight sets of FIFOs for a, b, and $result$. We could easily work with larger Field sizes by copying the block as many times as necessary and allocating more FIFOs. With Taurus, the computational constraints are not as tight as with P4 switches, so we can easily work with larger Finite Fields. We will explore this further in Chapter 5.

Listing 4.8: Pipe block with one iteration of RPA

```
val result_stage0 = FIFO[T](4)
val a_stage0 = FIFO[T](4)
val b_stage0 = FIFO[T](4)

val mask_a = 0x1.to[T] << 7

Pipe {
    val a = a_stage0.deq()
    val b = b_stage0.deq()
    val result = result_stage0.deq()

    result_stage1.enq(mux(b.bit(0) == 1, result ^ a, result))

    val flag = (a & mask_a) == mask_a
    a_stage1.enq(mux(flag, (a << 1) ^ irred, a << 1))
    b_stage1.enq(b >> 1)
}
```

Next, for Finite Field division, we implemented Algorithm 3.2, EBd. As previously, Listing 4.9 presents the Pipe blocks for one iteration. Like with the P4 version, the main challenge we faced was the $(a/2)_P$ operation.

All the code of the first Pipe block is a straightforward implementation of the Algorithm, where each new value of the variables is computed using the mux construct. Since the algorithm has nested if sections (lines 5 and 6 in Algorithm 3.2), we had to place mutexes inside mutexes to compute the new values correctly. In line 16, the last line of the Pipe block, we load to the FIFO a flag representing if the least significant bit of a is 1. That is important for the $(a/2)_P$ operation (line 15 of the algorithm).

We note that this code is for operating with the Finite Field $GF(2^8)$, which means that we work with the irreducible polynomial $P = x^8 + x^4 + x^3 + x + 1$. In the second Pipe block, in line 27, we execute a Rotate Right (ROR) operation by one bit. Spatial does not have a native Rotate operation, so we execute it by shifting the value to the right by one bit and then placing the least significant bit in the most significant position using a SHIFT to the left by seven bits and an OR. We know we must only manipulate bits a_0, a_2, and a_3 and keep the others unchanged. As such, we need three Pipe blocks to operate on each of the bits and correctly manipulate the value of a. The Pipe block that performs the ROR operation also manipulates the a_0 bit in lines 29 to 31. The final two Pipe blocks manipulate a_2 and a_3, respectively. However, these operations must only apply if the least significant bit of a is 1. Therefore, we use the previously mentioned flag in the mutex of the final instruction of these blocks.

In order to actually perform the operation, we XOR the bit with 1 and store it in a variable. We then use a mask that is composed of only 0s, except the bit at the required position, which is set to 1. By NOTing this mask and doing an AND with the operand, we get the same value but with the bit at the correct position set to 0. Finally, the OR instruction guarantees that the bit will get the value stored in the variable.

Listing 4.9: Pipe blocks with one iteration of EBd

```
1  Pipe {
2      val a: U8 = a_reg_0_3.deq()
3      val b: U9 = b_reg_0.deq()
4      val s: U9 = s_reg_0.deq()
5      val delta: Int = delta_reg_0.deq()
6      val result: U8 = result_0.deq()
7
8      a_reg_1_0.enq(mux(b.bit(0) != 1, a, a ^ result))
9      b_reg_1.enq(mux(b.bit(0) != 1, b >> 1, (b ^ s) >> 1))
10     s_reg_1.enq(mux(b.bit(0) != 1, s,
11                     mux(delta < 0, b, s)))
12     delta_reg_1.enq(mux(b.bit(0) != 1, delta - 1,
13                         mux(delta < 0, - delta - 1, delta - 1)))
14     result_1.enq(mux(b.bit(0) != 1, result,
15                     mux(delta < 0, a, result)))
16     if_swap_reg_1_0.enq(a.bit(0) == 1)
17 }
18
19 // For GF(2^8), we need to do the extra XOR for bits 0, 2 and 3
20 Pipe {
21     val a: U8 = a_reg_1_0.deq()
```

```

    val if_swap = if_swap_reg_1_0.deq()
    if_swap_reg_1_1.enq(if_swap)

    // Rotate right
    val new_a_0 = ((a >> 1) | (a << FFSize - 1)).as[U8]

    val bit0 = new_a_0.bit(0).as[U8] ^ 1
    val mask0 = 0x1.to[U8]
    val new_a_1 = ((new_a_0 & ~mask0) | (bit0)).as[U8]

    a_reg_1_1.enq(mux(if_swap, new_a_1, new_a_0))
}

Pipe {
    val a: U8 = a_reg_1_1.deq()

    val if_swap = if_swap_reg_1_1.deq()
    if_swap_reg_1_2.enq(if_swap)

    val bit2 = (a & mask_a_2) ^ 1.to[U8] << 2
    val mask2 = 0x1.to[U8] << 2
    val new_a_2 = ((a & ~mask2.as[U8]) | bit2.as[U8]).as[U8]

    a_reg_1_2.enq(mux(if_swap, new_a_2, a))
}

Pipe {
    val a: U8 = a_reg_1_2.deq()

    val if_swap = if_swap_reg_1_2.deq()

    val bit3 = (a & mask_a_3) ^ 1.to[U8] << 3
    val mask3 = 0x1.to[U8] << 3
    val new_a_3 = ((a & ~mask3.as[U8]) | bit3.as[U8]).as[U8]

    a_reg_1_3.enq(mux(if_swap, new_a_3, a))
}
```

The remaining algorithm is Finite Field inversion (Algorithm 3.3). Listing 4.10 presents one iteration of the said algorithm. Our implementation computes the inversion for the elements of the Finite Field $GF(2^8)$. Since for a Field of type $GF(2^n)$ a solution is found after $2n$ iterations, in this case ($n = 8$) the code of the listing needs to be copied 16 times. The code is a direct implementation of the algorithm, just like with the P4 version. We note, once again, the usage of mutexes inside mutexes to reflect the several nested decisions that the algorithm requires (lines 6, 10, 11, 15 and 19 of Algorithm 3.3) and ensure the correctness of our implementation.

