Towards Finite Field Primitives in Network Switches

Daniel Gouveia da Costa Seara
Instituto Superior Técnico
Lisbon, Portugal

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

1 Introduction

From the vast pool of networking applications present nowadays, a considerable percentage of them 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 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.

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 CPU, 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.

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.

The remainder of this paper is composed of 5 Sections. After presenting the back ground and related work in Section 2, Section 3 explains how to perform the most basic operations over Finite Fields: addition, subtraction, multiplication, and division. Section 4 showcases the solutions we designed and implemented to perform Galois Field operations in both current hardware as well as new data plane architectures. Section 5 evaluates the solutions, providing the grounds for a discussion and the conclusion of our work in Section 6.

2 Related Work

A Field F [22] is a set of elements that have some properties that need to be maintained for both addition and multiplication operations. These properties are (1) F is an abelian Group [26] with respect to the addition, (2) $F^* = F \setminus \{0\}$ is a Group with respect to the multiplication (Note that it does not need to be abelian), and (3) The multiplication is distributive with respect to the 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 $(x + y) \times z = x \times z + y \times z$.

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 [26]. The most common GFs come from the set of integer numbers $\mathbb{Z}$, but only considering the elements up to, but not containing, a specific value p. In this case, the operations are done modulo p. In general, 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 [43].

Fortunately, we are not bound only to use positive integers modulo p to create Galois Fields. We can work in the polynomial space, where each element of the Field is a polynomial. In computer science, the class of Fields 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. This property means we can think of any Finite Field $GF(2^m)$ as the set of all numbers that fit in at most m bits.

Finite Fields have been applied in a plethora of use cases. We present three – in-network Cryptography, Network Coding, and Forward Error Correction.

Implementing security solutions in the switch data plane is an appealing idea that can drastically change the network security landscape. New Internet architectures like SCION [34] and its data-plane algorithms, EPIC [27] leverage AES for their cryptography needs. 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 the 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 [28]. Secure AES modes widely used in practice, such as AES-GCM (e.g., as used in TLS and IPsec [9]), requires GF operations on a Field with (at least) 128 bits ($GF(2^{128})$) [8].

IP networks commonly use a *store-and-forward* mechanism for packet forwarding. Network Coding (NC) [12] proposes an alternative: *store-code-forward*. Network coding uses GF arithmetic at its core. One of the first known applications was Avalanche [14], an NC-based P2P content distribution system used for Microsoft Secure Content Distribution. Companies like Veniam [41] are also using NC techniques to improve throughput in WiFi in the IoT space. Recently, Network Coding has also been proposed to improve the throughput of inter-datacenter bulk transfers [40].

Finally, Forward Error Correction (FEC) codes [37] are an important mechanism for reliable network communication. The encoding of data to create the FEC codes and the decoding process are similar to that of Network Coding, which was previously discussed. FEC codes have seen a lot of usage in networking systems like low latency 5G networks [21], media streaming over wireless networks [29], and multiple description source coding [36].

If we want to perform Finite Field operations in the devices, we cannot rely on traditional IP networks and their hardware. Software Defined Networks (SDN) [10, 25] has emerged as a new network paradigm aiming to remove traditional networks' barriers. 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 network elements, and the decision process is *logically* centralized in the SDN controller.

P4 [3] was developed as a high-level programming language that could program any compliant device in a unified manner. P4 was designed to achieve three main goals (1) *target independence*, (2) *protocol independence* and (3), *reconfigurability*. 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 [32].

The main goal of P4 was to program Packet Processors, but other types of reconfigurable hardware exist, like FPGAs and CGRAs. One of the most recently proposed ways to program these devices is the Spatial domain specific language [24], 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.

The reason behind the creation of P4 was the appearance of a new switch chip architecture [4]. 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)* [35].

Although PISA is the main data plane architecture used in switches nowadays, a few others have been proposed recently. One of them is Taurus [38], an architecture for performing per-packet Machine Learning. Taurus takes as its basis a standard PISA switch, but it adds custom hardware based on a MapReduce abstraction to the switch pipeline.

Finite Field operations have been implemented directly in the hardware of various computing architectures, from the common CPU[6, 16, 17] we see in personal devices to FPGAs [11, 13, 20, 30] and networking switches ASICs [15], the latter being a greater challenge. There is also, unsurprisingly, a plethora of software implementations of said operations [1, 19, 33].

