FPGA-based indicator mining at line speed
At line speed

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Michaël Molenkamp
born in Hoorn, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

FPGA-based indicator mining at line speed

by Michaël Molenkamp

Abstract

Many devices currently connect to the internet. Some are pretty well secured, while others lack security due to bugs or other vulnerabilities. A scanner searches for available services on the internet or computer host using standard network protocols. An adversary uses a scanner to search for leaks in security. However, scanners encode their network traffic with a specific XOR pattern, also called a fingerprint of the scanner. However, finding those patterns is very computation-intensive on classical hardware.
This thesis aims to use a special FPGA, namely a DFE, to find the fingerprints at a higher speed of 10 GBits/s. Additionally, it aims to find the limits of this DFE in terms of computation power and speed. We created a performance model to find the design requirements and used this data to choose the most optimal algorithm to find the fingerprints.
The performance model showed that we would not reach the intended 10 GBits/s speed. Therefore, we chose the solution that would bring us as close as possible. The potential bandwidth we could reach is 2.4 GBits/s, approximately 12.7 times faster than an optimal high-end CPU implementation using a Ryzen Threadripper 3990x CPU.

KEYWORDS: FPGA design, DFE, Scanners, Fingerprint mining

Laboratory	:	Computer Engineering
Codenumber	:	
Committee Members	:	

Advisor:	Georgi Gaydadjiev
Advisor:	Christian Doerr
Member:	Christian Doerr
Member:	Georgi Gaydadjiev
Member:	Stjepan Picek
Member:	Joost Hoozemans

Contents

List of Figures

List of Tables

Introduction 1

As of 2018, more than 17 billion devices connect to the Internet [9]. This number of devices keeps increasing every day. Due to network-connected devices are available everywhere. An attacker or adversary can launch an attack from anywhere in the world.

Before an adversary attacks, it has to gather as much information about the victim as possible. This is called *reconnaissance*, and it is the first phase of the Cyber Kill Chain (CKC) developed by *Hutchins. et al.* [10]. The CKC is a set of phases that the adversary goes through to compromise a system and acquire assets. Throughout the reconnaissance phase, the adversary searches through publicly available information, including social media and Internet-facing computers. We are interested in identifying the tools an adversary uses for reconnaissance. Our suggested approach requires much computational power to find distinct network traffic patterns, infeasible on classical hardware. However, it will allow us to detect the adversary's suspicious behaviour before a victims' system is compromised. The tool we want to identify is called a scanner.

1.1 Scanners

Scanners are programs that search for computers and open ports on the Internet, where the act of searching for that information is called a scan. These scanners are often port scanners, which both the adversary and the defending party use. An adversary scans a targeted system searching for vulnerable services or other information. Once the adversary finds the weakest link on the system, it performs operations to exploit that weakness and gain access. On the other hand, the defence uses scanners to see if all security policies are in place. That way, the defence is aware of the ports and services visible to the outside and within the network. Scanners use the following different strategies:

Vertical Scan: The vertical scan directs scanning probes to a large number of ports on a single system.

Horizontal Scan: A horizontal scan targets one specific port. The adversary might use this scan to look for a vulnerable service listening to that specific port.

Block Scan: A block scan combines vertical and horizontal scans. This strategy scans multiple ports across multiple hosts.

Every strategy has its advantages and disadvantages. A vertical scan generally has a larger footprint on a singular host, which is why it is easier to detect. Meanwhile, a horizontal scan targets one host at a time, the footprint on one host is small, but it would still be suspicious if it targets multiple hosts in the same network.

1.2 Scanner Detection

Network security specialists use intrusion detection systems (IDS) to detect malicious traffic, such as scanners. These are either hardware devices or special software that monitor the network for anomalies such as scans. An IDS has multiple methods to detect scanners. Anomaly-based detection detects if the network behaves differently than what happens in a steady state. Meanwhile, signature-based detection uses a database that holds signatures or patterns of suspicious network traffic. As an example, a single host scans all the ports of another system. Both the amount of traffic and different ports would be detected as suspicious.

An adversary would try to avoid detection by scanning systems in a stealthy manner. One such stealthy method is performing a very slow sparse scan. Slow scans send a small amount of traffic over long intervals, and because of this, it has a low network footprint. However, the drawback of a slow scan is that it requires a long time to scan one system. For a scan speed up, an adversary could choose to distribute the scan across different hosts, as shown in Figure 1.2. Each of these hosts would then independently scan a different set of ports on multiple systems.

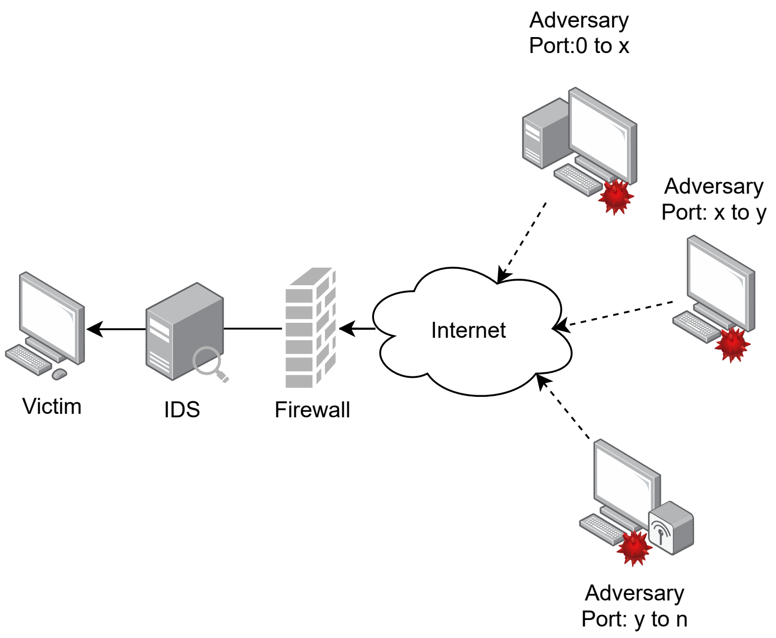


Figure 1.1: A distributed scan.

H.J.Griffioen et al. proposed a method to detect slow scans based on patterns that scanners encode into their network traffic [11]. These patterns remove the need for Scanners to keep track of the traffic they send to the target. Otherwise, they would need to account for all probes[1]. This includes the probes that do not get a response back. Instead, the scanner checks whether it can find its encoded pattern inside the packets it receives. Figure 1.2 showcases this behaviour. When a scanner sends a probe

[1]A probe is a packet send by a scanner to gain information

to the target, the target sends a response with the same pattern, which means it probed successfully. As the pattern is not inside regular network traffic, the traffic gets ignored by the scanner.

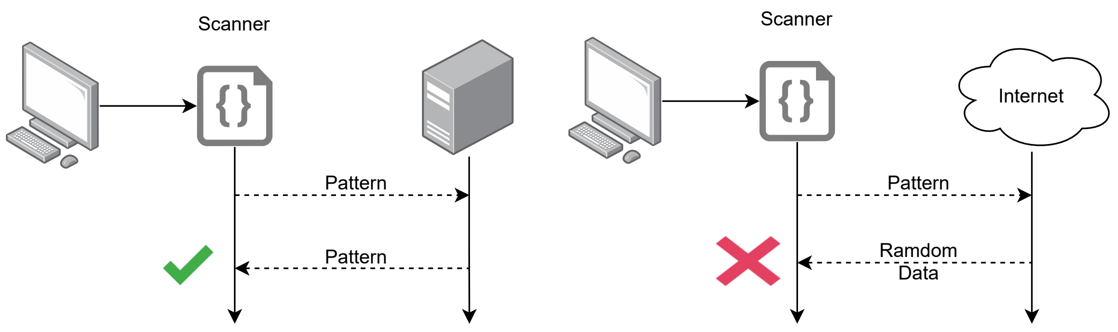


Figure 1.2: Scanner behaviour.

Because the pattern has to return inside the network response, the scanner uses network fields that remain consistent across network communication. A scanner uses these fields to send a probe to a target. In other words, the ports and addresses of the source and destination hosts. However, the sequence number contains the actual pattern. The function of that number is to synchronise the communication state between the two hosts. During network transfer, the client initiates the communication. The client can choose any sequence number it wishes, which is what scanners use to insert their pattern.

The pattern encoding method we focus on uses the XOR operation on shifted ports and addresses fields. The combination of these shifts is the pattern that the scanner validates. The scanner uses the same shifts to look through arbitrary network traffic. Equation (1.1) shows a generic pattern. In this equation, the *Key* in this equation is a secret number only known to the adversary, which obfuscates the pattern.

$$\begin{aligned}Sequence\ Number = {} & (Source\ Address \ll k_0) \oplus (Destination\ Address \ll k_1) \\ & \oplus (Source\ Port \ll k_2) \oplus (Destination\ Port \ll k_3) \\ & \oplus Key\end{aligned} \tag{1.1}$$

To find the pattern, we must search for the values k_i that construct the pattern. These shifts are in the range of $[0, 32]$, which is 33 possible values. We do not know which patterns a scanner would use. So the method for finding the pattern is to go through every possible combination of k_i. In total, that would be $33^4 = 1,185,921$ possibilities. Moreover, we need to compute this for every packet because we do not know what traffic belongs to a specific scanner.

It would take approximately 44 days to check 3 million packets on a general-purpose computer performing 1 million operations per second. This time is impractical to detect real-life scans. We propose to use a Field-Programmable Gate Array (FPGA) to speed up the process. An FPGA is a hardware device where the implemented hardware function is programmable, which would allow for higher productivity and efficiency in performing more operations at each clock cycle. We hypothesise that an FPGA can parallelise a massive amount of the operations directly on the chip's surface, which should significantly accelerate the pattern-finding process.

In the thesis, we answer the following research questions:

RQ1. Can FPGAs outperform conventional systems on XOR pattern detection at line rate for 10Gbit/s connections?
RQ2. What are the major limitations and opportunities of FPGA accelerators when implementing scanner detection?

1.3 Outline Thesis

This thesis describes the process of creating a design for an FPGA implementation. In Chapter 2, we introduce the necessary background information about the network, scanners, and FPGAs. Next, we talk about the speed of the FPGA platform itself. Chapter 3 describes both physical limits, the algorithms and shows the performance model of the algorithm we want to implement. It shows the viability of performing the FPGA algorithm, considering the targeted platform, communication limits, and hardware capacity. In this chapter, the design challenges come forward. Chapter 4 describes the setup configuration used in the thesis. The chapter explains how to use the setup and where to find the necessary data. Chapter 5 shows the design decisions for every part of the algorithm. It describes the details of the design's functionality and how every part of the system interconnects. Additionally, it summarises the design's resource usage and the number of computations performed in a second while comparing it to a CPU implementation. This chapter includes our experiments to test whether it functions as intended. Finally, Chapter 6 answers the research questions and describes directions for future work.

2 Background information

In this chapter, we describe the different layers of the Internet. Afterwards, we discuss scanners, where we describe their strategies and how to detect them. Then we follow it up by giving a general introduction to the FPGA and its resources, starting with explaining why we chose the FPGA for the thesis. Finally, we talk about Maxeler, the company that provided the tools for the thesis. Additionally, we show the specifications of those tools.

2.1 Network traffic

This section describes the network layer model and how the layers interact with each other. Afterwards, we explain the main protocols used in this thesis and why we use certain fields of those protocols.

2.1.1 Open Systems Interconnection model

The Open System Interconnection (OSI) model abstracts the Internet into seven layers to standardize their functionality. Each layer communicates with its adjacent layers. Figure 2.1 shows that data from the Internet flows bottom to top, while data to the Internet does the opposite. This process converts the human-readable data into a stream of bits understood by the network and vice versa. The layers have the following functions:

Application Layer: The Application Layer displays data to the user and processes their requests and actions.

Presentation Layer: The Presentation Layer takes care of the differences caused by the different operating systems between the two hosts.

Session Layer: The Session Layer manages the connection, also called session, between two processes on different hosts.

Transport Layer: The Transport Layer creates a communication link between two processes using the network layer's data.

Network Layer: The Network Layer is responsible for routing data across the world-wide network. This layer also segments the data into smaller pieces and reassembles received data.

Data Link Layer: The Data Link Layer receives the bits from the physical layer and routes it towards the local network's correct destination.

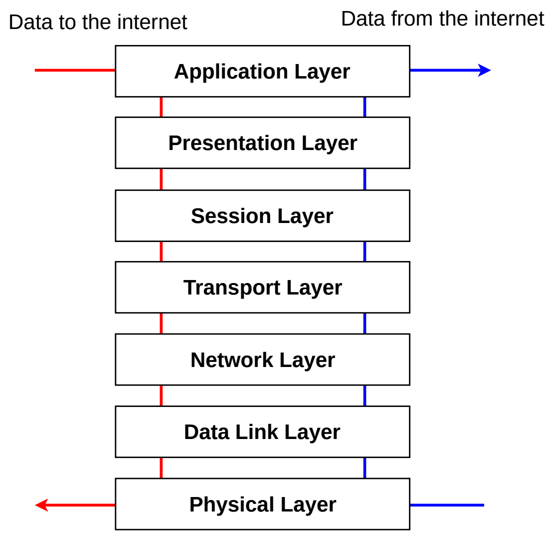
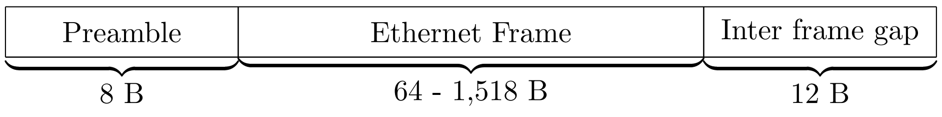


Figure 2.1: The layers of the OSI model.