Listing 4.10: Pipe block with one iteration of the Inversion Algorithm

```
Pipe {
      val a: U9 = a_reg_0.deq()
      val s: U9 = s_reg_0.deq()
      val v: U9 = v_reg_0.deq()
      val delta: Int = delta_reg_0.deq()
      val result: U9 = result_0.deq()

      // When true, the most significant bit of a is NOT 1
      val flag_a = (a & mask_msb) != mask_msb
      // When true, the most significant bit of s is 1
      val flag_s = (s & mask_msb) == mask_msb

      a_reg_1.enq(mux(flag_a, a << 1,
                    mux(delta == 0, s << 1, a)))
      s_reg_1.enq(mux(flag_a, s,
                    mux(flag_s,
                        mux(delta == 0, a, (s ^ a) << 1),
                        mux(delta == 0, a, s << 1))))
      v_reg_1.enq(mux(flag_a, v,
                    mux(delta == 0, result,
                        mux(flag_s, result ^ v, v))))
      delta_reg_1.enq(mux(flag_a, delta + 1,
                        mux(delta == 0, 1, delta - 1)))
      result_1.enq(mux(flag_a, result,
                    mux(delta == 0,
                        mux(flag_s, (result ^ v) << 1, v << 1), result >> 1)))
}
```

4.4 Summary

In this chapter, we presented our various solutions to run Finite Field operations in programmable switches. The challenge was to be able to design and implement equivalent versions of the original algorithms in the restricted computational model of these data planes. First, we started with solutions for modern programmable switches that leverage Intel's Tofino architecture and the P4 language, showcasing both the memory intensive and the computationally-intensive approaches. Afterward, we presented solutions for the computationally intensive approach algorithms in Taurus, a new data plane architecture, using the Spatial language.

5

Evaluation

Contents

In this chapter, we evaluate the implementations we have showcased previously. To guide the evaluation process, we try to answer these six questions:

1. Are the algorithms correctly implemented in both versions, P4 and Spatial?
2. Can we execute the operations at line rate?
3. Can we execute the operations with a satisfiable Finite Field size for both architectures?
4. How many resources are used with each approach and algorithm?
5. Can the switch perform other actions while also executing Finite Field operations?
6. How many multiplication operations can we execute inside a single packet?

5.1 Experimental Setup

Since we are working with two different architectures, an Intel Tofino switch that is a representation of a modern switch, and Taurus, a recently proposed data plane architecture, we have two different but similar environments we used to extract our results and simulate our solutions. For the evaluation of the Tofino switch, we used an Ubuntu 20.04 machine with 8GB of memory and a 2-core $x86\_64$ Intel CPU. We use P4-Software Development Environment (P4-SDE) version 9.7 as our simulation software and P4 Insight (P4i) version 9.7 to extract the resources and other metrics regarding our programs. For Taurus, the same machine runs Ubuntu 16.04. As for simulating the switch, we use the simulation software provided by [4]. Because Taurus is based on the Plasticine CGRA [69] architecture, we used a Plasticine simulator, already extended to operate with Taurus applications, to extract the necessary metrics. The simulator can be found in this GitHub repository [70]. The configuration of the MapReduce block is similar to the one in the original Taurus paper [4]. We set the maximum combined number of CUs and MUs to 120, just like Taurus does, but we do not enforce a 3:1 ratio of CUs to MUs. We allow the simulator to select the ratio that will bring the best results in terms of the number of MUs and CUs. This is because our use case is not dependent on this ratio. We do, however, maintain the configuration of each CU, 16 lanes and 4 stages, and each MU, 16 banks with 1024 entries each. This way, each CU consumes $0.044mm^2$ in area and each MU $0.029mm^2$. We also do not allow the combined area to exceed the value of the original architecture, $4.83mm^2$ (90 CUs $\times 0.044 + 30$ MUs $\times 0.029$). This is because this value is considered a negligible overhead with regards to a reference switch with $500mm^2$, when compared to the improvements it brings [4].

5.2 Evaluation with the Tofino Switch

First and foremost, we try to understand if our implementations, both memory and computationally intensive, correctly compute multiplications, divisions, and inversions with elements of a Finite Field. For the memory intensive approach, with the $GF(2^8)$ Field, we copied the tables from [5] (Figures 3.1 and 3.2) and loaded them into the switch MATs. We loaded the program in the simulator and sent packets via the Scapy tool [71], using the header showcased in Figure 4.1.

As for the computationally intensive approaches, we empirically tested their correctness. We also installed each program in the switch and sent packets via the same tool using the header presented in Figure 4.1.

Since all the Field sizes we were able to run with the Tofino were relatively small – $GF(2^8)$ was the largest one – we tested our programs for all possible combinations of elements of the Fields. We compared the results with several online resources like [72] and got 100% of our computations correct for both approaches.

Turning our attention over to the **second and third** questions from this chapter's introduction, we know that for the operations to be executed at line rate, we cannot have any kind of recirculation of packets. For the computationally intensive approaches that are based on iterations, it is easy to argue that we could work with any Finite Field size if we let each packet go through the switch as many times as needed. For example, a multiplication using RPA for the Finite Field $GF(2^{128})$ (128 iterations) using our P4 program that can perform eight iterations of the algorithm would require at least 16 passes, which completely shatters the capability to process packets at line rate. As such, the operation needs to be completed in a single pipeline pass of the packet through the switch.