3 Finite Field Operations

3.1 Finite Field Addition and Subtraction

When we 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. But, crucially, the coefficients still need to remain in $GF(p)$, so the modulo operand must be applied. 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 exactly the same as an XOR operation. As such, adding or subtracting values in $GF(2^m)$ is straightforward. It is a simple bit-wise XOR [42] over the operands.

3.2 Finite Field Multiplication

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, and then move on to the computationally-intensive approach.

The core idea behind the memory-intensive approach is that the product of two numbers a and b, on a finite field F with generator g, can be computed as $a \times b = g^{\log_g a + \log_g b}$ [42]. At first sight, one may think that this approach requires the computation of multiple complex operations. A closer inspection, however, unveils a critical advantage. Specifically, it is possible to *pre-compute* all operations and store them in memory in advance. In other words, a multiplication table can be created only once and can then be reused as many times as needed to compute the product between any two numbers. By decomposing it this way, the multiplication operation can be performed with a small number of table accesses (usually three).

The computational-intensive approach relies upon using number decomposition and manipulating the operands. The most common solution that follows the number decomposition approach is the Russian Peasant Algorithm (RPA) [7].

RPA uses a doubling and halving 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. It is an iterative algorithm, and, in each iteration, a is multiplied by 2, and b is divided by 2; this process is repeated multiple times until b is equal to 1. When this point is reached, the various values of a are summed together, but only for the iterations where b was an odd number. As an important note, for Finite Field multiplication, 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 [7].

3.3 Finite Field Division

The memory-intensive approach is easy to extend to perform division by leveraging the fact that $\frac{a}{b}$ can be expressed as $a \times b^{-1}$. Based on this observation, to perform GF divisions, we first need to compute the inverse of the second operand and then use the multiplication table to calculate the product. Fortunately, we can also pre-compute all the inverse values beforehand. As such, division takes only one more table access than multiplication.

The division operation using more computationally intensive methods can be executed in two different ways. We 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.

For the first option, we rely on the EBd algorithm [44], which is a derivate of Stein's algorithm to find the Greatest Common Divider (GCD) between two numbers (i.e., a and b). [18]. The EBd algorithm for Finite Field division contains two additional helper variables, which we will call v and s. The former is responsible for storing the result of the division and will also manipulate the a operand. 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, which will track the difference between the degrees of the two polynomials we are dividing. This is a significant 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. Inside each operation, simple SHITs and XORs are used. Still, we would like to point out the $(a/2)_P$ operation, 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 SHIFTs and XORs following these rules [44]:

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

The second option for Finite Field division relies on an algorithm that finds the inverse of the second operand [23], which is heavily inspired by the extended Euclid's algorithm. The algorithm of [23] also relies on multiple iterations over the operands, using only simple SHIFTs and XORs as well. This algorithm finds the inverse of a value after, at most, $2n$ iterations, with n being the number of bits of the Finite Field. This algorithm also uses three helper variables u, v and s, the latter storing the irreducible polynomial P and manipulating the operand b, as well as a variable δ that tracks the degree of the polynomial u.

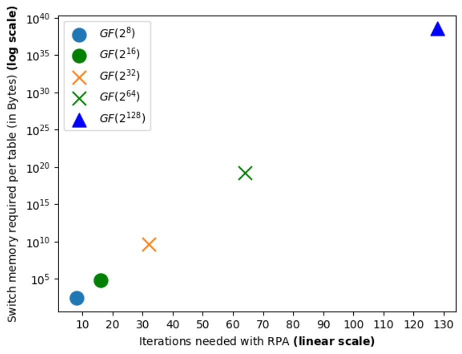

Figure 1. Memory cost vs iteration cost for various finite fields

3.4 Analysis

In summary, memory-intensive approaches leverage the properties of the logarithms and the fact that all the values can be pre-computed beforehand. This fact, 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.

Computationally 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 much better. The cost grows *linearly* with the size of the field, whereas the tables' size *scale exponentially.*

We illustrate this point in Figure 1.

4 Implementation

We focused on implementing multiplication and division operations over Finite Fields and did not implement addition and subtraction because it is a simple `XOR` operation, 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.

4.1 Finite Field Operations in a Programmable Switch

We defined a new header as illustrated in Figure 2. 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). The following two fields store the operands a and b and, finally, the *result* Field will store

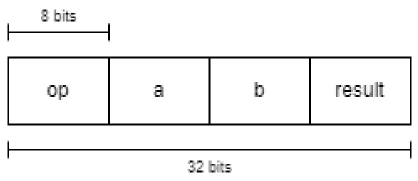

Figure 2. Header for Finite Field operations

the result of the operation. As a note, for the inverse algorithm we do not need both operands, so we ignore one of them.