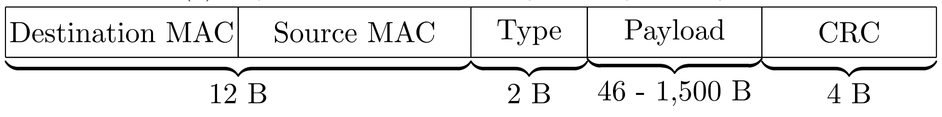
Physical Layer: The Physical Layer transforms the network's data into a bitstream that the Data Link layer can decode. The physical layer abstracts the hardware for transceiving data.

2.1.2 The Physical and Link layer

The Physical Layer receives raw data from the hardware and transforms the raw data into a bitstream interpreted by the Link Layer. Figure 2.2 shows the packet format received by both of these layers according to the 802.3 standard [1]. The device used in this thesis receives data from the hardware layer. So, it is helpful to know the minimum and maximum bytes between packets.



(a) Physical Frame received by the physical layer.



(b) The Ethernet Frame according to the 802.3 standard.

Figure 2.2: The 802.3 packet format[1].

Figure 2.2a shows the raw packet format. The meaning of these fields is as follows:

Preamble: The Preamble is used for synchronization when a packet arrives in the hardware interface. This field has 7 bytes of hardware information and 1 byte of only 1's. The latter denotes the Start of Frame, which tells the network interface where the Ethernet Frame starts.

Ethernet Frame: The Ethernet Frame is data that the Link Layer interprets. This data contains routing and protocol information.

Inter-frame gap: The inter-frame gap is usually 96 bits that are at least between every packet. This gap adds a minimal delay between two packets, which allows the hardware interface to know when a frame ends.

Figure 2.2b shows the fields received by the Link Layer. These fields have the following meaning and function:

MACs The MAC fields are an identifier of the source and destination of a packet within the local network.

Type: Denotes the type of data inside the payload interpreted by the Network Layer.

Payload: The payload contains protocol headers and data used by the next Network Layer and those layers above it.

Cyclic Redundancy Check (CRC): The CRC is used to detect whether data arrived correctly. When a fault occurs during transmission, it drops the Ethernet frame.

The information from Figure 2.2 allows us to calculate the minimum and maximum transfer unit of a physical frame, which Table 2.1 represents. When smaller than allowed frame sizes occur, the ethernet frames are padded with 0's to fit the minimum size. However, if a frame exceeds its maximum size, the ethernet frame is split into multiple frames by the Network Layer.

	Minimum	**Maximum**
Bytes	84	1,538

Table 2.1: Minimum and maximum physical frame sizes.

2.1.3 Network Layer

The Network Layer is responsible for routing data between two hosts and creates a logical path between them. These paths use machine addresses to communicate over the Internet. The Link Layer payload is in essence the datagram that Figure 2.3 shows.

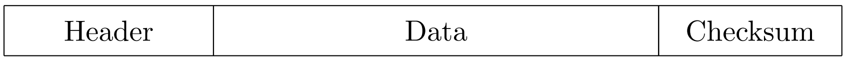


Figure 2.3: A representation of a datagram.

The header contains the source and destination of the data between the hosts. The data field contains the data required by the transport layer, explained in Section 2.1.4. Finally, the layer uses a checksum field to determine if an error occurred during the data transfer. Besides providing the path between hosts, the Network Layer takes care of packets' fragmentation. Fragmentation occurs when the data is too large to send in one go, dividing it into several smaller packet. The destination of those packets is responsible for reassembling the data to its original.

The Network Layer consists of many different protocols. However, the protocol we focus on is Internet Protocol version 4 (IPv4) due to being more widely used.

2.1.3.1 Internet Protocol version 4

The Internet Protocol (IP) primarily routes data between hosts. Figure 2.4 shows the IPv4 header. To better understand how the protocol uses these fields, we clustered them according to their functionality.

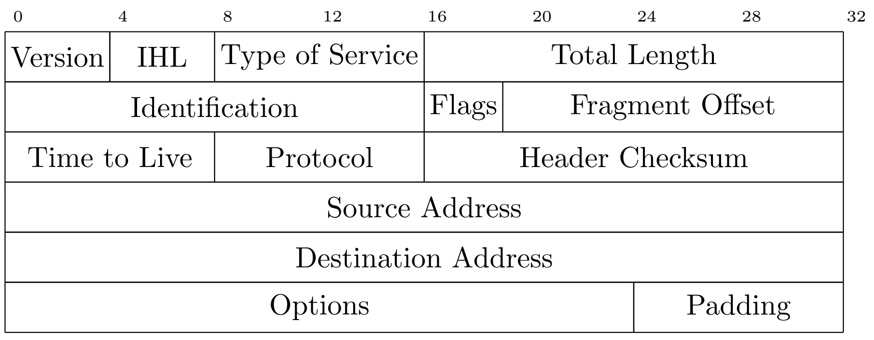


Figure 2.4: The IPv4 header defined in RFC791 [2].

2.1.3.2 Routing fields

IPv4 uses the following fields to route data across the network:

Source Address *32-bits*: This is the address of the host that sends the data. During communication, the returning packet uses this address as its destination.

Destination Address *32-bits*: This is the host's address that will receive packets.

The packets arrive at a router, which redirects the packets from one network to another. Figure 2.5 shows a schematic of how IPv4 transfers data across the network. Routers read the destination address inside a packet and determine the best route for the packet to go. So, as shown in the figure, there are multiple routes that a packet can take to arrive at its destination.

2.1.3.3 Control Fields

Controlling IPv4 uses the following fields:

Time to Live *8-bits*: Indicates the maximum amount of redirects between hosts, also called hops, this packet can take to remain in the network.

Type of Service *8-bits*: This field contains information about precedence for the current datagram; a router uses the value when switching from one network to another.

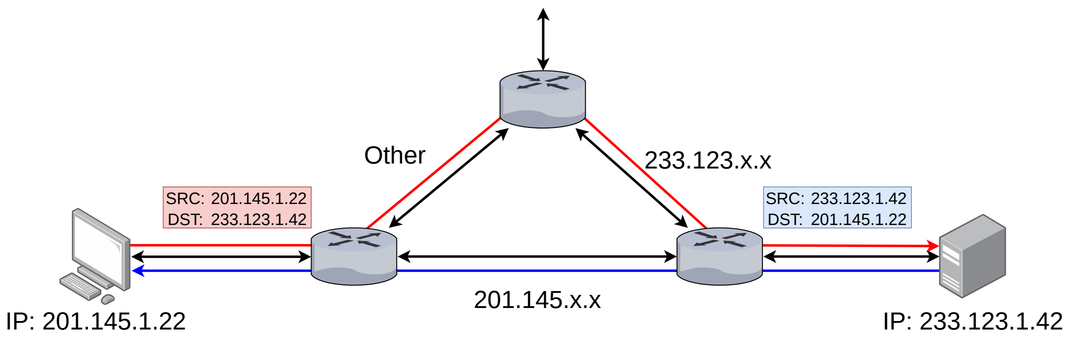


Figure 2.5: IP traffic example.

IHL *4-bits*: This is the Internet header length in units of 32-bit words. The minimum value is 5 to contain all the necessary information.

Total length *16-bits*: This is the total length of the datagram in octets, used to specify the payload's size.

Header Checksum *16-bits*: contains a checksum for the whole packet. This value gets recomputed at every host to verify if the value is valid.

During the routing, the time to live decreases each time it arrives at a router. Once the value becomes zero, the router drops the packet to remain inside the network forever. The changing time to live also requires two computations for the header checksum, as the time to live changes at every routing hop.

2.1.3.4 Reassembly Fields

Reassembling the fragments received from the Internet uses these fields:

Identification *16-bit*: an identifying value that aids the recipient to group fragments for reassembly.

Flags *3-bits*: The three bits in this field determines if the data is fragmented and if it is the first or last fragment.

- **reserved:** should be 0.
- **DF:** This is the do not fragment flag. It requests the computer not to fragment the ethernet frame. However, operating systems ignore the value if the data does not fit on the physical medium.
- **MF:** More fragments, if set to one, the data is fragmented, and more fragments are coming.

Fragment offset *13-bit*: This number represents the starting position of this fragment in 8-bits.

The Internet Protocol collects all the fragments that belong together determined by the identification number, and of course, the source and destination addresses. When a fragment is received, the fragment offset determines the placement of that fragment.

2.1.3.5 Other Fields

Version *4-bit*: The version of the Internet Protocol.

Protocol *8-bits*: The next protocol inside the payload.

Option: This is a variable-sized field, which can contain zero to multiple options. These options contain additional security or routing information.

Padding: This value is also variable-sized to guarantee that the header is a multiple of 32-bit words.

2.1.4 Transport Layer

While the network layer provides a path between hosts, the transport layer connects two processes. While doing that, the layer keeps the connection state of the two hosts.

The two main protocols in this layer are the User Datagram Protocol (UDP) and the Transmission Control Protocol (TCP). UDP allows for stateless communication between two hosts. Meanwhile, TCP keeps track of a session and is the more reliable between the two, and makes sure all data arrives at its destination. We are more interested in the TCP protocol, which we will discuss below.

2.1.4.1 Transmission Control Protocol

The TCP protocol maintains a state of communication between two hosts and ensures that both the client and the host are synchronized. The main point is to provide a safe and reliable way to communicate data between two hosts. Although, it is more accurate to say that the protocol creates a process-to-process connection between two hosts. Figure 2.6 showcases the header information of the TCP protocol, where we clustered the fields with similar functions together to explain them better.

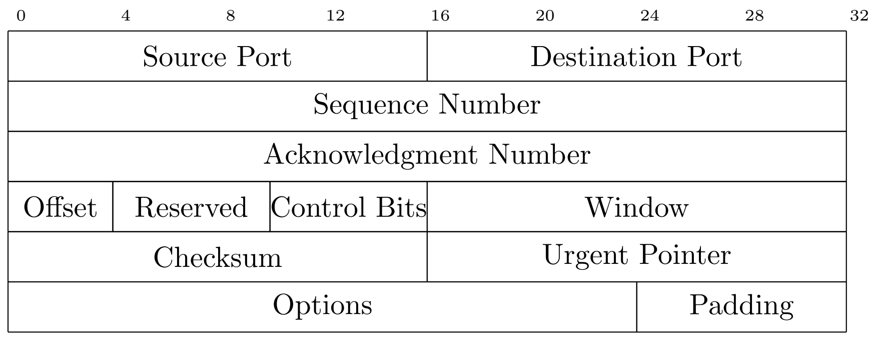


Figure 2.6: The TCP header defined in RFC793 [3].

2.1.4.2 Initialization Fields

A TCP connection starts with a three-way handshake to initialize a connection. Figure 2.7 shows the handshake, which creates the initial communication state by synchronizing the Sequence Numbers. The handshake creates a socket that the adjacent layers use to communicate between the processes. Henceforth, the sequence number indicates the current point of communication.

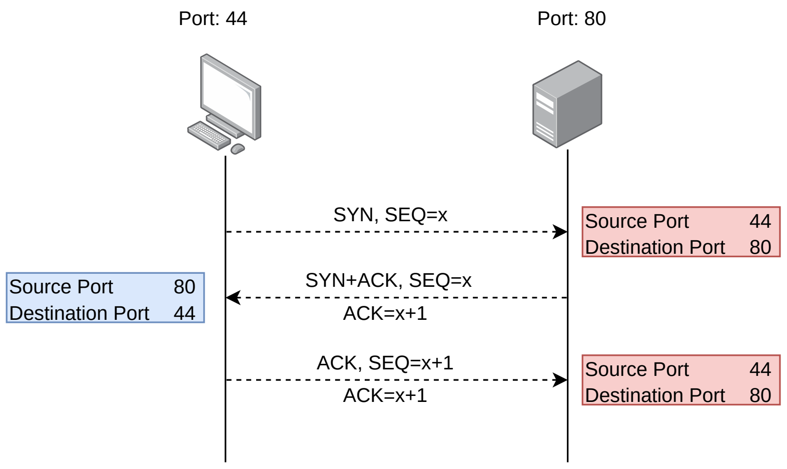


Figure 2.7: Establishing a TCP connection.

The TCP handshake requires the following fields:

Source Port *16-bits*: The source port sends the information in the TCP protocol.

Destination Port *16-bits*: The destination port is the location where the recipient receives the data.

Sequence Number *32-bits*: The sequence number is the synchronization point between two hosts. Every time there is a data transfer between two hosts, the sequence number increases, allowing synchronized data transfer.

Acknowledgement Number *32-bits*: The acknowledgement number contains the next sequence number for the communication, indicating that it received the packet successfully. This number is usually the sequence number added to the length of the packet.

After establishing the communication socket, the response to every piece of data is an ACK packet. This packet tells the sending host that the data arrived safely. If no ACK is received, the packet is re-transmitted. Due to every packed needing to be acknowledged, TCP is a reliable protocol for transferring data.

2.1.4.3 Communication Fields

The control bits field is a 6-bit field that controls the next communication state. These flags synchronize the state between two hosts:

ACK ***Bit 2*****:** The ACK flag indicates a successful arrival of a packet.

SYN ***Bit 5*****:** The SYN flag indicates that the synchronization of sequence numbers between two hosts.

The following flags determine the end of the communication:

RST ***Bit 4*****:** The RST flag indicates a reset of the communication between two hosts due to invalid received data.

FIN ***Bit 6*****:** The FIN flag tells the recipient that there is no more data, and it should close the connection.

The other flags give information about the contents of the payload.

URG ***Bit 1*****:** The URG flag indicates that the Urgent Pointer Field in the header has a meaningful value and is used to increase the Sequence Number

PSH ***Bit 3*****:** The PSH flag indicates that the receiving end should immediately push all of this communication line towards the application.