As an important note, we only focused on Finite Fields whose elements are represented as multiples of one Byte (or a fraction of one Byte, when that size was not possible) – $GF(2^4), GF(2^8), GF(2^{16})$, and so on. Finding these limits is straightforward since the P4 compiler is responsible for allocating the resources. In other words, if the compiler accepts the program, it means it runs at line rate in the switch. Our results show that the maximum Field size for the various operations and approaches is:

- 16 bits ($GF(2^{16})$) for multiplication, division, and inversion using the table-based, memory-intensive approach
- 8 bits ($GF(2^8)$) for the Russian Peasant Algorithm for multiplication
- 3 bits ($GF(2^3)$) for the EBd algorithm for division
- 4 bits ($GF(2^4)$) for the inversion algorithm
- 3 bits ($GF(2^3)$) for division using inversion and RPA

	Stages	Header size (Bytes)	VLIW	SRAM	Logical Table ID
Table approach	7	29	2.86%	14.27%	5.73%
RPA	9	22	3.64%	1.35%	15.1%
EBd	11	32	5.73%	1.35%	28.13%
Inverse	12	28	7.29%	1.35%	29.17%

Table 5.1: Tofino resources used by the several approaches and algorithms implemented

With the table-based approach, 16 bits is the maximum possible size, as when we try to advance to $GF(2^{24})$, we know we would have 16777216 values with three bytes each, totaling 48MB per table. Unfortunately, the P4 compiler could not allocate that amount of memory. The computationally intensive algorithms had an even worse performance. This was to be expected, as a Tofino switch is focused on table matching efficiency in looking up values, and has limited computational power.

Starting with RPA, the data dependencies between iterations meant that many operations had to be performed in different stages, as they could only occur after the operation on the previous stage was completed. This means the maximum workable size of the Finite Field is bound by the number of stages offered by the Tofino switch. For the next Field we experienced with, $GF(2^{16})$, the number of stages is no longer enough.

EBd in P4 could only work using the Field with three bits. This is mainly due to the last operation $(a/2)_P$. The fact that a `SHIFT` operation consumes all the resources of a stage, that there is no native `ROR` instruction, and that the manipulation of several bits was not parallelizable by the compiler were crucial factors. To this, we add all the other operations that are executed inside a single iteration and the need for $2 * n - 1$ iterations as the justification for the reduced size of the workable Finite Field.

Moving on to the inversion algorithm, we were able to work with $GF(2^4)$, but the issues faced were similar to EBd, even though the $(a/2)_P$ operation is not present here. Again, the amount of `SHIFT`s and `XOR`s needed, the required number of iterations, $2*n$, and the data dependencies between each iteration were the limiting factors. Coupling this with RPA to perform division reduced even further the workable Finite Field size to $GF(2^3)$, meaning that there is no gain when compared to EBd.

Looking at these values, it is clear that they are not satisfactory for many of today's use cases. We will do a more thorough discussion of the implications in Chapter 6.

Moving on to the **fourth and fifth** questions, we used P4i to extract the resources needed for each approach and algorithm, using the maximum Finite Field sizes for each. The results are showcased in Table 5.1 for the number of stages used, the bytes allocated in the PHV, the number of necessary VLIWs, the SRAM usage and the percentage of used Logical Table IDs. Some interesting insights are taken from these results. All of these approaches do not require much header space, with the EBd approach consuming the most but only 6.25% of the total capacity. The discrepancy in header size from the three computational intensive algorithms comes from the metadata used in order to execute them, which goes to the PHV. Recall that our defined header for these algorithms was presented in Chapter 4. Although not computationally complex, the table approach still requires seven stages due to the size of the tables for $GF(2^{16})$ and a significant portion of the total SRAM. In one stage, we also saw that more than 90% of the SRAM was being used. As expected, all other approaches consume much less SRAM, but the trade-off appears in the number of logical table IDs, $2.6\times$ to $5.1\times$ more. The number of stages used is also more significant, even though all these approaches work on smaller Finite Field sizes. Finally, it is interesting to see that the percentage of VLIW for all the approaches implemented was under 10%. With these results, although a fraction of resources is being used, we are far from allocating all the available PHV space (512 Bytes) or the maximum amount of VLIW, SRAM, and Logical Table IDs. This means a switch running these programs can execute other functions and perform other tasks necessary by the use case being implemented.

The **sixth** question comes from the fact that most of the use cases we presented in Chapter 2 will require the switch to perform more than one Finite Field operation for each packet. And, in the majority of cases, that operation is multiplication. In Network Coding, for example, if we want to encode the entire packet using a Field with a smaller size than the packet, we need to execute several multiplications of some packet bits with the selected coefficient. For AES, a similar situation happens.

As such, we conducted some tests in order to see how many multiplications we could execute within a single packet using both the memory and computationally intensive approaches. In other words, we wanted to see how many operands we could perform operations on inside a single packet and obtain the results once the packet left the switch. For a fair comparison, we set the Field to $GF(2^8)$. As a first step, we had to change the header used. We started with the header from Figure 4.1, removed the op header, and added more operands and more space for the results, as presented in Figure 5.1.

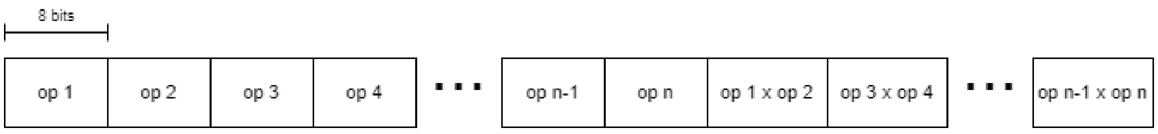


Figure 5.1: Header for multiple Finite Field multiplications

For the memory-intensive approach, recall that each table (logarithm and anti-logarithm) can only match with *one* operand in order to extract the necessary value. As such, for each multiplication, we needed three tables (one table for the logarithm of each operand and another for the anti-logarithm of the sum) with all the possible values of the Field. Since the tables are all similar, we just had to copy the code from Listing 4.1 from lines 11 to 19 in the *apply* block, as many times as the number of operations. We also had to copy the table's definition and change to which operand they should match in the *key* block. Listing 5.1 showcases the tables for a single multiplication operation that must be copied. Using this approach, we reached a maximum of 15 multiplications inside a single packet before the compiler failed to allocate all the necessary fields to the PHV.