For the memory-based algorithms, 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.

In the *apply* block of the Ingress pipeline, 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.

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 all the algorithms we first started with trying to implement operations over the Finite Field $GF(2^8)$. 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.

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 and repeat it eight times. At each iteration, a flag is set to 1 when the b operand is odd. This value is then used in an action to decide whether or not to `XOR` a with the accumulator. This action is also responsible for setting a flag, in case a needs to be `XOR`ed with the irreducible polynomial. This operation is done by looking at the most significant bit of a, which for this Finite Field, is the eighth bit. Afterwards, the action multiplies a by 2 and divides b by 2. These are simple `SHIFT` operations by one bit. Since P4 is a pipelined architecture, in order to calculate the correct values for multiplications over $GF(2^8)$, we had to repeat this process eight times.

The maximum size we were able to work with for the EBd algorithm was $GF(2^3)$. In each iteration, we have to perform, among other operations, $(a/2)_P$, which we described in the previous Section. Due to the absence of a `ROR` primitive, we need to explicitly execute the rotate operation and change the bits accordingly, which consumes many resources. We start by saving the a_0 bit, which is placed in the most significant bit before the end of the iteration (a_2). Afterward, we collect the bits a_1a_0 to help us compute what the new value of a_0 should be. The irreducible polynomial P for $GF(2^3)$ is $x^3 + x + 1$ and so the bit P_1 is 1. As such, 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. In the end, we save the value of the a_2 bit, since that will be the new value of a_1 (the bit $P_2 = 0$, so the `XOR` is irrelevant here).

Finally, we implemented the inverse algorithm for the Finite Field $GF(2^4)$. With this Field, we had to repeat each iteration's code eight times, limiting the maximum Finite Field size we could reach. The main challenge of this algorithm was working with the amount of auxiliary variables needed, as well as the number of manipulations per iteration. This translated in a high number of actions and instructions.

4.2 Finite Field Operations in Taurus

In the next section 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 [38]. 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, 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 a slightly modified version of the Spatial language described in Section 2.

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, represented by one or more *Pipe* blocks, 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.

As for RPA, the code we implemented follows the algorithm precisely. We dequeue the results from the previous iteration and use mutexes, which act like *if-else* constructs where needed. 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 $result$. We then 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. Finally, we enqueue the new value of b, which is a simple division by 2. Our implementation works for the Finite Field $GF(2^8)$ but could easily work for larger Field sizes.

Continuing to Finite Field division, and just like with the P4 version, the main challenge we faced was the $(a/2)_P$ operation of the EBd algorithm. We implemented this operation over the $GF(2^8)$ Field. 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, we had to place mutexes inside mutexes to compute the new values correctly. In the end, 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. In the second Pipe block, we execute a Rotate Right 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`. Due to the specific Field we are working on, 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. The final two Pipe blocks use 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.

To conclude, regarding the inversion operation, our implementation also works for the Finite Field $GF(2^8)$. The code is a direct algorithm implementation. We note, once again, the usage of mutexes inside mutexes that are needed to reflect the several nested decisions that the algorithm requires and ensure the correctness of our implementation. For the concrete case of this Field, we had to copy the *Pipe* block sixteen times.

5 Evaluation

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 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 [42] and loaded them into the switch MATs. We loaded the program in the simulator and sent packets via the Scapy tool [2], using the header showcased in Figure 2.

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 2.

We tested our programs for all possible combinations of elements of the Fields. Then, we compared the results with several online resources like [31] 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. As an important note, we only focused on Finite Fields whose elements are represented as a multiple of one Byte (or a fraction of one Byte when that size was impossible). 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

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 the next Section.

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 1.

	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 1. Tofino resources used by the several approaches and algorithms implemented

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 to execute them, which goes to the Packet Header Vector (PHV). 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. As expected, all other approaches consume much less SRAM, but the trade-off appears in the number of logical table IDs, 2.6× to 5.1× 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%. Although many resources are being used with these results, we are far from allocating all the available PHV size (512 Bytes) or the maximum amount of VLIW, SRAM, and Logical Table IDs. This result means a switch running these programs can execute other functions and perform other tasks necessary by implementing the use case.

The **sixth** question comes from the fact that most of the use cases we presented in Section 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.