The URG flag tells the recipient that essential data is available at the location pointed to by the urgent pointer and should be worked on fast.

2.1.4.4 Control Fields

These fields give additional information about the header.

Offset ***3-bits*** This defines the length of the header in multiples of 32-bits.

Window ***16-bits*****:** A field determined by the recipient. It holds the maximum of octets that the host can accept at once.

Checksum ***16-bits*****:** A checksum calculated over the header and data to determine if it has not changed.

2.2 Scanners explained

This section describes what scans are and why they occur. Afterwards, we expand upon how to perform and detect scans.

2.2.1 Origin

As shown in Section 2.1.4, the transport layer is responsible for creating and maintaining a connection between the two hosts. However, before a host can establish a connection, a service has to be listening on a port. In a TCP connection, the client performs the handshake shown in Section 2.1.4.1. However, when a request arrives at a closed port, it returns an RST packet, as shown in Figure 2.8. The traffic incidentally leaks host information, where scanners extract that information from the protocol.

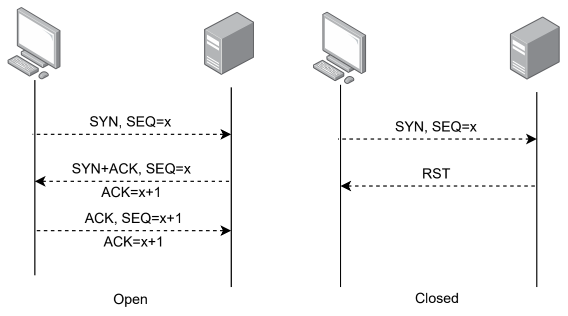


Figure 2.8: Open port vs Closed port.

Scanners are programs that use the TCP connections to find open ports on the system, which could link to a vulnerable service. Listing 2.1 gives an example of the information gathered during a scan, where the service column comes from a list of well-known port numbers.

Listing 2.1: Result of using an NMAP SYN scan on a localhost

```
nmap -sS localhost

Starting Nmap 7.91 ( https://nmap.org ) at 2020-10-14 07:02 W. Europe
     Daylight Time
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0024s latency).
Other addresses for localhost (not scanned): ::1
rDNS record for 127.0.0.1: kubernetes.docker.internal
Not shown: 994 closed ports
PORT     STATE SERVICE
135/tcp  open  msrpc
445/tcp  open  microsoft-ds
902/tcp  open  iss-realsecure
912/tcp  open  apex-mesh
2869/tcp open  icslap
5357/tcp open  wsdapi

Nmap done: 1 IP address (1 host up) scanned in 3.61 seconds
```

2.2.2 Scan types

This section explains how scans use the TCP protocol and the goal of these scans. While there is also an UDP scanning method, these probes are easily fitered by a firewall.

2.2.2.1 TCP Connection Scan

A TCP connection scan uses the three-way-handshake to look through the ports on the target. It is a simple but inefficient scan that requires the operating system to manage and end connections. With a completed connection, this scan would leave a trail on the receiver. For an adversary to remain stealthy, leaving trails on the target system would be a significant enough not to complete a TCP handshake.

2.2.2.2 SYN Scan

A SYN scan limits itself by sending the initial SYN packet, which stops the host from actually finishing the connection. Figure 2.9 shows the response of a SYN scan whether the port is opened or closed. Because this scan uses part of the handshake, it is considered standard behaviour and is more arduous to detect.

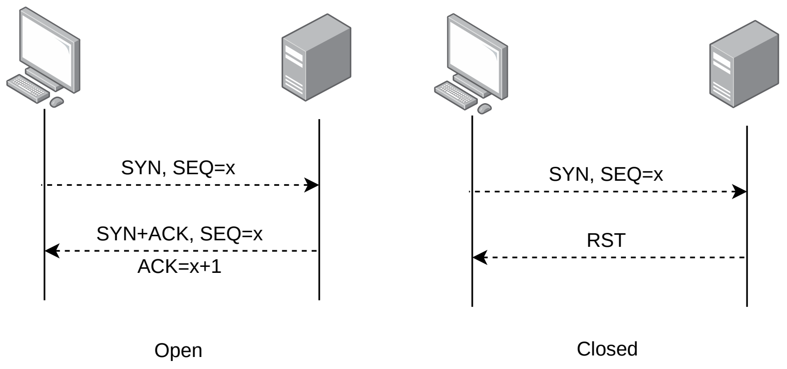


Figure 2.9: The communication during a SYN scan.

2.2.2.3 Uncommon Scans

These uncommon scans all receive an RST packet when a probe arrives at a closed port. However, if the sender receives nothing, the port may be either open or filtered. The following scans are these different ones:

The NULL scan has none of the control bits set.[12]

The FIN scan only sends a FIN packet to its target. It would interpret it as a non-existing connection that wants to terminate the connection gracefully. However, as the connection is unknown to the host, it returns an RST packet.[12]

The XMAS scan uses a combination of the FIN, PSH, and URG flags. This combination of flags is undefined behaviour, which makes it stand out from other packets. The multiple flags inside the packet light it up, this is why it is aptly named XMAS tree scans.[12]

The Maimon Scan specifically targets BSD-derived systems. In BSD there are implementation differences of the network stack. So, when a packet arrives with both FIN and ACK bits set, BSD systems drop the packet when a port is open. While usually an RST packet is returned.[12]

2.2.2.4 ACK and Window Scan

The ACK scan determines if ports are reachable through a firewall. The scan sends a packet with the ACK flag set. Unfiltered ports return an RST packet, to reset the connection. However, if a firewall filters a specific port, there will is no response, or it returns an ICMP packet.

The window scan has the same functionality as the ACK scan. The scan looks at the windows field in the TCP traffic. Some operating systems return a positive value when a port is open, and a zero value when it is closed.

2.2.3 Scanning Strategies

Both the adversary and the defence use scanners. The latter scans its network for any anomalies, while an adversary uses it to gain information about the network. With the different scan types previously discussed, there are a few different strategies to perform the scans.

Figure 2.10 shows the main strategies. The vertical, horizontal, and block scans are enumeration strategies, which means they are used to scan through the hosts' ports. Other characteristics of scans include speed. With speed, the focus on the delay between probes.

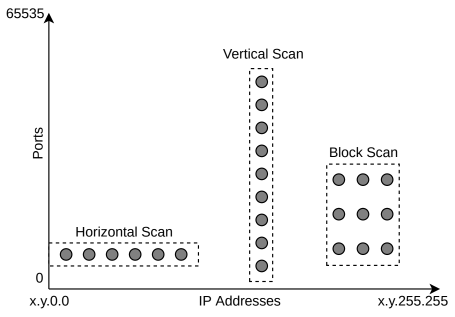


Figure 2.10: Scanning Strategies [4].

A slow scan is a strategy that sends scanning probes very infrequently. Due to the slow pace, this type is harder to detect. The long intervals between packets cause them to not stand out among the regular network traffic. A patient adversary can thoroughly scan a host by letting the host send scans at very infrequent intervals. However, Section 2.1.4.1 showed us that there are 16 bits available for port numbers, which would be a total of $2^{16} - 1$ different possibilities. In the case of 5 minutes between probes, it would take more than half a year to scan one system. By distributing the scan across multiple hosts will create a significant speedup. In that case, multiple hosts work together to scan one singular victim. Figure 2.11 illustrates this process where multiple malicious hosts scan their victim. However, having too many hosts attack a system looks suspicious, so it is essential to balance speed and stealth. The detection method we implement in this thesis can detect these scans and the other types of scans.

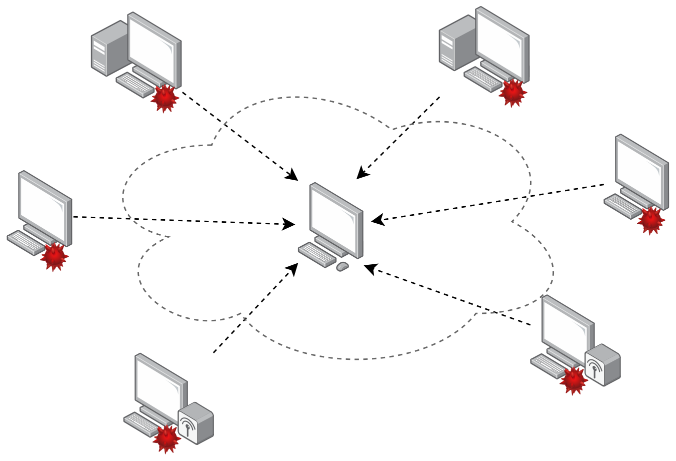


Figure 2.11: A distributed scanning strategy.

2.2.4 Scan detection

Scanners probe a network to gain information. Depending on the combination of scan type and strategy, scans are detected. A known detection method is with intrusion detection systems (IDS). These systems passively detect malicious network traffic, where the IDS notifies administrators when it detects malicious traffic. There are two general methods in which an IDS detects malicious traffic. These are signature and anomaly-based detection.

2.2.4.1 Signature-Based detection

A signature-based IDS A signature is a pattern inside malicious network traffic. These patterns are either a combination of network fields or some specific piece of data. A signature-based IDS uses a database containing known malicious signatures, where the detection methods validate if a specific pattern is inside its database. This method is similar to how an anti-virus functions.

There are multiple drawbacks to a signature-based detection mechanism. First, only known signatures are validated, which means that the IDS is only as good as its database. Furthermore, the adversary can avoid detection by changing a programs' signature. Secondly, there can only be a finite set of signatures inside the database. How more significant the set, how slower the IDS functions.

Let us take the NULL, FIN, and XMAS scans from Section 2.2.2 as an example. An IDS would detect these easily as they are known abnormal traffic. Furthermore, an IDS detects a fast vertical strategy due to the number of probes originating from one host.

2.2.4.2 Anomaly Based Detection

The anomaly-based IDS detects changes or anomalies in the network traffic. Initially, the IDS would learn the normal state of the network as a baseline, and whereafter the IDS compares the incoming traffic to the baseline. When traffic unknown to the IDS arrives, the system would see it as an anomaly, which would either be a positive or a false positive. By teaching the system about what is and what is not malicious traffic, the IDS evolves and removes those false positives.

This method allows the IDS to evolve itself and update its internal state. The anomaly-based detection detects scan traffic in particular as they would diverge from the baseline network state.

2.3 Field Programmable Gate Arrays

This section will first describe a general overview of the FPGA technology, and how it compares against other computation hardware types. Afterwards, we give a general overview of the components used by an FPGA that gives it its flexibility and computational ability.

2.3.1 Speed vs Flexibility

Currently, four main classes of computational hardware can be identified, all with their advantages and disadvantages. This section gives a general overview of these hardware units and illustrates some of their strengths and weaknesses. Afterwards, we clarify why we choose an FPGA.

Central Processing Unit (CPU) The CPU is a hardware structure that performs a generic set of instructions. The flexibility of a CPU is that it can execute its instructions in any order. A programmer creates a list of CPU instructions and puts them inside a program to perform the desired operation and puts them in a program. The CPU executes these instructions sequentially until the program finishes. A programmer can easily change the program written for a specific CPU. It can also port the program to a different CPU architecture, so it functions there too. However, due to a CPU sequential nature, it takes a lot longer to perform the function than with hardware implementation.

To illustrate how a CPU executes, Figure 2.12 shows the Von Neuman architecture. These CPUs use memory to store the program data and calculations.

1. In the first stage, the fetch stage, the CPU fetches the instruction from memory and sends it to the decode stage.

2. At the decoder stage, the control unit interprets the command while it fetches the correct register to send to the computation stage.

3. The Arithmetic Logical Unit (ALU) receives the control unit's operation and performs it on the provided registers.

4. Finally, the computed result is stored back into memory.

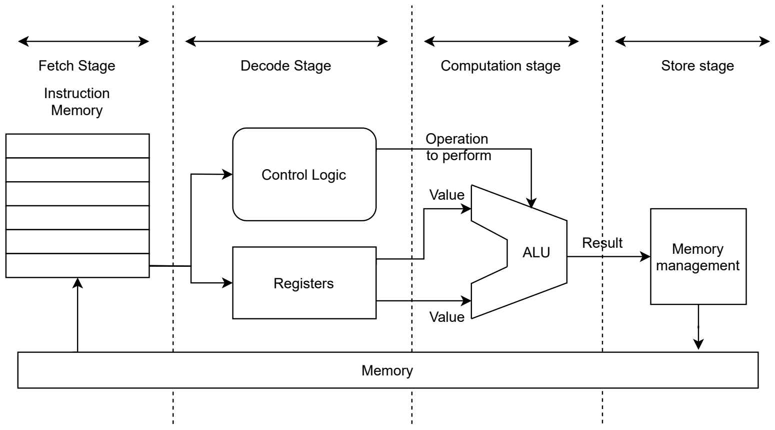


Figure 2.12: CPU representation.

Graphical Processing Unit (GPU) The GPU's were designed to offload graphical computations from the CPU. The GPU architecture uses many streaming processors with more specialized ALUs for graphical operations. These streaming processors connect with high-speed graphic memory that allows for fast computations in parallel. Figure 2.13 shows a typical architecture of a GPU based on Nvidia GPUs.

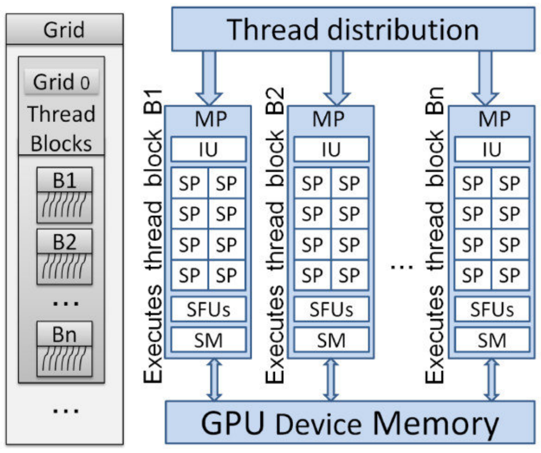


Figure 2.13: Simplified GPU architecture [5].