Listing 5.1: P4 tables for Finite Field multiplication

```
table table_log_a {
     key = {
        hdr.ff_calc.a: exact;
    }
    actions = {
         get_log_a;
    }
    size = 256;
}

table table_log_b {
     key = {
        hdr.ff_calc.b: exact;
    }
    actions = {
         get_log_b;
    }
    size = 256;
}

table table_antilog_0 {
    key = {
        meta.tmp_sum_value_0: exact;
    }
    actions = {
         get_antilog_0;
    }
    size = 256;
}
```

As for the computationally intensive approach, the RPA algorithm, we know we have to unroll the loop of the algorithm as we did for the version with only one operation. Since the number of stages available is low, it made no sense to follow the same idea as with the memory-intensive approach. In other words,

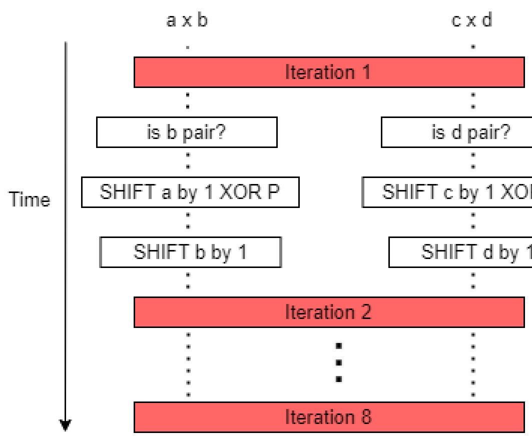


Figure 5.2: Two simultaneous multiplications using RPA

our approach could not be performing all the iterations of the first multiplication and only after having that result starting the iterations for the second multiplication and so on. Our solution was then to, leveraging the parallelism of the Tofino switch, execute the instructions of each iteration on all the multiplications simultaneously. For example, when we need to `SHIFT` the first operand by one bit and `XOR` it with the irreducible polynomial P depending on its value (lines 5 to 8 in Algorithm 3.1), we do these instructions for *all* of the multiplications we are performing. Figure 5.2 illustrates this approach and the produced code is shown in Listings 5.2 and 5.3. Our solution reached a maximum of eight multiplications inside a single packet before the compiler failed to allocate more resources, especially Logical Table IDs.

Listing 5.2: Instructions for 2 simultaneous multiplications using RPA (apply block)

```
    low_bit_f_0 = hdr.ff_calc.b & 0x1;
    low_bit_f_1 = hdr.ff_calc.d & 0x1;
    if (low_bit_f_0 != 0)
        hdr.ff_calc.result_0 = hdr.ff_calc.result_0 ^ hdr.ff_calc.a;
    if (low_bit_f_1 != 0)
        hdr.ff_calc.result_1 = hdr.ff_calc.result_1 ^ hdr.ff_calc.c;

    table_forward.apply();

    if (high_bit_f_0 != 0)
        hdr.ff_calc.a = hdr.ff_calc.a ^ IRRED_POLY;
    if (high_bit_f_1 != 0)
        hdr.ff_calc.c = hdr.ff_calc.c ^ IRRED_POLY;
```

	Number of Multiplications	Stages	VLIW	SRAM	Logical Table ID
Table approach	15	6	11.20%	10.73%	39.58%
RPA	8	12	14.58%	1.87%	70.83%

Table 5.2: Tofino resources used for multiple multiplications in a single packet

Listing 5.3: Instructions for 2 simultaneous multiplications using RPA (action block)

```
    high_bit_f_0 = hdr.ff_calc.a & HIGH_BIT_MASK;
    hdr.ff_calc.a = hdr.ff_calc.a << 1;
    hdr.ff_calc.b = hdr.ff_calc.b >> 1;

    high_bit_f_1 = hdr.ff_calc.c & HIGH_BIT_MASK;

    hdr.ff_calc.c = hdr.ff_calc.c << 1;
    hdr.ff_calc.d = hdr.ff_calc.d >> 1;
```

Finally, we loaded our programs in P4i and extracted the resources in Table 5.2 in order to see what was being used. It is interesting to see that, for the table-based approach, 15 parallel multiplications still only consume a fraction of the SRAM memory of the Tofino switch and a relatively small number of stages. The limitation was in allocating all the data and metadata necessary to the PHV. As expected, the number of Logical Table IDs increased significantly but was still under 40%. Although, as expected, the SRAM usage is much lower in RPA, at less than 2%, the amount of stages doubles as does the number of Logical Table IDs (70%) to execute almost half the number of multiplications. This fact further supports the computational limitations of the Tofino switch and how the table-based approach is more suitable, although limited by its intrinsic scalability issues.

5.3 Evaluation with the Taurus Switch

Again, we **first** assess whether the algorithms we implemented were correct. To that end, we also tried every possible combination of values from the Finite Field and compared them with the online resources presented previously. Since we are just analyzing the MapReduce block of Taurus, which is simulated on top of an FPGA, we were not required to simulate packets and therefore create a new packet header. We can just load as input the values we wanted to work on and print the result – our results showed that 100% of the computations were correct for all three algorithms.

In order to answer the **second** question, we used the Plasticine simulator, which allows us to simulate a chip with a 1GHz clock (1ns cycle time), send the values we want to operate on, and calculate how long the MapReduce block takes to compute and process them. For each of the algorithms, we sent

	CUs	**MUs**	**Area added per pipeline (mm^2)**	**$\%$ of Area added (vs. [4])**
RPA	0	16	0.464	0.37
EBd	40	56	3.384	2.71
Inverse	45	56	3.604	2.88

Table 5.3: Resources consumed in Taurus for $GF(2^8)$

1024 packets with random values from $GF(2^8)$ and got the following results:

- Multiplication completes in 205ns
- Division completes in 763ns
- Inversion completes in 691ns

These results allow us to answer affirmatively to the question of whether or not we can execute these operations at line rate, as our values remain in the order of the hundreds of nanoseconds. Since data center level switches have around $1\mu s$ of latency [73], the added latency is acceptable.