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. 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 2, removed the *op* header, and added more operands and more space for the results, as presented in Figure 3.

For the memory-intensive approach, recall that each table (logarithm and anti-logarithm) can only match with *one* operand 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. Using this approach, we reached a maximum of fifteen multiplications inside a single packet before the compiler failed to allocate all the necessary fields to the PHV.

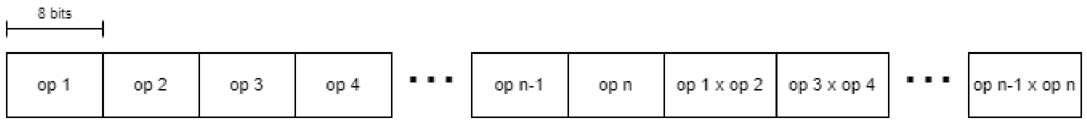

Figure 3. Header for multiple Finite Field multiplications

	Stages	VLIW	SRAM	Logical Table ID
Table approach	6	11.20%	10.73%	39.58%
RPA	12	14.58%	1.87%	70.83%

Table 2. Tofino resources used for multiple multiplications in a single packet

	CUs	MUs	Area added per pipeline (mm^2)	% of Area added (vs. [38])
RPA	0	16	0.464	0.37
EBd	40	56	3.384	2.71
Inverse	45	56	3.604	2.88

Table 3. Resources consumed in Taurus for $GF(2^8)$

As for the computational 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. Our solution, leveraging the parallelism of the Tofino switch, executes 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, we do these instructions for *all* of the multiplications we are performing. Our solution reached a maximum of eight multiplications inside a single packet before the compiler failed to allocate more resources, especially Logical Table IDs.

Finally, we loaded our programs in P4i and extracted the resources in Table 2 in order to see what was being used. It is interesting to see that, for the table-based approach, fifteen 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 increases a lot, but it is 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.

5.2 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 provided by [38], which allows us to simulate a chip with a 1GHz clock (1ns cycle time), send the values we want to operate on, calculate how long the MapReduce block takes to compute and process them. For each of the algorithms, we sent 1024 packets with random values from $GF(2^8)$ and experienced $205ns$, $763ns$ and $691ns$ for multiplication, division and inversion, respectively. Since data center level switches have around $1\mu s$ of latency [39], the added latency is acceptable, and so we can answer affirmatively to this question.

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 3 showcases the number of Compute Units (CUs) and Memory Units (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 programmable reference switch with four reconfigurable pipelines, which takes $500mm^2$ [38].

For the RPA algorithm, no CUs are being used to compute the results. This fact might seem counter-intuitive, but MUs can perform some simple computations, which is exactly what happens. The number of MUs, sixteen, 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. 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.

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

Table 4. Maximum Field size achievable by the architecture, using the computationally-intensive approach

	Multiplications	CUs	MUs
Parallel	9	53	63
Sequential	6	56	56

Table 5. Taurus Resources for multiple multiplications

Looking at the EBd algorithm, we observe that we have a significant increase in the number of used resources, consuming 96 units, close to the 120 we set as the maximum. This usage is mainly due to three reasons:

1. There are more iterations to be performed versus RPA for the same Finite Field.
2. More iterations and more variables mean more FIFO queues to be allocated and more memory necessary.
3. The instructions inside a single iteration are more complex.

The resources suggest that around 2.67 CUs and 3.73 MUs are needed per iteration. Of course, with the current architecture of Taurus, we cannot compute divisions in Finite Fields greater than $GF(2^8)$.

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.

We summarize the findings regarding the maximum workable Finite Field size for each architecture, using the computationally intensive approach in Table 4.

Moving on to the **fifth** question, there are two very different results. For multiplication, using RPA, we are only using sixteen 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, still leaving 56 units to other tasks.

Once we take a look into division and inversion, 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 twenty-four accessible units with division and nineteen with inversion.

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*. The second approach is to follow what we did for the Tofino Switch with the memory-based algorithm. We can copy sets of eight blocks as many times as multiplications we want to perform. We call this approach *Sequential*.

We ran both approaches and extracted the resources used in Table 5. As can be seen, and as is expected, the *Parallel* version is superior and is capable of doing nine parallel multiplications, consuming a total of 116 units, four less than the total Taurus has available.

6 Conclusion

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 [5, 15]). 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 Section 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.

6.1 Limitations and Future Work

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.

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.

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.