GPUs are helpful for computation because they perform simple calculations on multiple sets of data in parallel. A CPU is not helpful for this due to its sequential nature. However, using a GPU can induce more latency due to transferring data to and from the GPU using the CPU. So it is always a trade-off between the speedup and latency.

Field Programmable Gate Array (FPGA) An FPGA is a programmable hardware unit that functions at hardware speed. The programmability of the hardware gives the FPGA its flexibility. An FPGA contains many configurable units, such as routing and logic blocks that are interconnected. It can perform any desired function as long as there are enough resources availlable. A price an FPGA pays for its flexibility is a higher power consumption compared to an ASIC. Additionally, as it still more general hardware it considerably slower. However, it performs operations in parallel if specified. Because the hardware itself is programmable, the development cycle is a lot shorter than creating a chip from scratch. So when developing for the unit, it is faster to validate the function by performing it on an actual device. Figure 2.14 shows a general schematic of an FPGA.

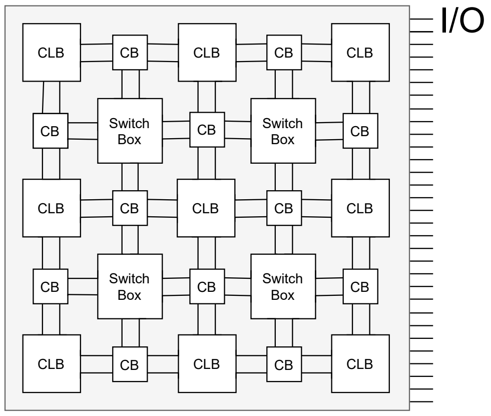


Figure 2.14: FPGA schematic.

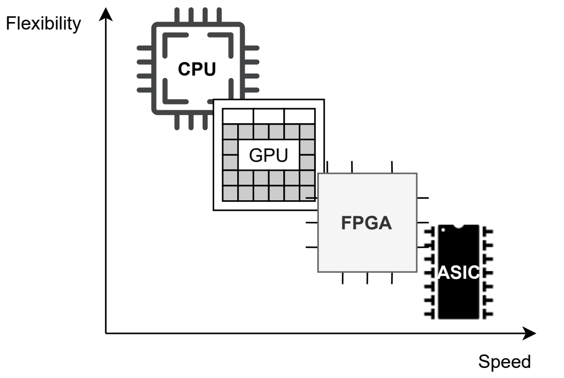


Figure 2.15: Flexibility vs Speed of hardware.

Application-specific integrated circuits (ASICs) These circuits specialize in doing one specific function, allowing designers to optimize it in power consumption and speed. An example of an ASIC are embedded systems, which control everyday appliances, such as keyboards, network chips, televisions, and many more.

A downside of ASICs is that it has a long development cycle. Furthermore, it requires costly tools to create ASICs. However, these chips can be sold and produced in high volume due to their generally small size if the market is large enough to sell all those chips. Once the chip exists, the physical product does not change. When someone discovers a bug, the chip has to be revised. This revision requires additional development, testing, and creation costs for that specific chip. It would come down to 10s of millions of dollars for just that iteration.

Summary Figure 2.15 shows the differences between the systems in terms of operations per second and flexibility. The flexibility originates from being programmable but has a trade-off in the number of operations. In the case of a CPU, its instruction set is general enough to perform any function. However, it performs each operation sequentially, which costs time. The GPU is faster than the CPU due to executing operations in parallel. The trade-off here is the limited functionality of each small GPU core. However, due to its regular structure, it is relatively simple to program compared to an FPGA. The FPGA is a configurable hardware unit that can perform any arbitrary function directly on hardware. In the case of an FPGA, it is not the number of operations but how many operations we can implement considering the resource limitations. An ASIC is, however, still faster as it focuses on only one specific function. So, it is a trade-off between speed, flexibility and costs.

We chose an FPGA implementation because the economic costs are much lower than an ASIC. Furthermore, they take significantly less time to develop. While still being able to perform numerous operations in parallel. That parallelism increases the throughput of the application.

2.3.2 Resources

This section will explain the primary resources of an FPGA and describe the Stratix V FPGA resources. The structure is as follows, for each unit, we first define a general overview. Afterwards, we explain the physical resources for the Stratix V FPGA. All of these resources communicate through high-speed interconnect, which facilitate the data transfer between units.

2.3.2.1 Configurable Logic Block

The Configurable Logic Blocks (CLB) are the building blocks of an FPGA. These units hold the programmable function. The main parts of the CLB are registers and lookup tables (LUTs), where registers hold intermediate results of computations.

The LUT is the programmable unit of the CLB, where each CLB may contain multiple LUTs. Each LUT uses a small amount of Static Random Access Memory (SRAM) to hold its output. Figure 2.16 shows a representation of a CLB that uses multiple LUTs and registers. The routing inside a CLB interconnects the inputs and outputs with the right resources.

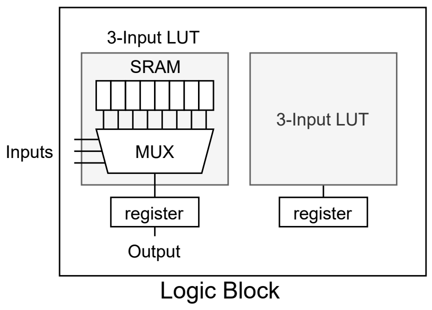
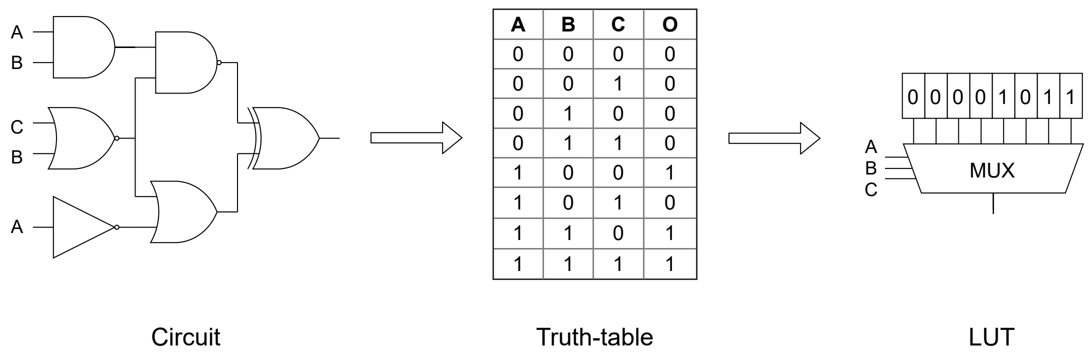


Figure 2.16: A CLB representation.

Figure 2.17 shows an example of how a simple circuit maps into a LUT, where each step goes as follows:

1. We have a physical function we want to encode. This function's direct implementation requires many different logic gates, which we do not have on an FPGA. We can, however, emulate the entire function with LUTs.

2. The truth table describes the complete relationship between the inputs and the given boolean function's outputs.



A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Figure 2.17: How to go from a circuit to a LUT input.

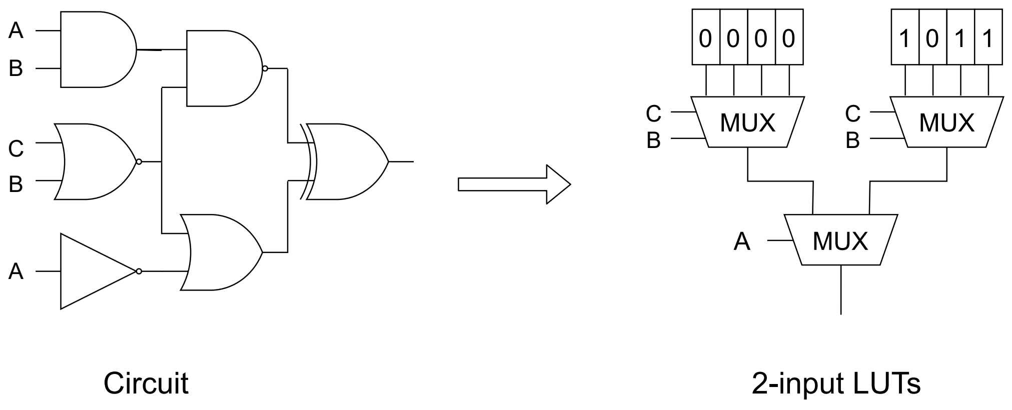


Figure 2.18: Using multiple 2-input LUTs to create a 3-input function.

3. The output of the truth table will determine the LUTs' contents. The inputs of the implemented function, together with the multiplexer, select the correct output.

With LUTs, the number of inputs determines the number of output bits. There is a trade-off between the size of the LUT and how many physically fit on a device. The bigger the size of a LUT, the larger the function it can hold. When the function is too small for the LUT, it is a waste of resources. In comparison, when the LUT is small, there would be more control of the functions. However, it creates more complex and more massive interconnects between logic units. So by choosing the sweet spot between the sizes, there is more room for resources. There is a method to perform more significant logic functions using smaller LUTs by clustering them. As an example, Figure 2.18 shows how to map the same 3-input function using LUTs with only 2 inputs.

An Adaptive Logic Module (ALM) is the primary resource of the Stratix V FPGA.

The structure of an ALM is similar to that of the CLB as shown in Figure 2.19. There are a total of $359,200$ ALMs inside our FPGA, where an ALM contain the following units [6]:

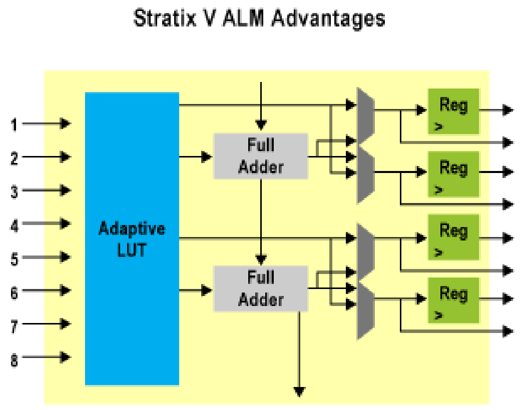


Figure 2.19: ALM Block Diagram [6].

a) **8-input lookup table (LUT):** The LUTs inside ALM divide into different modes of operation. For example, 2×4-input LUTs or 2×6-input LUTs. The latter mode occurs when both LUTs share the same inputs.

b) **2 hardware adders:** These are hardwired 3-bit full adders that allow for additions without any additional logic costs. Furthermore, ALM adders are interconnected. So high-speed calculations are possible.

c) **4 registers:** These allow for storing values between calculations.

d) **4 outputs:** There are two different sets of outputs inside an ALM. These are registered outputs or computations outputs. The former contains the calculation from the previous cycle, while the latter is the immediate result of either the LUT or adders.

2.3.2.2 Static Random Access Memory (SRAM)

SRAM is the memory on the FPGA chip itself. Other names include on-chip memory or block memory (BRAM). It is a high-speed memory that retains its information as long as it is powered. Being available on the chip allows for high-speed reading and writing to and from memory.

On-chip memory is also configurable. It can be split into multiple parts to align it efficiently for the desired function. Some of the use-cases of SRAM are as follows:

- **Storing intermediate values.** SRAM stores intermediate calculation results, which other parts of a design can reuse.

- **As a Read-Only Memory (ROM).** A ROM holds pre-computed results, which are afterwards accessed to find the result of a function. An example would be keeping the results of an x-bit logarithmic function. So the logarithm itself does not have to be computed, which would save other resources.
- **As a First-in First-Out (FIFO) queue.** A FIFO stores a value until it is required. The first stored item is also the first one that goes to the output. An example of a FIFO would be for value transition between logic blocks that use different frequencies.

On-chip memory on the Stratix V FPGA has two different types.

a) **Block Random Access Memory** is memory directly available on the FPGA, and can be accessed quickly.

b) **Memory Logic Array Blocks** These are ALMs configured as static memory.

Table 2.2 shows the total amount of SRAM available on the FPGA.

	M20K	MLAB
Blocks	2,640	17,960
Kbits/block	20	0.640
Blocks×(Kbits/block)	52,800	11,225

Table 2.2: Specifications of SRAM.

Input Output Blocks (I/O) I/O blocks are at the edge of the FPGA. These are individual configurable blocks that allow an FPGA to communicate with external components. The combination of CLBs, Input Output blocks and routing it is possible to create an interface to communicate with external resources. Double Data Rate as an example.

2.3.2.3 Hardwired units

Hardwired units on the FPGA offload expensive operations, such as multiplications, so they do not have to be implemented in CLBs. Additionally, these circuits are faster and more energy-efficient, which is why it is the preferred mode of operation.

The on-chip hardware units The Stratix V FPGA are the adders on the ALU and the digital signal processing (DSP) units. DSP units specialize in floating-point operations and multiplications. There are 352 DSP blocks in total on the Stratix V FPGA. Which have the following features:

Multiply: In this mode, the DSO unit multiplies two numbers. Where these two numbers are both 9, 16, 18, or 27 bits. Larger numbers can be multiplied too but are limited to 36-bit and 18-bit operators.

Multiply addition: This mode performs both multiplication and addition of numbers simultaneously. However, this is only possible with the 18x18 bit selection.

A DSP unit calculates floating-point numbers by cascading multiple DSP units. That method can create a larger precision when performing multiplications, allowing it to reach 36x36 bit multiplications.