Since Taurus is just a prototype and there is no physical switch available to work on, in order to answer the **third, fourth, and fifth** questions, we have to use the simulator to extract the resources used for our algorithms that operate on the $GF(2^8)$ Field, and then extrapolate the results for larger Finite Fields. Table 5.3 showcases the number of CUs and MUs that each algorithm requires, as well as the chip area that it would consume per reconfigurable pipeline. We will compare the area occupied against a reference programmable switch with four reconfigurable pipelines, which takes $500mm^2$ [4].

For the RPA algorithm, no CUs are being used to compute the results. This result might seem counter-intuitive, but, as we referred in Section 2.4.2, MUs can perform some simple computations, and that is exactly what happens. The mutexes we implemented performed simple operations, and the compiler was able to map those operations to MUs only. The area occupied per pipeline is, therefore, relatively small. Comparing to the reference switch, if all pipelines performed RPA in the $GF(2^8)$ Field, we would only occupy 0.37% more.

The number of MUs, 16, seems to indicate that we are only using two MUs per iteration of the algorithm, which, at first sight, seems like a good result. However, if we extrapolate this to $GF(2^{128})$, we would perform 128 iterations and need 256 MUs, which already exceeds the maximum number of units of Taurus. It would also consume about 6% of the area of the reference switch. This extrapolation, however, is optimistic, as we are not accounting for the number of new FIFO queues we would have to allocate. Although we cannot operate with $GF(2^{128})$, if we wanted to use all the available resources of Taurus, we could work with the Field $GF(2^{56})$ or values with seven bytes. This result is a significant improvement concerning what the Tofino switch can achieve.

As for the EBd algorithm, we observe a great increase in the amount of used resources, consuming 96 units, close to the 120 we set as the maximum. This, of course, translates to a greater consumed area, of around $3.384mm^2$ per pipeline and 2.71% of the reference switch. This usage is mainly due to three reasons. First, there are more iterations to be performed versus RPA for the same Finite Field. This fact, of course, translates into more computations, which need to be executed sequentially. Second, more iterations and variables mean more FIFO queues to be allocated and more memory necessary. Third, the instructions inside a single iteration are more complex. As presented in Chapter 4, we have several nested mutexes and, above all, the $(a/2)_P$ operation, which required three different Pipe blocks and more FIFO queues.

The resources suggest that around 2.67 CUs and 3.73 MUs are needed per iteration. With the current architecture of Taurus, we cannot compute divisions in Finite Fields greater than $GF(2^8)$. This is an improvement with regard to the computational algorithm implemented in Tofino, but still worse than the memory-based approach.

The Inverse algorithm presents similar results to EBd, almost exhausting all the available CUs and MUs of Taurus. This is also due to the number of iterations necessary for the algorithm to correctly compute the results, two times the size of the Finite Field. But it is also due to the instructions that must be executed and the number of FIFO queues necessary, five per iteration. Although there are no complex operations like $(a/2)_P$, several variables require three nested mutexes, which consumes a lot of resources.

With 2.81 CUs and 3.5 MUs needed per iteration, the current settings of Taurus only allow for the inversion to be calculated for the Field $GF(2^8)$ as well. Once again, these results are an improvement compared to the computational algorithm in Tofino, but not the memory-intensive one.

Moving on to the **fifth** question, there are two very different results. For multiplication, using RPA, we are only using 16 units, corresponding to 1.33% of all the available units. As such, the remaining ones could easily be used to perform any other tasks required by the use case being implemented. For larger Fields, for example $GF(2^{32})$, which Tofino cannot operate on with either of the approaches, we would use 64 MUs, or around 53.33%, still leaving 56 units to other tasks.

Once we take a look into division and inversion, however, the results are different. With the Field $GF(2^8)$, we are using 80% and 84.2% of the available units, respectively. Although it is not 100% of usage, it only leaves 24 free units with division and 19 with inversion.

Other tasks could be performed, but their complexity cannot be high. We recall, however, that Taurus still has MATs for pre and post-processing, which we are not interfering with and are free to be used for tasks like forwarding, changing header values (destination IP addresses, MAC addresses), etc.

Finally, for the **final** question, we can look at it from two different approaches. The first one is similar to what we did for the Tofino switch using the RPA algorithm. We know each *Pipe* block is responsible for executing one iteration of the algorithm, so we can perform the instructions of that iteration to all of the multiplications at the same time. We call this version *Parallel*. Listing 5.4 illustrates this idea. The

Pipe block dequeues all the values from the FIFOs and performs the necessary operations on all the operands in parallel, before queuing them again for the next *Pipe* block to use.

Listing 5.4: Pipe block with instructions for 2 simultaneous multiplications

```
Pipe {
    val a = a_stage0.deq()
    val b = b_stage0.deq()
    val c = c_stage0.deq()
    val d = d_stage0.deq()

    val result_0 = result_0_stage0.deq()
    val result_1 = result_1_stage0.deq()

    result_0_stage1.enq(mux(b.bit(0) == 1, result_0 ^ a, result_0))
    result_1_stage1.enq(mux(d.bit(0) == 1, result_1 ^ c, result_1))

    val flag_a = (a & mask_a) == mask_a
    val flag_c = (c & mask_a) == mask_a

    a_stage1.enq(mux(flag_a, (a << 1) ^ irred, a << 1))
    c_stage1.enq(mux(flag_c, (c << 1) ^ irred, c << 1))
    b_stage1.enq(b >> 1)
    d_stage1.enq(d >> 1)
}
```

The second approach is to follow what we did for the Tofino Switch with the memory-based algorithm. Recalling that for $GF(2^8)$, we need eight iterations and, therefore, eight *Pipe* blocks, we could just copy sets of eight blocks as many times as multiplications we want to perform. We call this approach *Sequential*. We use Listing 5.5 to showcase it. The first *Pipe* block in line 1 is only responsible for the first multiplication, and, of course, there are eight similar *Pipe* blocks for that operation. Therefore, the block in line 15 is responsible for the second multiplication alone, and it only executes after the first one is completed.