References

[1] arkworks. 2020. algebra/ff at master · arkworks-rs/algebra · GitHub. https://github.com/arkworks-rs/algebra/tree/master/ff

[2] Philippe Biondi. 2021. Scapy. https://scapy.net/

[3] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming protocol-independent packet processors. *Computer Communication Review* 44 (2014). Issue 3. https://doi.org/10.1145/2656877.2656890

[4] Pat Bosshart, Glen Gibb, Hun Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *Computer Communication Review* 43. Issue 4. https://doi.org/10.1145/2534169.2486011

[5] Xiaoqi Chen. 2020. Implementing AES Encryption on Programmable Switches via Scrambled Lookup Tables. *Proceedings of the 2020 ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure, SPIN 2020.* https://doi.org/10.1145/3405669.3405819

[6] Yajing Chen, Shengshuo Lu, Cheng Fu, David Blaauw, Ronald Dreslinski, Trevor Mudge, and Hun-Seok Kim. 2017. A Programmable Galois Field Processor for the Internet of Things. *ACM SIGARCH Computer Architecture News* 45 (2017). Issue 2. https://doi.org/10.1145/3140659.3080227

[7] Yan-Haw Chen and Chien-Hsing Huang. 2020. EFFICIENT OPERATIONS IN LARGE FINITE FIELDS FOR ELLIPTIC CURVE CRYPTOGRAPHIC. *International Journal of Engineering Technologies and Management Research* 7 (2020). Issue 6. https://doi.org/10.29121/ijetmr.v7.i6.2020.712

[8] M Dworkin. 2005. The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP).

[9] Morris Dworkin. 2007. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. (2007). https://doi.org/10.6028/NIST.SP.800-38d

[10] Nick Feamster, Jennifer Rexford, and Ellen Zegura. 2014. The road to SDN: An intellectual history of programmable networks. *Computer Communication Review* 44 (2014). Issue 2. https://doi.org/10.1145/2602204.2602219

[11] Shane T. Fleming and David B. Thomas. 2013. Hardware acceleration of matrix multiplication over small prime finite fields. *Lecture*

Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 7806 LNCS. https://doi.org/10.1007/978-3-642-36812-7_10

[12] Christina Fragouli, Jean Yves Le Boudec, and Jörg Widmer. 2006. Network coding: An instant primer. *Computer Communication Review* 36. Issue 1. https://doi.org/10.1145/1111322.1111337

[13] Mario Alberto García-Martínez, Rubén Posada-Gómez, Guillermo Morales-Luna, and Francisco Rodríguez-Henríquez. 2005. FPGA implementation of an efficient multiplier over finite fields GF(2 m). *Proceedings - ReConFig 2005: 2005 International Conference on Reconfigurable Computing and FPGAs* 2005. https://doi.org/10.1109/RECONFIG.2005.18

[14] Christos Gkantsidis, John Miller, and Pablo Rodriguez. 2006. Comprehensive view of a live network coding P2P system. *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC.* https://doi.org/10.1145/1177080.1177104

[15] Diogo Goncalves, Salvatore Signorello, Fernando M.V. Ramos, and Muriel Medard. 2019. Random linear network coding on programmable switches. *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2019.* https://doi.org/10.1109/ANCS.2019.8901883

[16] Johann Groschädl and Erkay Savaş. 2004. 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)* 3156 (2004). https://doi.org/10.1007/978-3-540-28632-5_10

[17] Shay Gueron and Michael E Kounavis. 2014. White Paper Intel ® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode. (2014).

[18] J. H. Guo and C. L. Wang. 1998. Hardware-efficient systolic architecture for inversion and division in GF(2m). *IEE Proceedings: Computers and Digital Techniques* 145 (1998). Issue 4. https://doi.org/10.1049/ip-cdt:19982092

[19] Matt Hostetter. 2020. GitHub - mhostetter/galois: A performant NumPy extension for Galois fields and their applications. https://github.com/mhostetter/galois

[20] Jose L. Imana. 2021. Low-Delay FPGA-Based Implementation of Finite Field Multipliers. *IEEE Transactions on Circuits and Systems II: Express Briefs* 68 (2021). Issue 8. https://doi.org/10.1109/TCSII.2021.3071188

[21] Mohammad Karzand, Douglas J. Leith, Jason Cloud, and Muriel Medard. 2017. Design of FEC for Low Delay in 5G. *IEEE Journal on Selected Areas in Communications* 35 (2017). Issue 8. https://doi.org/10.1109/JSAC.2017.2710958