2.4 Dataflow Engines

There are FPGA based dataflow accelerators developed and used by Maxeler Technologies. DFEs were used to build state-of-the-art solutions in various sectors, such as finance, government, science, health, security, and high-performance computing. Maxeler provided a Juniper switch with an integrated FPGA module and all necessary development tools and licenses under their Academic program. The following section explains the concept of dataflow platforms, the programming tools, and a short explanation of MaxCompilers toolset. Afterwards, we go into depth about the resources of the intergrated FPGA module.

2.4.1 Dataflow computing

Dataflow computing is different from classical computing done by a CPU. This section will discuss the differences between classical computing and dataflow computing. Moreover, it will go into more detail about the dataflow architecture.

2.4.1.1 Control flow vs Dataflow

Classical computing follows the von Neumann control flow architecture. This is the CPU architecture described in Section 2.3.1.

In contrast, a dataflow architecture executes an algorithm differently. Dataflow architecture is data-driven as it streams data through the different operations it has to perform. Figure 2.20 gives a visual representation of the architecture.

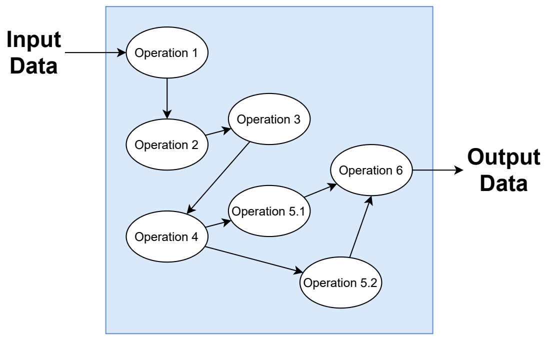
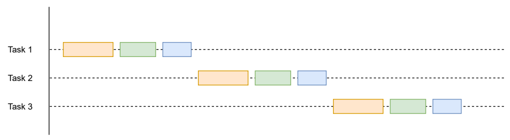


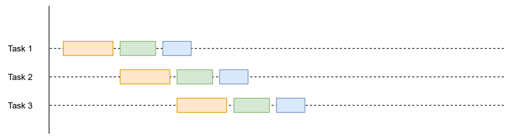
Figure 2.20: Computing in space.

Every dataflow architecture is distinct from another to perform one unique algorithm. The data that flows through the algorithm decides which route it may take. Let us take a conditional as an example. A dataflow architecture calculates all the different branches. Whereafter depending on the data, it chooses which branch of data to use. The dataflow algorithm does not know which of the two to execute. So it executes both and chooses the correct answer afterwards based on the condition outcome.

Pipelining is a concept that allows for more throughput in a program, which is a property of hardware. If there is a set of operations that every task has to perform, it might be the best idea to pipeline it. Figure 2.21 shows an example of pipelining. Figure 2.21a shows tasks that are not pipelined. In the non-pipelined example, every task performs sequentially. However, if pipelining is applied, a task executes if the resource is available. Figure 2.21b shows how a pipeline would increase the throughput of a program, and in turn, allows it to execute more efficiently and faster.



(a) Tasks executed in a sequential manner.



(b) The example of Figure 2.21a pipelined.

Figure 2.21: Non pipelined vs Pipelined.

With an FPGA, we can achieve pipelining by inserting registers between stages of execution. These registers hold the intermediate values between computational stages. Each executable part uses a separate register for both input or outputs. Adding registers between operations allow the circuit to run at a higher frequency because as it shortens the delay between operations as the hold times between functions becomes shorter.

The dataflow architecture also uses pipelining. Figure 2.22 shows the differences between the two architectures using the following simple formula $x^2 + 42 \times x + 3$. A CPU would require four instructions for every input element $'x'$. If we assume a dataset of $1,000$ elements, it would be $4,000$ instructions in total. In a dataflow architecture, all elements stream through the operations in a pipelined fashion. Each operation functions independently in a dataflow architecture, where it pipelines each operation. Because of pipelining, the data streams through the functions, where the time between the first and last operation would be $1,003$ units of time. Here we do assume that all actions take the same time. Otherwise, the throughput gets limited to the longest stage inside the pipeline under consideration.

2.4.1.2 Dataflow engine

A dataflow engine (DFE) is a hardware implementation that uses computing in space to perform an algorithm. Computing in space is a method where data streams between the different components of a system. Every component performs one specific function, such as memory communication or additions. The software tools and on the DFE provided by Maxeler, these components are called kernels. Figure 2.23 shows an example of how a dataflow program would execute on a DFE.

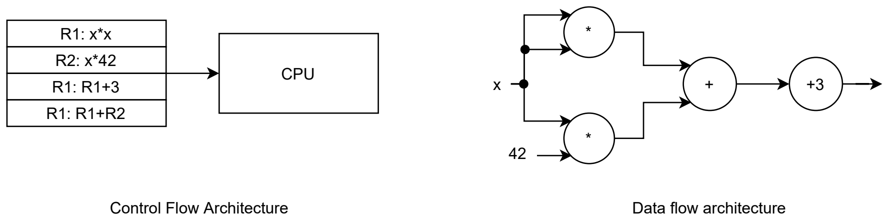


Figure 2.22: Control Flow vs Dataflow.

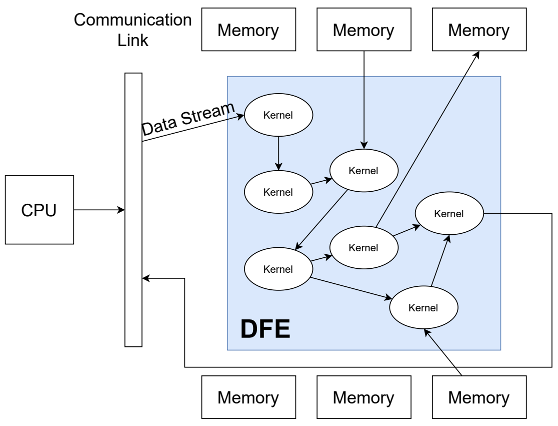


Figure 2.23: Dataflow engine.

2.4.2 Development tools

Maxeler Technologies provided us with the dataflow platform that contains a Stratix V FPGA and a toolset to develop for that platform.

2.4.2.1 The Networking Dataflow platform

The dataflow platform that Maxeler provided is the QFX5100 series switch from Juniper Technologies with an additional QFX-PFA-4Q module that contains an FPGA based DFE. The platform itself allows for packet forwarding between the Juniper Switch's Switching fabric, directly to the FPGA device. Figure 2.24 shows the interconnection of these two modules, where we explain their function from left to right below.

The network ports: These are numbers $0-23$ at the left side of the figure. These are 40Gbit ports that can split into $4 \times 10Gbit$ connections.

The packet forwarding engine (PFE): can route the traffic on the network ports to a port understood by the FPGA.

The middle 10G/40G ports: are logical interfaces able to connect and send data to the FPGA.

QFX-PFA-4Q: The module inside the QFX5100 switch contains the FPGA chip and the additional resources.

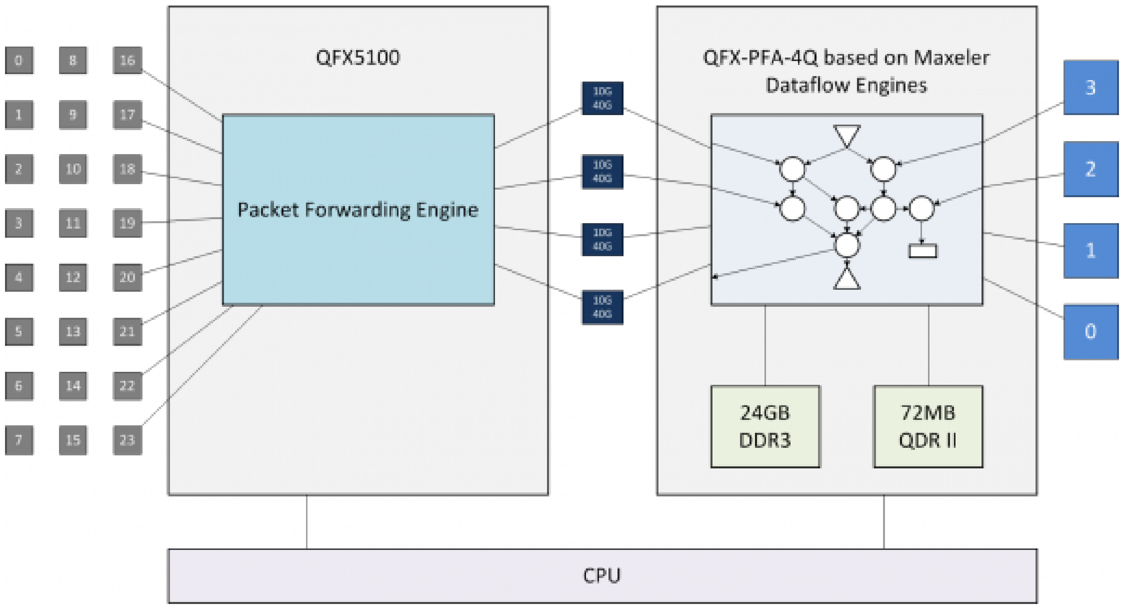


Figure 2.24: System Diagram Maxeler Platform [7].

The network ports on the right: Signify the ports connect directly to the FPGA chip.

2.4.3 FPGA development Cycle

Developing an FPGA design uses a development cycle. Figure 2.25 shows the development cycle that we use inside the thesis.

$$\begin{tikzcd}
\boxed{\text{Analyse}} \arrow[r] & \boxed{\text{Architect}} \arrow[r] & \boxed{\text{Program}} \arrow[r] & \boxed{\text{Generate Dataflow}} \arrow[d] \\
\boxed{\text{Custom HW}} & \text{OK?} \arrow[l, "\text{Yes}"'] \arrow[u, "\text{No}"'] & & \boxed{\text{Simulate and debug}} \arrow[ll] \arrow[ul]
\end{tikzcd}$$

Figure 2.25: Development Cycle of an FPGA implementation

Analyse: In the first phase of the development cycle, we analyze the problem that should be accelerated. The purpose of the analysis is to create a performance model. So we can find any problems before creating designs that would not work according to the platform specifications.

Architect: After creating the performance model, the next step is designing the system.

This step aims to solve or avoid any of the problems that the performance model indicated.

Program: Once the design is ready, both the CPU and DFE have to be programmed. The CPU application controls the DFE and steers the transfer of data among the CPU and DFE.

Generate Dataflow: After finishing the programming phase, the tools generate a dataflow design for our hardware. This design holds the bitstream that the DFE loads to perform the function.

Simulate and Debug: The generated solution will be simulated on its correctness and debugged to search for unexpected behaviour. If any undefined behaviour takes place, we go back to the programming step.

As a DFE is more beneficial for parallel applications, it does not mean that there might not be something to gain by moving a sequential function onto the DFE. Figure 2.26 shows an example where migrating a sequential function might be is justified.

In the left figure, the CPU has to execute a lengthy function 1, where it takes 1000s to perform the function. In the middle figure, option 1, we perform 'function 1' on the FPGA, which speeds up the whole algorithm by $1001/11 = 91$ times. In the figure on the right, option 2, we move function 2 to the FPGA, which increased the function's time. However, that is only at the kernel level. The overall system speeds up due to removing the FPGA transfer latency to the CPU, giving the system a speedup of $11/8 = 1.38$ times compared to option one.

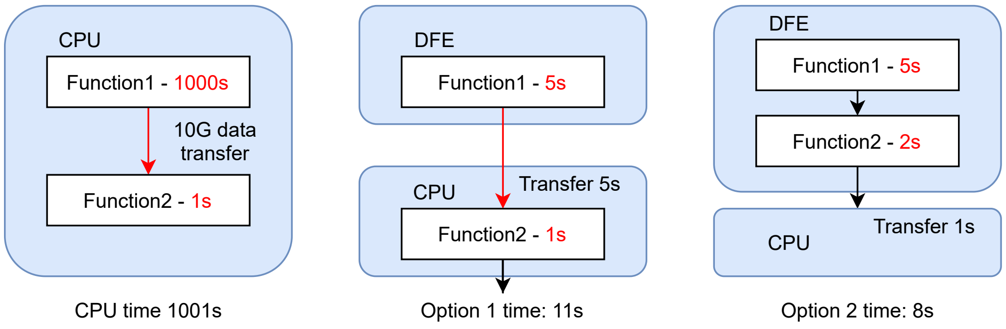


Figure 2.26: Methods to accelerate programs.

2.4.4 Off-chip Resources

This section focuses on the additional available resources of the DFE platform. These are interfaces that communicate with the Juniper system itself or its other resources, such as the CPU, memory and the network ports. Additionally, we define how DFE programmers address theser resources.

In Section 2.3.2.2 SRAM is introduced. This is a fast Static RAM located close to the FPGA. DFE programmers address this memory as FMEM, where it stands for Fast Memory.

2.4.4.1 Quad Data Rate Memory

Quad Data Rate (QDR) memory is a special memory that uses two different clock signals. During one clock cycle, both clocks transition from high to low and low to high once. On each of these clock transitions, one word transfers to memory, which results in a concurrent transfer of four words each cycle. There are two QDR memory interfaces available on the DFE platform. The DFE sends streams of commands, shown in Table 2.3, to communicate with the memory. QDR memory is exposed to the designer as QMEM, which is shorthand for Quick Memory.

Field	**Bits**	**Explanation**
Write address	21	Indicates where to write the data to.
Read address	21	Indicates where to read to data from.
Read Enable	1	Enables reading the data from the memory.
Write Enable	1	Enables writing the data to the memory.
Write Byte Enable	8	Allows for writing just specific parts of the write data
Write Data	144	The data to write to the memory

Table 2.3: Fields inside a QDR command stream.

QDR memory is a Static Random Access Memory (SRAM) with separate ports for concurrent reading and writing. Due to the above, reading and writing times are always consistent. Table 2.4 shows the specifications of the QDR memory on the DFE platform. Table 2.5 shows the physical implementation of the QDR memory, retrieved from the generated VHDL code of the Maxeler toolset.