Listing 5.5: Pipe blocks with instructions for 2 multiplications

```
1  Pipe {
2      val a = a_stage0.deq()
```

```
    val b = b_stage0.deq()
    val result = result_0_stage0.deq()

    result_0_stage1.enq(mux(b.bit(0) == 1, result ^ a, result))

    val flag_a = (a & mask_a) == mask_a
    a_stage1.enq(mux(flag_a, (a << 1) ^ irred, a << 1))
    b_stage1.enq(b >> 1)
}

[...]

Pipe {
    val c = c_stage0.deq()
    val d = d_stage0.deq()
    val result = result_1_stage0.deq()

    result_1_stage1.enq(mux(d.bit(0) == 1, result ^ c, result))

    val flag_c = (c & mask_a) == mask_a
    c_stage1.enq(mux(flag_c, (c << 1) ^ irred, c << 1))
    d_stage1.enq(d >> 1)
}
```

	Multiplications	**CUs**	**MUs**
Parallel	9	53	63
Sequential	6	56	56

Table 5.4: Taurus Resources for multiple multiplications

We ran both approaches and extracted the resources used in Table 5.4. As can be seen, and as is expected, the *Parallel* version is superior. It is capable of doing nine parallel multiplications, consuming a total of 116 units, four less than the total Taurus has available. On the other hand, the *Sequential* version is only able to do six, consuming a similar amount of resources, only four units less than the other approach. Comparing this to the Tofino results we obtained, it is clear that there is only a marginal gain with respect to the number of parallel multiplications that can be performed using the RPA algorithm. Both approaches perform significantly poorer than the memory-based approach we tested with the Tofino switch.

	Multiplication	**Division**	**Inversion**
Tofino	$GF(2^8)$	$GF(2^3)$	$GF(2^4)$
Taurus	$GF(2^{56})$	$GF(2^8)$	$GF(2^8)$

Table 5.5: Maximum Field size achievable by the architecture, using the computationally-intensive approach

	Tofino	**Taurus**
Q1: Are the algorithms correctly implemented?	Yes	Yes
Q2: Can we execute the operations at line rate?	Yes, for small FF sizes	Yes, for larger FF sizes
Q3: Can we execute the operations with a satisfiable FF size?	No, Table 5.5	No, but improvement over Tofino. Table 5.5
Q4: How many resources are used for each approach?	Table 5.1	Table 5.3
Q5: Can the switch perform other actions while executing FF operations?	Yes	Yes
Q6: How many multiplication operations can we execute inside a single packet?	15 table-intensive 8 computationally-intensive	9

Table 5.6: Summary of the evaluation results

5.4 Summary

We summarize our findings in Table 5.6, succinctly answering the six proposed questions for each one of the architectures.

6 Conclusion

Contents

6.1 Conclusions

In this thesis, we designed, implemented, and evaluated Finite Field operations in programmable switches, both using state-of-the-art commercially available architectures, and new prototype architectures recently proposed. Importantly, all our solutions guarantee that packets are processed at line rate, thus guaranteeing Tbps packet processing throughputs and sub-microsecond latencies.

We divided the approaches to performing Finite Field operations into two philosophies. Memory-intensive approaches, which rely on the memory capabilities of the devices, and computationally intensive methods, which leverage their computational power. We presented algorithms to perform Finite Field multiplication, division, and inversion for both of the approaches, all of which already established literature in the area. The key challenge was to adapt these algorithms in order to fit into the strict constraints of the programmable switches that enable line rate processing.

We designed and implemented switch-compatible adaptations for all algorithms for a Tofino switch, the reference architecture of modern switch ASIC and evaluated our solutions for correctness, resource usage, and parallelization of multiplication operations.

From this, we conclude that current hardware can perform Finite Field operations over Fields with at most eight bits, which can actually cater to the needs of some specific use cases, or just as Proofs-of-Concept (like [60, 61]). However, it is not able to

1. perform Finite Field operations for larger Field sizes, and
2. perform enough of those operations in parallel.

These issues are even more prevalent in the operations of division and inversion. Indeed, they are not so common, and many use cases rely more on multiplication. But taking NC as an example, although the majority of the performed operations are multiplication and addition, division still needs to occur in order to decode the information. The maximum size of the Field we were able to achieve in the Tofino switch, using the computationally intensive approaches, is very small, and the number of resources needed was significant. For the memory-intensive approach, although the size was better, the problem of scalability is always present.

Of course, there are easy solutions like adding more stages to the switch or even allowing recirculation, but, this has important drawbacks: it either increases the switch cost (more chip area), packet latency, and/or severely reduces throughput. None of which is acceptable.

Faced with this, we also implemented and evaluated several approaches adapted to the new Taurus switch. The results we achieved showcase how Taurus can be a step in the right direction. The architecture was able to complete the division and inversion algorithms for the $GF(2^8)$ Field and the multiplication algorithm for $GF(2^{56})$ one.

For the $GF(2^8)$ Field, a small amount of area was occupied for the multiplication operation and the latency values were in the order of the hundreds of nanoseconds, as we showcased in Chapter 5. However, the inversion and division algorithms still consumed almost all of the resources Taurus possesses for that same size of Finite Field, and the latency values were close to $1\mu s$.

These results seem to point out that Taurus is a step forward, but it either needs to be refined for these specific use cases, or a new architecture is needed. In terms of refinements to Taurus, there are several avenues we could explore and make part of the Future Work of this thesis. Since number decomposition algorithms are not memory-intensive, one idea is to reduce the size of the MUs and their overall number. Keeping the same overall area, we could reduce the number of lanes, potentially decreasing the number of parallel operations we can perform. Still, we could have more stages per CU or perhaps even more CUs, which means we could operate on larger Finite Fields (recall that the size of the Field defines how many iterations the algorithms have to perform). We could also go the other way and increment the number of lanes in order to work with smaller Finite Fields, potentially being able to perform more operations in parallel.