[22] Maurice R. Kibler. 2017. Galois fields and galois rings made easy. , 40-40 pages. https://doi.org/10.1016/C2016-0-01243-3

[23] Katsuki Kobayashi, Naofumi Takagi, and Kazuyoshi Takagi. 2007. An algorithm for inversion in GF(2m) suitable for implementation using a polynomial multiply instruction on GF(2). *Proceedings - Symposium on Computer Arithmetic.* https://doi.org/10.1109/ARITH.2007.9

[24] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A language and compiler for application accelerators. *ACM SIGPLAN Notices* 53 (2018). Issue 4. https://doi.org/10.1145/3192366.3192379

[25] Diego Kreutz, Fernando M.V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-defined networking: A comprehensive survey. *Proc. IEEE* 103 (2015). Issue 1. https://doi.org/10.1109/JPROC.2014.2371999

[26] Serge Lang. 2005. *Undergraduate Algebra* (3rd ed.). Springer. https://doi.org/10.1007/0-387-27475-8

[27] Markus Legner, Tobias Klenze, Marc Wyss, Christoph Sprenger, and Adrian Perrig. 2020. EPIC: Every packet is checked in the data plane of a path-aware internet. *Proceedings of the 29th USENIX Security Symposium.*

[28] Alfred J Menezes, Paul C van Oorschot, and Scott A Vanstone. 2001. *Handbook of Applied Cryptography.* CRC Press. http://www.cacr.math.uwaterloo.ca/hac/

[29] Abdelhamid Nafaa, Tarik Taleb, and Liam Murphy. 2008. Forward error correction strategies for media streaming over wireless networks. Issue 1. https://doi.org/10.1109/MCOM.2008.4427233

[30] Parham Hosseinzadeh Namin, Roberto Muscedere, and Majid Ahmadi. 2017. 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* 25 (2017). Issue 5. https://doi.org/10.1109/TVLSI.2016.2646479

[31] Department of Electrical and Computer Engineering University of New Brunswick. 2013. $GF(2^m)$ Calculator. https://www.ece.unb.ca/cgi-bin/tervo/calc2.pl

[32] P4 Organization. 2021. P4Runtime Specification. https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[33] Morten V. Pedersen, Janus Heide, and Frank H.P. Fitzek. 2011. Kodo: An open and research oriented network coding library. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6827 LNCS. https://doi.org/10.1007/978-3-642-23041-7_15

[34] Adrian Perrig, Pawel Szalachowski, Raphael M Reischuk, and Laurent Chuat. 2017. SCION: A Secure Internet Architecture. *Scion* (2017).

[35] Larry Peterson, Carmelo Cascone, Brian O'Connor, Thomas Vachuska, and Bruce Davie. 2021. Software-Defined Networks: A Systems Approach. https://sdn.systemsapproach.org/index.html

[36] Rohit Puri and Kannan Ramchandran. 1999. Multiple description source coding using forward error correction codes. *Conference Record of the 33rd Asilomar Conference on Signals, Systems, and Computers* 1. https://doi.org/10.1109/ACSSC.1999.832349

[37] Luigi Rizzo. 1997. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review* 27 (1997). Issue 2. https://doi.org/10.1145/263876.263881

[38] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. 2022. Taurus: A Data Plane Architecture for per-Packet ML. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1099–1114. https://doi.org/10.1145/3503222.3507726

[39] Dell Technologies. 2020. Data Center Networking - Quick Reference Guide. https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-Networking-Data-Center-Quick-Reference-Guide.pdf

[40] Shih Hao Tseng, Saksham Agarwal, Rachit Agarwal, Hitesh Ballani, and Ao Tang. 2021. CodedBulk: Inter-datacenter bulk transfers using network coding. *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021.*

[41] Veniam. 2012. Veniam - The Internet of Moving Things. https://veniam.com/

[42] Neal R Wagner. 2003. *The Laws of Cryptography with Java Code.* University of Texas San Antonio.

[43] Thomas A. Whitelaw. 2020. *Introduction To Abstract Algebra.* https://doi.org/10.1201/9780203750230

[44] Chien Hsing Wu, Chien Ming Wu, Ming Der Shieh, and Yin Tsung Hwang. 2001. Systolic VLSI realization of a novel iterative division algorithm over GF(2): A high-speed, low-complexity design. *ISCAS 2001 - 2001 IEEE International Symposium on Circuits and Systems, Conference Proceedings* 4. https://doi.org/10.1109/ISCAS.2001.922162