Total Memory MBytes	72
Interfaces	2
Frequency MHz	500 [13]
Bits	18
Words a cycle	4

Table 2.4: Specifications of QDR interface.

Total Memory MBytes	72
Interfaces	2
Frequency MHz	350
Bits	36
Words a cycle	4

Table 2.5: Specifications of physical QDR.

With the results of Table 2.5, we can estimate the bandwidth limits of the QDR memory for one of the streams. Equation (2.1) shows the transfer rate for one of the streams. The maximum amount of bits/s we can transfer in one second for one interface is $50.4 \times 2 = 100.8$ Gbits/s. The maximum bandwidth of the entire QDR memory, which uses two interfaces, is $100.8 \times 2 = 201.6$ Gbits/s.

$$\begin{aligned} \text{Bandwidth}_{\text{QDR}} &= \text{Bits} \times \text{Words a Cycle} \times \text{Frequency} \\ &= 36 \times 4 \times 350 \cdot 10^6 \\ &= 50.4 \text{ Gbit/s} \end{aligned} \tag{2.1}$$

2.4.4.2 Double Data Rate Memory

Double Data Rate (DDR) memory is Dynamic Random Access Memory (DRAM) that is readily available in high quantities. Data transfers occur on either the falling edge or rising edge of the clock signal. However, compared to the QDR memory, only two words of data are transferred every cycle. Furthermore, DRAM is also not predictable compared to the QDR memory. Performing memory operations on these edges allows for data transfers at memory frequency. The QFX-PFA-4Q module has the memory specifications seen in Table 2.6.

Control bits	8
Data bits	64
# Interfaces	6
Frequency MHz	933

Table 2.6: Specifications of DDR3 interfaces.

RAM blocks (DIMMs)	3
RAM block size GByte	8
Frequency MHz	800
Total DDR3 Mem GByte	24

Table 2.7: Specifications of Large memory (LMEM).

Each memory interface has a maximum transfer rate of 72-bits in one clock transition. The maximum number of data bits transferred at once is $(64 \times 2)/8 = 16$ bytes per cycle. Table 2.7 shows the memory specifications of the system. As a limitation, each memory transfer is in bursts of 64-bytes of data. As 16 bytes transfer in one clock cycle, one burst would take four cycles in total.

$$\begin{aligned} \text{Bandwidth}_{\text{DDR}} &= \text{Bytes/cycle} \times \text{Frequency} \\ &= 16 \times 800 \cdot 10^6 \\ &= 12.8 \text{ GByte/s} \end{aligned} \tag{2.2}$$

Equation (2.2) shows that the total bandwidth of one memory module is 12.8 GByte/s. So as there are three memory blocks available, the total bandwidth at most is 38.4 GByte/s. DFE programmers can address this memory as LMEM, which stands for Large Memory.

2.4.4.3 Peripheral Component Interconnect Express

The CPU connects to the DFE platform through the third generation Peripheral Component Interconnect Express (PCIe) bus. The system uses this bus to read and write streams of data on the DFE platform. Table 2.8 shows the maximum transfer limit of PCIe IP blocks available for the FPGA. However, as the DFE platform uses a PCIe 2nd generation bus to connect with the CPU, it cannot reach this maximum.

	Theoretical max	Configuration Used
Gen	3	2
Lanes	8	4
Speed/Lane Gbit/s	8	5
Total Gbit/s	64	20

Table 2.8: Theoretical max PCIe vs actual max of PCIe.

2.4.4.4 Network interfaces

The DFE module has four QSFP-40G-SR4 Ethernet modules with a total transfer rate of 40 Gbit/s. Each module uses 4x10 GBASE-R lanes, with a bit rate of 10.3125 Gbit/s and a 64b/66b encoding. A lane, in this case, is a different wavelength in the same cable. So a 40 Gbit SR4 connection would have four different 10 Gbit wavelengths across the same cable. In 64b/66b encoding, the last 2 bits are control signals, with the remaining 64 bits being data. This type of encoding allows for ≈ 96.96% efficiency of the total line rate, which results in 10 Gbit/s per lane. So in the case of the 40 Gbit connection, four lanes are used. Table 2.9 shows a summary of the resulting data. The network interface for a 10 Gbit connection can increase its frequency to twice the original speed. Throughout this section, we will assume the original network frequency.

Lanes	bits/c	bytes/c	Frequency in MHz	Gbit/s
1	64	8	156.25	10
1	32	4	312.5	10
4	256	32	156.25	40

Table 2.9: Characteristics of the 10 Gbit and 40 Gbit interface.

Figure 2.27 shows the minimum and maximum size of a physical network frame. We use those two different sizes to get an idea about the minimum amount of cycles between two packets. Table 2.10 shows the result of the two packets.

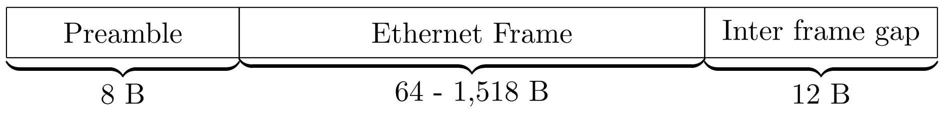


Figure 2.27: Physical frame received by the physical layer

	Min	Max
Bytes	84	1,538
Cycles @ 10 Gbit	10.5	192.25
Cycles @ 40 Gbit	2.625	48.0625

Table 2.10: The minimum and maximum cycles between packets

However, not all network traffic is interesting to us. As defined in Section 2.1 we only look for patterns inside TCP packets. Using the previous data as a basis and use stored network information, we can see the number of network packets that would approximately go over a network connection. That would allow for more accurate cycles between TCP packets.

TCP traffic This section will analyze network data received from the network telescope. The network telescope is at the TU Delft, whose primary purpose is to monitor the TU network. Table 2.11 shows the data received from several network dumps, including the percentage of TCP traffic.

Measurement	**Packets**	**AVG bytes/packet**	**TCP packets**	**% TCP**
Dump_1	2,060,491	65	1,963,017	95.27
Dump_2	1,774,052	80	1,544,477	87.06
Dump_3	2,146,477	61	2,012,088	93.74
Dump_4	1,939,514	71	1,764,164	90.96
Total	7,920,534	69	7,283,746	91.96

Table 2.11: Summary traffic network data.

The data in Table 2.11 includes the additional bits from the physical transfer over the wire. These bits are the preamble and inter-frame gap mentioned in Section 2.1.2. Monitor software does not include these bits, as they monitor from the Link Layer onward. Adding the 20 bytes to Table 2.11 results in an average packet size of 89 bytes. This change results in Table 2.12. Meanwhile, the average percentage of TCP traffic is 91%, using Table 2.11. With this information, we can estimate a worst-case number of clock cycles between two TCP packets. Table 2.13 represents the number of cycles together with the percentage of TCP traffic.

	Average
Bytes	89
Cycles @ 10 Gbit	11.13
Cycles @ 40 Gbit	2.78

Table 2.12: Cycles between packets.

	Average
10 Gbit	12.23
10 Gbit Rounded	12
40 Gbit	3.06
40 Gbit Rounded	3

Table 2.13: Cycles needed for TCP traffic.

Algorithm and Performance 3

This chapter explains the current scanner detection methods, and along with their main differences. Next, we describe the different algorithms mentioned in *Griffioen et al.*'s thesis to have a thorough understanding of how they function and find methods to simplify it for an FPGA based solution. Finally, we show a performance model that best fits the FPGA based on computation and memory limitations.

3.1 Scan detection methods

This section explains scanner detection methods. Afterwards, we go deeper into the method that looks for fingerprints inside the network traffic.

3.1.1 Connection based

M. Dabbagh et al. proposed a method of detecting slow scans by keeping track of three of the following distinct IP groups [14]:

Legitimate: This group stores the regular IP addresses that have not changed.

Suspicious: The group where IP addresses behaved suspiciously.

Scanner: Here are the confirmed scanner IP addresses.

This method stores K windows of T minutes long that contain three groups of IP addresses. The metrics used to separate every IP address inside a window are as follows:

N_{hc}**:** These are the number of half connections created, by not sending back an ACK packet after initializing the connection.

N_{closed}**:** These are the number of connection attempts that target a closed port.

N_{FIN}**:** These are the number of FIN packets received, without first establishing a connection.

These metrics combine in the following manner to put those addresses into groups:

$$state(IP) = \begin{cases} legitimate, N^{IP}_{closed} = 0 \; and \; N^{IP}_{hc} = 0 \; and \; N^{IP}_{FIN} = 0 \\ suspicious, N^{IP}_{closed} = 1 \; or \; N^{IP}_{hc} = 1 \; or \; N^{IP}_{FIN} = 1 \\ scanner, N^{IP}_{closed} > 1 \; or \; N^{IP}_{hc} > 1 \; or \; N^{IP}_{FIN} > 1 \end{cases}$$

An IP only belongs to the legitimate group when there was no abnormal behaviour in the previous windows. As there are K groups stored, there is a history of abnormal behaving IP addresses. When the method detects a new suspicious IP address, it compares it with past information. This method can only hold K windows. When it records a new window, it erases the oldest. The history will reduce the number of probes an adversary can send from the same source. As an example, there are ten windows of 5 minutes each. These will limit an adversary to one packet every 50 minutes if he wants to remain undetected, which would take a long time to scan all network ports.

3.1.2 Honeypot

The honeypot is a system that keeps itself vulnerable on purpose, to make itself look attractive to an adversary. As an example, a honeypot-based IDS stores of every suspicious IP it detects until it confirms it as regular traffic. This system stores these addresses for several days and looks at the number of occurrences of that address. When the number exceeds a certain threshold, the system detects it as scanner traffic. An example of this would be ten suspicious requests in 3 days [15].

3.1.3 XOR Patterns

Griffioen et al.[11], after a hint from *Dainotti et al.* [16], found a method that detects patterns that scanners encode in their network traffic. The main difference between this and the previous methods is that it looks for patterns instead of waiting for suspicious behaviour.

3.1.3.1 Reasons for patterns

Scanners search for open ports on multiple hosts on the Internet. The adversary has to use its resources to keep track of every scan probe, while the probe might never yield a response. Thus, scanners encode a pattern inside their traffic to distinguish a scan response from regular network traffic. Figure 3.1 represents this behaviour, where a scanner does not accept any traffic that does not contain a pattern.

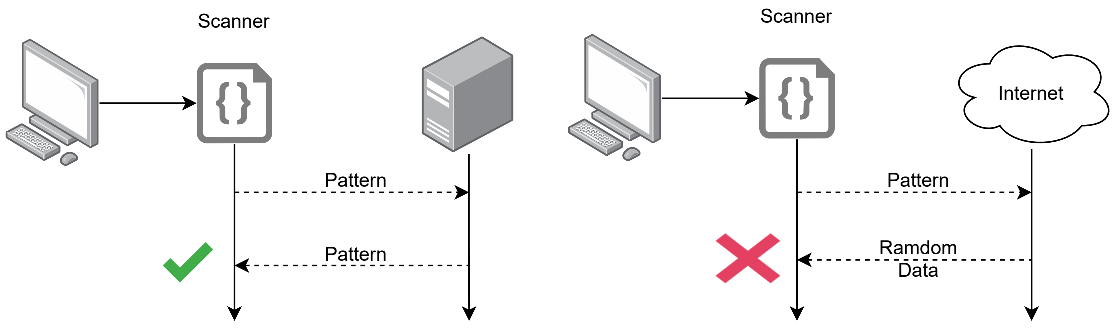


Figure 3.1: Scanner Behavior.

A patterns main requirement is that it should return in a scan probes response. This requirement is why a scanner uses network fields that are the same when send and received back, which limits a scanner to the fields in Table 3.1. The reason behind these fields are as follows:

Source address and port: These fields define where the scanner traffic comes from and where it should return. Because it has to return, a scanner cannot set an arbitrary address, as he needs to retrieve the response.

Destination address and port: These two specify the fields that define the scanning target and the service.

Sequence number: This is the number chosen by the client in regular TCP traffic and comes back as the acknowledgement number inside the response.

The sequence number is the largest number a scanner can choose, making it the perfect field to encode the pattern.

Send	**Receive**
Source address	Destination address
Source port	Destination port
Destination address	Source address
Destination port	Source port
Sequence number	Acknowledgement Number+1

Table 3.1: Send and receiving scanner traffic.

3.1.3.2 Different patterns

Griffioen et al. described different types of patterns that scanners use. This section explains those types.

The first pattern chooses a random sequence number. With this method, scanners would only need to check the acknowledgement number in their response. However, there are some downsides.

1. To reduce the number of connections to keep track of, the sequence number would remain the same for every probe. However, receiving probes from one source that has all have the same sequence number is very suspicious.
2. Otherwise, the scanner has to keep track of all the used sequence numbers, which still increases the adversary's resources and would defeat the reason for using patterns in the first place.

The second pattern shuffles specific parts of the fields. For example, we divide the source address into chunks of 4 bits and shuffle them. The result would be a relatively random-looking number, where the scanner only has to check if the returning packet contains the same shuffled field.

The third pattern uses a combination of the fields shown in Table 3.1. This pattern is less visible and allows for a more stealthy approach. There are multiple methods to combine them, addition, multiplication, using shifted versions of the fields. However, we are interested in a combination of shifts and XOR operations, which create a random number. Equation (3.1) is an example of a strategy that only encodes the source and destination ports into the sequence number. Now the whole 32-bit field is filled with those two fields.