We hope this thesis is a spark to start the discussion into the design of more powerful switch architectures, which are able to operate on larger Finite Fields while maintaining the capability to operate at line rate.

6.2 Limitations and Future Work

6.2.1 Algorithms

We researched the most commonly used algorithms for performing Finite Field operations that were not created for a specific use case. For example, there are algorithms that perform some computations on one of the operands and store those results in the cache. However, that only works when the device knows one of the operands beforehand. Regardless, there are other algorithms that can be explored and implemented in both architectures, which might give better results than what we achieved.

6.2.2 Optimizations

Our implementations of the algorithms tried to follow the original, adapting them to the feed-forward pipeline of a modern network switch, and we did not explore any optimizations. An interesting area of future work is trying to explore optimizations that are fine-tuned to the target architecture, e.g. by exploiting parallelization opportunities. As an example, we could explore the usage of meta-programming mechanisms to decrement the experienced latency in Taurus.

6.2.3 Other architectures

Taurus is a recently proposed, hybrid architecture that includes two computational models, one that fits the needs of conventional match-based packet processing, and a MapReduce model that is tailored to a different sector of applications, such as those that require Finite Field operations. However, this solution is still not enough for important use cases that require large Finite Field sizes. For this purpose the investigation of new data plane architectures is an area we see as with tremendous opportunities.

Bibliography

[1] C. Fragouli, J. Y. L. Boudec, and J. Widmer, “Network coding: An instant primer,” vol. 36, 2006.

[2] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, “Software-defined networking: A comprehensive survey,” *Proceedings of the IEEE*, vol. 103, 2015.

[3] L. Peterson, C. Cascone, B. O’Connor, T. Vachuska, and B. Davie, “Software-defined networks: A systems approach,” 2021. [Online]. Available: https://sdn.systemsapproach.org/index.html

[4] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, “Taurus: A data plane architecture for per-packet ml.” Association for Computing Machinery, 2022, pp. 1099–1114. [Online]. Available: https://doi.org/10.1145/3503222.3507726

[5] N. R. Wagner, *The Laws of Cryptography with Java Code*. University of Texas San Antonio, 2003.

[6] M. Dworkin, “Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac,” 2007.

[7] M. Legner, T. Klenze, M. Wyss, C. Sprenger, and A. Perrig, “Epic: Every packet is checked in the data plane of a path-aware internet,” 2020.

[8] M. V. Pedersen, J. Heide, and F. H. Fitzek, “Kodo: An open and research oriented network coding library,” vol. 6827 LNCS, 2011.

[9] S. Lang, *Undergraduate Algebra*, 3rd ed., S. Axler, F. W. Gehring, and K. A. Ribet, Eds. Springer, 2005.

[10] M. R. Kibler, “Galois fields and galois rings made easy,” pp. 40–40, 2017.

[11] T. A. Whitelaw, *Introduction To Abstract Algebra*, 2020.

[12] C. Skalka, J. Ring, D. Darais, M. Kwon, S. Gupta, K. Diller, S. Smolka, and N. Foster, “Proof-carrying network code,” 2019.

[13] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, “Simple-fying middlebox policy enforcement using sdn,” *SIGCOMM*, 2013.

[14] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, “Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags,” 2014.

[15] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson, “Fresco: Modular composable security services for software-defined networks,” 2013.

[16] R. Hand, M. Ton, and E. Keller, “Active security,” 2013.

[17] J. Sherry, C. Lan, R. A. Popa, and S. Ratnasamy, “Blindbox: Deep packet inspection over encrypted traffic.” ACM, 2015, pp. 213–226.

[18] A. Perrig, P. Szalachowski, R. M. Reischuk, and L. Chuat, “Scion: A secure internet architecture,” *Scion*, 2017.

[19] J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard (Information Security and Cryptography)*, 1st ed. Springer, 2002.

[20] Aviatrix, “Is amazon inter-region peering encrypted? - aviatrix.” [Online]. Available: https://aviatrix.com/learn-center/answered-access/is-amazon-inter-region-peering-encrypted/

[21] A. Cabañas, “Managing security at aws,” 2020.

[22] M. Baldwin, “Azure encryption overview — microsoft docs,” 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-overview

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 2001. [Online]. Available: http://www.cacr.math.uwaterloo.ca/hac/

[24] M. Dworkin, “The use of galois/counter mode (gcm) in ipsec encapsulating security payload (esp),” 2005.

[25] C. Gkantsidis, J. Miller, and P. Rodriguez, “Comprehensive view of a live network coding p2p system,” 2006.

[26] Veniam, “Veniam - the internet of moving things,” 2012. [Online]. Available: https://veniam.com/

[27] S. H. Tseng, S. Agarwal, R. Agarwal, H. Ballani, and A. Tang, “Codedbulk: Inter-datacenter bulk transfers using network coding,” 2021.

[28] R. Koetter and M. Médard, “An algebraic approach to network coding,” *IEEE/ACM Transactions on Networking*, vol. 11, 2003.

[29] T. Ho, R. Koetter, M. Médard, D. R. Karger, and M. Effros, “The benefits of coding over routing in a randomized setting,” 2003.

[30] N. Zilberman, G. Bracha, and G. Schzukin, “Stardust: Divide and conquer in the data center network,” 2019.

[31] L. Rizzo, “Effective erasure codes for reliable computer communication protocols,” *Computer Communication Review*, vol. 27, 1997.

[32] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*, 2010.

[33] M. Karzand, D. J. Leith, J. Cloud, and M. Medard, “Design of fec for low delay in 5g,” *IEEE Journal on Selected Areas in Communications*, vol. 35, 2017.

[34] A. Nafaa, T. Taleb, and L. Murphy, “Forward error correction strategies for media streaming over wireless networks,” 2008.

[35] R. Puri and K. Ramchandran, “Multiple description source coding using forward error correction codes,” vol. 1, 1999.