Equation (3.1) shows an example of a strategy that uses the source and destination ports into the sequence number.

$$Sequence\ Nummber = (Port_{src} << 16) \oplus (Port_{dest}) \tag{3.1}$$

Additionally, scanners can choose to use a secret value, also called a Key, to obfuscate the pattern. The generic pattern that shows all possible combinations is Equation (3.2). The fields of the patterns are shifted k_i spaces to the left in the range of $[0, 32]$. Afterwards, they combine into one sequence number. That number would look random but is, in essence, the pattern we aim to find.

$$\begin{aligned} Sequence\ Number = {} & (Addr_{src} \ll k_0) \oplus (Addr_{dest} \ll k_1) \\ & \oplus (Port_{src} \ll k_2) \oplus (Port_{dest} \ll k_3) \oplus Key \end{aligned} \tag{3.2}$$

3.2 XOR detection methods

This section goes into depth about the XOR-based detection methods for finding fingerprints. First, we explain the XOR operation. Afterwards, we go into depth about the different methods to find XOR-based fingerprints.

3.2.1 Exclusive-OR operation

An Exclusive-OR (XOR) operation is an operation that is equal to adding two bits modulo 2. When the sum of the bits is odd, it results in a '1'. Otherwise, the result is '0'. A XOR operation is denoted with $\oplus$, Table 3.2 shows the truth table of a 2-bit XOR operation.

This operation applies to every bit position of a number, so with 16-bit operators, we get Equation (3.3).

$$\begin{array}{rl} & \{1001\ 1010\ 1110\ 0101\} \\ \oplus & \{1111\ 1101\ 1001\ 1110\} \\ \hline & \{0110\ 0111\ 0111\ 1011\} \end{array} \tag{3.3}$$

The XOR operation has the following few properties:

$$X \oplus 0 = X$$

Inputs		Output
X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

Table 3.2: XOR Truth table.

When XORing something with 0, every odd bit of X is still odd, so nothing changes.

$$X \oplus X = 0$$

When XORing the same numbers together, the result becomes zero due to having an even number of bits at every bit position.

$$\begin{aligned} X \oplus Y &= N \\ X \oplus N &= Y \\ Y \oplus N &= X \end{aligned} \tag{3.4}$$

When XORing two numbers, we can recalculate its inputs from the output if one of the operators is known. In this case, X and Y are random numbers, which result in the number N. Afterwards, X and Y can be retrieved from N, if either X or Y is known. If there is no knowledge about the inputs of N, there is no method to retrieve them. This property preserves randomness, meaning that XORing two random numbers result in another random number.

3.2.2 Brute-force

The brute-force approach is the most straightforward approach to find fingerprints. This performs every possible shift combination $[k_0, k_1, k_2, k_3]$ for their respective fields, as shown in Algorithm 1. The range for one shift is $[0; 32]$, meaning there are a total of 33 different values, where Table 3.3 show the unique shift values.

There is a total of $1,185,921$ operations that are both independent and thus parallelizable. The drawback is that these operations have to be performed in a limited amount of cycles, resulting in a bottleneck. For example, if we take the maximum cycles between packets from Section 2.4.4.4, which is approximately 12; $1,185,921/12 = 98,826.75$ operations have to be performed in one cycle. The other drawback of this method is that it does not take the Key value into account. When a scanner uses a Key, it obscures the pattern. However, in that case, multiple patterns might be detected. Either this is a coincidence by random noise in the network or the Key value.

3.2.3 Related packets

Besides brute-forcing the fingerprint, there is also the option to use two packets with fields in common. When XORing those together, it removes the same fields from the

```
**Input:** Network Fields
**Result:** List of Fingerprint
Fingerprints = [];
foreach $k_0$ in $[0;32]$ do
    foreach $k_1$ in $[0;32]$ do
        foreach $k_2$ in $[0;32]$ do
            foreach $k_3$ in $[0;32]$ do
                $XOR_{Addrs} = (Addr_{src} \ll k_0) \oplus (Addr_{dest} \ll k_1)$;
                $XOR_{Ports} = (Port_{src} \ll k_2) \oplus (Port_{dest} \ll k_3)$;
                if $(XOR_{Addrs} \oplus XOR_{Ports}) == Sequence\ Number$ then
                    Fingerprints.add($k_0$, $k_1$, $k_2$, $k_3$);
                end
            end
        end
    end
end
```

Algorithm 1: Brute force pattern finding algorithm.

Shift	Explanation
0	The field isn't changed
32	The field has become 0
Other	The field is a shifted version of its original

Table 3.3: Special shift values.

pattern. This operation reduces the complexity from $1,185,921$ operations to a range of $[33, 35,937]$. As an example, XORing Equation (3.5) and Equation (3.6) together results in Equation (3.7).

$$\begin{aligned} Sequence\ Number_a = {} & (Addr_{src} \ll k_0) \oplus (Addr_{dest_a} \ll k_2) \\ & \oplus (Port_{src_a} \ll k_2) \oplus (Port_{dest_a} \ll k_3) \end{aligned} \tag{3.5}$$

$$\begin{aligned} Sequence\ Number_b = {} & (Addr_{src} \ll k_0) \oplus (Addr_{dest_b} \ll k_1) \\ & \oplus (Port_{src_b} \ll k_2) \oplus (Port_{dest_b} \ll k_3) \end{aligned} \tag{3.6}$$

$$\begin{gathered}
Sequence\ Number_a \oplus Sequence\ Number_b \\
\equiv \\
(Addr_{src} \ll k_0) \oplus (Addr_{dest_a} \ll k_1) \oplus (Port_{src_a} \ll k_2) \oplus (Port_{dest_a} \ll k_3) \oplus \\
(Addr_{src} \ll k_0) \oplus (Addr_{dest_b} \ll k_1) \oplus (Port_{src_b} \ll k_2) \oplus (Port_{dest_b} \ll k_3) \\
\equiv \\
(Addr_{src} \oplus Addr_{src}) \ll k_0 \oplus (Addr_{dest_a} \oplus Addr_{dest_b}) \ll k_1 \oplus \\
(Port_{src_a} \oplus Port_{src_b}) \ll k_2 \oplus (Port_{dest_a} \oplus Port_{dest_b}) \ll k_3 \\
\equiv \\
(Addr_{dest_a} \oplus Addr_{dest_b}) \ll k_1 \\
\oplus (Port_{src_a} \oplus Port_{src_b}) \ll k_2 \\
\oplus (Port_{dest_a} \oplus Port_{dest_b}) \ll k_3
\end{gathered} \tag{3.7}$$

As XORing packets remove fields from the equation if the pattern contains a Key, it would remove it. However, XORing two related packets give two problems. Storing the packets in memory and comparing only the packet with the most substantial relation.

3.3 Performance model

This section focuses on the different methods to implement the XOR-based pattern-finding algorithm. First, we describe the operation and memory limitations and make a decision based on them. Afterwards, we discuss the chosen algorithm and give alternative approaches to the problem.

3.3.1 Limits

First, we define the limits to see which of the algorithms described here would fit on the DFE platform. These are the maximum amount of performable calculations and memory-based limitations. With the pattern finding methods defined in the previous section, we aim to answer the following questions:

1. What is the maximum amount of parallel operations?
2. How many packets fit in the DFEs large memory?
3. How long does it take before the memory overflows?
4. What is the required memory bandwidth?
5. How should the fingerprints be stored?
6. How many fingerprints can be stored in the QMEM?

After answering these questions, we decide on which algorithm to implement.

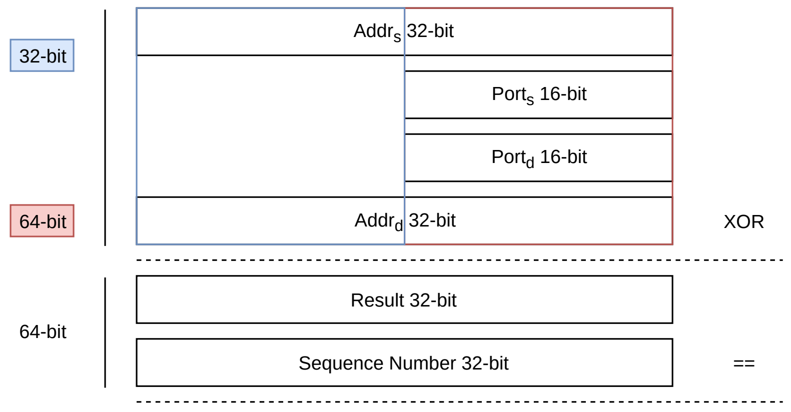


Figure 3.2: XOR and validate.

3.3.1.1 Maximum operations

Figure 3.2 shows the number of bits and the operations needed for XOR-based pattern finding. As previously discussed in Section 2.3.2.1, an ALM uses 8-input adaptable LUTS, which is the main computational resource.

The calculation of the possible computations requires us to know the ALMs of one operation. From Figure 3.2 we see that the XOR operation needs 96-bits of input data and an additional 64-bits for validation. By mapping those bits on ALMs, the XOR operation costs approximately $96/(8 \text{ bits}) = 12$ ALMs. The validation operation adds $64/(8 \text{ bits}) = 8$ ALMs for comparing the two values.

There are a total of $359,200$ ALMs available on the FPGA. Taking the previously calculated ALM cost of the XOR and validation operation, it costs 20 ALMs. With those 20 ALMs, the maximum is $359,200/20 = 17,960$ operations on each clock cycle. That does not reach the target performance of $98,826.75$ operations a cycle. However, there are optimizations possible to get more operations a cycle.

3.3.1.2 Maximum packets in memory

There are 24 GBytes of LMEM available to store the fields, where each set is 128-bits or 16 bytes of memory. Equation (3.8) calculates the maximum number of fields that fit inside the memory.

$$\begin{aligned} \text{Total Packets} &= \text{Total Memory/Bits to store} \\ &= (24 \times 2^{33})/128 \\ &= 1,610,612,736 \end{aligned} \tag{3.8}$$

If we use the source address as an identifier to the memory, there will be a problem as there are $2^{32} = 4,294,967,296$ possible source addresses available, where 37.5% fits inside the memory. Using a hash value of the source address would keep closely related packets together.

3.3.2 Burst of memory

The LMEM transfers data in bursts. The size of these bursts depends on the number of memory modules used. The burst size has a maximum burst size of 192 bytes, which uses all 6 DIMMs. More information on the DIMMs can be found in Section 2.4.4.2. In that case, all 24 Gbyte gets used. These transfers are called bursts. One burst can hold 12 packets at a maximum. Equation (3.9) shows the number of bursts that the memory can hold.

$$\begin{aligned}\text{Total memory bursts} &= \frac{\text{Packets in memory}}{\text{Fields in burst}}\\ &= \frac{1,610,612,736}{12}\\ &= 134,217,728\\ &= 2^{27}\end{aligned} \tag{3.9}$$

An LMEM interface addresses memory in bursts. For example, when requesting address 0 of the LMEM, it would return the first 192 bytes. Coincidentally requesting address 1 returns the next 192 bytes. Because the burst length limits the LMEM, and there are 2^{32} possible IP addresses. Its number of bursts limits the number of unique addresses of the memory.

3.3.2.1 Time to fill

When the memory is full, every new packet replaces one inside the memory. We need to know how long it takes to fill the memory, and packets are lost. First, we use the cycles between packets in the worst-case scenario. The assumption here is that one TCP packet arrives once every 12 clock cycles with a 10 GBits/s network interface operating at a 156.25 MHz frequency, as shown in Section 2.4.4.4. By multiplying the total amount of packets that fit inside the memory with those 12 cycles, we get the number of cycles it takes to fill the memory. Afterwards, we divide cycles by the network interface frequency to get to the seconds. Equation (3.10) shows to calculate it with a 10Gbit/s network interface.

$$\begin{aligned}\text{Time To Fill} &= \frac{\text{Total Packets} \times \text{Cycles between packets}}{\text{Frequency}}\\ &= \frac{\textit{Equation } (3.8) \times \text{Cycles between packets}}{\text{Frequency}}\\ &= \frac{1,610,612,736 \times 12}{156.25 \cdot 10^6}\\ &= \frac{19,327,352,832}{156.25 \cdot 10^6}\\ &\approx 123.7\ s\end{aligned} \tag{3.10}$$

3.3.2.2 DDR memory Bandwidth

We need to know how much bandwidth it uses before we know if a memory-based design is possible. For writing, in the worst-case scenario, a packet arrives every 12 cycles, which writes 16 bytes to memory. Equation (3.11) calculates the bandwidth for a 10 Gbit connection, where a 40 Gbit connection is four times faster.

$$\begin{aligned}\text{Bandwidth} &= (\text{Frequency}/\text{Cycles between packets}) \times \text{bytes to store} \\ &= (156.25\text{ MHz}/12) \times 16 \\ &\approx 208.33\text{ MByte/s}\end{aligned} \tag{3.11}$$

However, in the worst-case scenario, the incoming packet has to be compared to all the packets in memory. The maximum read speed, defined in Section 2.4.4.2 is 38.4 GByte/s. A maximum of 38.4 GByte/16 bytes= $2,400,000,000$ sets of fields can be read in one second. Meanwhile, there is space for $1,610,612,736$ fields fit in memory. Memory is read and written in bursts of 192 bytes, which contain 12 sets of fields. Equation (3.12) shows the DDR bandwidth calculation.

$$\begin{aligned}\text{Bandwidth}_r &= \text{Burst in memory} \times \text{Bytes in burst} \\ &= 2^{27} \times 192 \\ &= 134,217,728 \times 192 \\ &\approx 25.77\text{ GByte/s}\end{aligned} \tag{3.12}$$

3.3.2.3 Compare incoming packets to all packets in memory

When comparing packets, the worst-case scenario occurs when the matching fields are at the end of the memory. However, while it is searching for a matching packet, additional packets arrive. These also need to be validated for patterns. As previously mentioned, LMEM uses bursts of 192 bytes to read and write to the LMEM. During this scenario, there is only time to request 12 memory bursts before the next packet arrives. The amount of bursts needed to read the memory entirely is $134,217,728$. So, while waiting to find that one set, $134,217,728/12 \approx 1,184,810.66$ packets arrive. That many packets make it impossible to compare each incoming packet to one specific set.