[36] N. Sultana, J. Sonchack, H. Giesen, I. Pedisich, Z. Han, N. Shyamkumar, S. Burad, A. DeHon, and B. T. Loo, “Flightplan: Dataplane disaggregation and placement for p4 programs,” 2021.

[37] N. Feamster, J. Rexford, and E. Zegura, “The road to sdn: An intellectual history of programmable networks,” *Computer Communication Review*, vol. 44, 2014.

[38] S. Agarwal, M. Kodialam, and T. V. Lakshman, “Traffic engineering in software defined networks,” 2013.

[39] V. Inc., “Vmware nsx virtualization platform,” 2013. [Online]. Available: https://www.vmware.com/products/nsx.html

[40] C. Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, B. K. Naidu, C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat, “B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan,” 2018.

[41] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, and G. Parulkar, “Onos: Towards an open, distributed sdn os,” 2014.

[42] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM Computer Communication Review*, vol. 38, 2008.

[43] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *Computer Communication Review*, vol. 44, 2014.

[44] P. Organization, "P4runtime specification," 7 2021. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[45] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun, "Spatial: A language and compiler for application accelerators," *ACM SIGPLAN Notices*, vol. 53, 2018.

[46] P. Bosshart, G. Gibb, H. S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," vol. 43, 2013.

[47] I. Corporation, "Intel tofino 3," 2021. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html

[48] arkworks, "algebra/ff at master · arkworks-rs/algebra · github," 2020. [Online]. Available: https://github.com/arkworks-rs/algebra/tree/master/ff

[49] P. Gaudry, L. Sanselme, and E. Thomé, "Mpfq : Fast finite fields." [Online]. Available: https://mpfq.gitlabpages.inria.fr/doc/doc.html

[50] M. Hostetter, "Github - mhostetter/galois: A performant numpy extension for galois fields and their applications," 2020. [Online]. Available: https://github.com/mhostetter/galois

[51] S. Gueron and M. E. Kounavis, "White paper intel ® carry-less multiplication instruction and its usage for computing the gcm mode," 2014.

[52] J. Groschädl and E. Savaş, "Instruction set extensions for fast arithmetic in finite fields gf(p) and gf(2m)," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3156, 2004.

[53] Y. Chen, S. Lu, C. Fu, D. Blaauw, R. Dreslinski, T. Mudge, and H.-S. Kim, "A programmable galois field processor for the internet of things," *ACM SIGARCH Computer Architecture News*, vol. 45, 2017.

[54] J. L. Imana, "Low-delay fpga-based implementation of finite field multipliers," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, 2021.

[55] M. A. García-Martínez, R. Posada-Gómez, G. Morales-Luna, and F. Rodríguez-Henríquez, "Fpga implementation of an efficient multiplier over finite fields gf(2 m)," vol. 2005, 2005.

[56] P. H. Namin, R. Muscedere, and M. Ahmadi, “Digit-level serial-in parallel-out multiplier using redundant representation for a class of finite fields,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, 2017.

[57] S. T. Fleming and D. B. Thomas, “Hardware acceleration of matrix multiplication over small prime finite fields,” vol. 7806 LNCS, 2013.

[58] F. Hauser, M. Häberle, M. Schmidt, and M. Menth, “P4-ipsec: Site-to-site and host-to-site vpn with ipsec in p4-based sdn,” *IEEE Access*, vol. 8, pp. 139 567–139 586, 2020.

[59] D. Scholz, A. Oeldemann, F. Geyer, S. Gallenmüller, H. Stubbe, T. Wild, A. Herkersdorf, and G. Carle, “Cryptographic hashing in p4 data planes,” 2019, pp. 1–6.

[60] X. Chen, “Implementing aes encryption on programmable switches via scrambled lookup tables,” 2020.

[61] D. Goncalves, S. Signorello, F. M. Ramos, and M. Medard, “Random linear network coding on programmable switches,” 2019.

[62] J. D. Ruiter and C. Schutijser, “Next-generation internet at terabit speed: Scion in p4,” 2021.

[63] Y.-H. Chen and C.-H. Huang, “Efficient operations in large finite fields for elliptic curve cryptographic,” *International Journal of Engineering Technologies and Management Research*, vol. 7, 2020.

[64] C. H. Wu, C. M. Wu, M. D. Shieh, and Y. T. Hwang, “Systolic vlsi realization of a novel iterative division algorithm over gf(2): A high-speed, low-complexity design,” vol. 4, 2001.

[65] J. H. Guo and C. L. Wang, “Hardware-efficient systolic architecture for inversion and division in gf(2m),” *IEE Proceedings: Computers and Digital Techniques*, vol. 145, 1998.

[66] K. Kobayashi, N. Takagi, and K. Takagi, “An algorithm for inversion in gf(2m) suitable for implementation using a polynomial multiply instruction on gf(2),” 2007.

[67] E. Rescorla, “Rfc 2631 - diffie-hellman key agreement method,” 6 1999. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc2631#section-2.2

[68] L. Batina, N. Mentens, K. Sakiyama, B. Preneel, and I. Verbauwhede, “Low-cost elliptic curve cryptography for wireless sensor networks,” vol. 4357 LNCS, 2006.

[69] R. Prabhakar, Y. Zhang, D. Koeplinger, M. Feldman, T. Zhao, S. Hadjis, A. Pedram, C. Kozyrakis, and K. Olukotun, “Plasticine: A reconfigurable architecture for parallel paterns,” vol. Part F128643, 2017.

[70] Y. Zhang, “Github - stanford-ppl/pir,” 2017. [Online]. Available: https://github.com/stanford-ppl/pir

[71] P. Biondi, “Scapy,” 2021. [Online]. Available: https://scapy.net/

[72] D. of Electrical and C. E. U. of New Brunswick, “$gf(2^m)$ calculator,” 2013. [Online]. Available: https://www.ece.unb.ca/cgi-bin/tervo/calc2.pl

[73] D. Technologies, “Data center networking - quick reference guide,” 2020. [Online]. Available: https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-Networking-Data-Center-Quick-Reference-Guide.pdf