3.3.2.4 Compare incoming packets using hashing

Using a hashing algorithm would allow us to retrieve data from LMEM without comparing it to the whole memory. By hashing the source address, it keeps the related packets closer together. However, there is a tradeoff. The number of unique hashes is the same as the number of bursts, which allow for a total of 2^{27} unique memory addresses. The ratio of unique memory addresses compared to the 2^{32} source addresses is $2^{27}/2^{32} = 1/32$. Therefore, it is reasonable to assume that more than two source addresses will result in the same hash value, called a collision. As an upper bound for the seconds, we use the time it takes to fill the memory as shown in Equation (3.10). The number of packets required to do this is coincidental 12/32 of the IP address space.

There is a slight chance of $1/2^{27}$ that the following packet has the same hash as the previous one. Using the 31/32 chance of a different IP address triggering the collision, we get a chance of $(31/32)/(2^{27}) \approx 7.21 \cdot 10^{-8}$. With that number, we can be almost sure to have a collision after $1/7.21 \cdot 10^{-8} = 138547332.129 \approx 2^{27}$ packets. We require 12 of tose collisions, which requires $138547332.129 * 12 = 1,662,567,985.548$ packets. That number is higher than the maximum amount of packets inside the memory, but it is close. With that information, we can conclude that hashing does not add time to the 123.7 seconds. However, that only works with a uniform distribution of both the hashing function and incoming IP addresses.

3.3.2.5 Using QDR memory

When the algorithm finds a fingerprint, it needs to be stored. We chose to use the Quad Data Rate (QDR) memory, also called QMEM, for this purpose due to its fast read and write speeds.

One fingerprint has four values in the range of $[0; 32]$. Using that pattern as a unique combination to the memory, we get a 21-bit address, the same width as the QDR memory address. It fits because $\log_2(33^4) \approx 20.18$, where every fingerprint creates a unique index.

The total required memory for one module is $144 \times 33^4 = 170,772,624$ bits, approximately 20.3 MByte. In total, there are 72 MBytes available, separated into two modules. 20 MByte utilizes a modules memory capacity of approximately $20.3/(72/2) * 100 = 56.5\%$, which means that not all the QMEM gets utilized.

One option for the remaining memory is to use more bits for each fingerprint. For example, instead of 144 bits for each fingerprint, 216 bits get used with one module. Two modules would us 432 bits for each fingerprint, allowing for $432 \times 33^4 = 512,317,872$ bits ≈ 61.07 MByte of utilization. However, this would create a more complex design. This case requires additional logic to keep track of which addresses in the QMEM hold the fingerprint data.

Another option would be double buffering, a technique to read from one buffer while writing to the other. At a certain point, the buffer that gets read gets swapped. That would require $20.3 \times 2 \approx 40.7$ Mbyte of space for one module, which is too large. The fingerprint would need to be reduced from 144 to 127 bits for every module for it to work. It adds additional resources required to account for the location of those fingerprints. However, a double buffer does not give value to the system as a fingerprint could arrive at every cycle.

Consider the following scenario. A fingerprint gets read while that same data is getting processed. While the data still needs to be written to the memory, the QMEM does not see this and returns the same processed data. This behaviour causes inconsistent memory, as our processed data would get overwritten. Double buffering would not aid in that issue as the same buffer needs to get continuously read and written.

As discussed in our situation, a match could arrive at every cycle. Equation (3.13) shows the required bandwidth for one QMEM module. The total bandwidth is below the limit of 144 Gbit/s for one module, utilizing $45/144 * 100 \approx 31.25\%$ of the available bandwidth.

$$\begin{aligned}\text{Bandwidth} &= 2 \times (\text{QDR bits}) \times \text{Frequency} \\ &= 2 \times 144 \times 156.25 \text{ MHz} \\ &= 45 \text{ Gbit/s}\end{aligned} \tag{3.13}$$

With two modules, there are 288 bits of memory to store information about the matched pattern. We used the probabilistic bloom filter structure to use the 144 data bits of each QMEM interface fully. With a bloom filter, the number of false positives increases the more data it contains. However, it does not allow for false negatives. This filter aims to store the source address of the pattern that gets recognized efficiently to understand which pattern gets triggered the most and by whom.

3.3.2.6 Conclusion

In the section above, we discussed the limits of the system using its resources. Our use case is to find a pattern inside the packet as the FPGA receives it.

In A memory-based solution, there would be much reading and writing of data to the memory. Meanwhile, from our calculations, it would get filled in approximately 123.7 seconds in the worst-case scenario. Afterwards, there is data loss when new packets arrive. The new packet might remove data with a specific pattern inside it, which is a problem as in that specific case, a packet with that source address would arrive it gets removed. If a very slow scan scans with one packet every 5 or 10 minutes, there would be no gain to store it in memory as it would disappear before using it.

Meanwhile, the incoming packet has a one-to-many relationship to find an optimum packet. Reading the memory linearly would take a lot of time and bandwidth, where many packets arrive when searching for a matching packet. This problem gets mitigated by using the hash value of the source address, which puts related packets closer together. However, there is a mismatch between the number of possible source addresses and the number of bits used for the LMEM memory space. The ratio between a hash value and source address is 1:32. However, it would have no time impact if the hashing algorithm and packets have a uniform distribution. That probably is not the case, so there would be a time cut. Therefore, we chose to use the brute-force method and find optimizations to increase operations per clock cycle.

3.3.3 Brute-force scaling

This section explains variations on the brute-force algorithm and looks at how the FPGA resources scale compared to the number of operations. We synthesize these designs for the FPGA to get a better representation of the resource costs. Afterwards, we compare them and select the version that we implement on the actual FPGA.

3.3.3.1 Naive brute-force pattern-finding function

The naive function is the standard brute-force algorithm defined in Section 3.2.2 directly implemented on hardware. Figure 3.3 is the used model, while Table 3.4 shows the approximated resource usage at every part. Equation (3.14) and Equation (3.15) shows our approximation of the resources, where r represents the number of XOR operations.

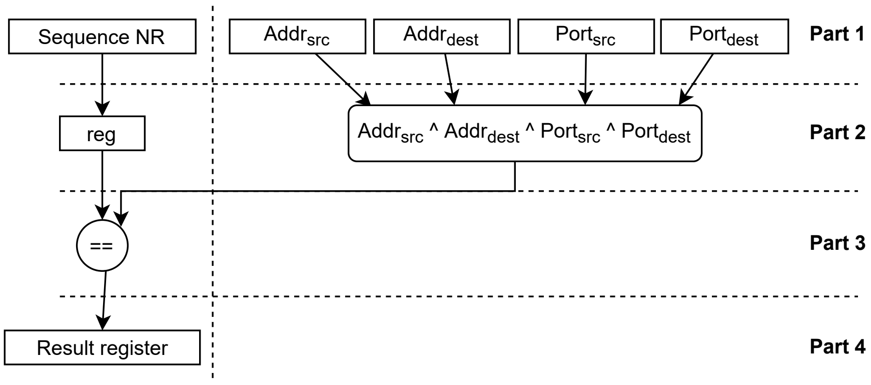


Figure 3.3: Naive Model.

Parts	**ALMs**	**Register - bits**
1	0	128
2	$12 \times r$	32
3	$8 \times r$	-
4	0	r

Table 3.4: Resources Naive.

$$ALMs = 20 \times r \tag{3.14}$$

$$Register - bits = 5 \times 32 + r \tag{3.15}$$

Utilization Scale In Figure 3.4, we compare the scaling of our estimation and the actual costs, where the goal is to get a better understanding of how the resources scale. The average ALM cost for each operation is $\approx$ 32 ALMs every operation, which is 12 ALMs higher than expected. We assume that memory configured ALMs are used as registers. Furthermore, the tools insert additional register for intermediate values for the XOR function and the sequence number. These registers add at least 64 register bits/8 = 8 ALMs every computation. From now on, we keep this behaviour in mind during resource calculation.

3.3.3.2 Pipelined brute-force pattern-finding function

The second option implements the brute-force algorithm using a pipelined solution to have more throughput. This approach requires to change the algorithm to allow for more pipelining and reduce resource costs.

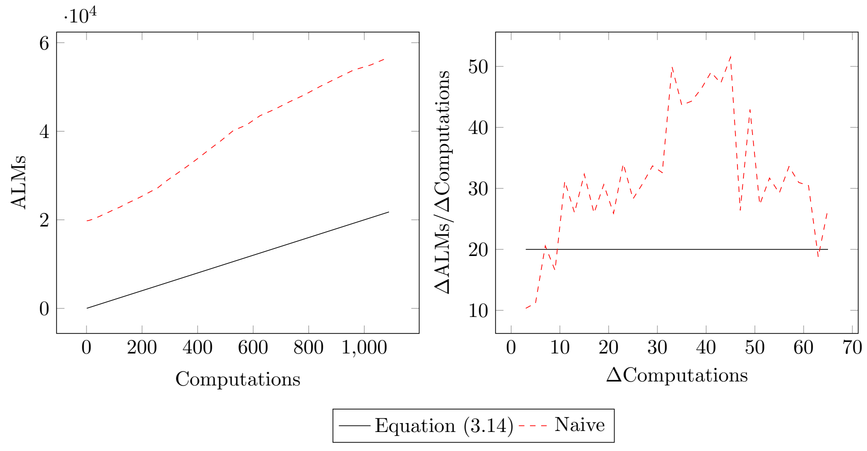


Figure 3.4: Naive: Theoretical vs Actual.

Changing the algorithm The main idea behind the changed algorithm is to compare using a shifted field instead of the sequence number. In this case, there is only one expensive XOR operation, while there are many comparison operations. In Equation (3.16), $Addr_{src}$ is compared k times to one XOR operation.

$$\begin{aligned} Sequence\ Number &= (Addr_{src} \ll k_0) \oplus (Addr_{dest} \ll k_1) \\ &\oplus (Port_{src} \ll k_2) \oplus (Port_{dest} \ll k_3) \\ (Addr_{src} \ll k_0) &= Sequence\ Number \oplus (Addr_{dest} \ll k_1) \\ &\oplus (Port_{src} \ll k_2) \oplus (Port_{dest} \ll k_3) \end{aligned} \tag{3.16}$$

To illustrate the method even further, Algorithm 2 shows how to derive k results from one XOR operation. In the algorithm, we substitute the XOR operation for R, as shown in Equation (3.17). Now R is compared to all the values of k_0, which moves the operation costs to compare two registers.

$$R = Sequence\ Number \oplus (Addr_{dest} \ll k_1) \oplus (Port_{src} \ll k_2) \oplus (Port_{dest} \ll k_3) \tag{3.17}$$

```
Input: Network Fields
Result: List of Fingerprint
Fingerprints = [];
$R = Addr_{dest} \oplus Port_{src} \oplus Port_{dest} \oplus Sequence\ NR$;
foreach $k_0$ in $[0; 32]$ do
    if $Addr_{src} \ll k_0 == R$ then
        Fingerprints.add($k_0, \cdots$);
    end
end
```

Algorithm 2: Pipelined algorithm.

The architecture As stated in Section 2.4.1.1, pipelining allows for more throughput through a function. Figure 3.5 shows the model of Algorithm 2. In the model, we only perform the XOR operation once. Whereafter, we compare it in each of the k stages of the pipeline with a shifted version of $Addr_{src}$.

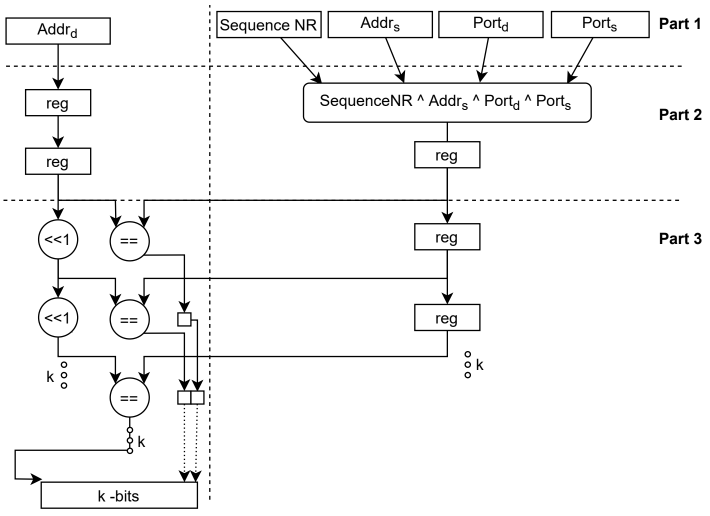


Figure 3.5: Pipelined Model.

Each stage uses the previous values as its inputs and stores the outputs in registers. These registers cost ALMs, where we calculate the cost of ALMs in Equation (3.18) and Equation (3.19). In these formulas, r is the number of XOR operations.

Parts	**ALMs**	**Register - bits**
1	0	128
2	$16 \times r + 8$	0
3	$20 \times k$	$r \times \sum_{i=1}^{k} i$

Table 3.5: Resources Pipelined.

$$\begin{aligned} ALMs &= 16 \times r + 20 \times r \times k + 8 \\ &= 4 \times r \times (4 + 5 \times k) + 8 \end{aligned} \tag{3.18}$$

$$\text{Register bits} = 128 + r \times (\sum_{i=1}^{k} i) \tag{3.19}$$

Utilization Scale The average of our assumed usage is ≈ 21, as shown in Figure 3.6. These designs keep the number of repetitions equal to k. So the pipelined and naive function perform the same number of calculations. The results show a linear growth in resource usage. However, there is a spike of resources around the thousand operation mark. We assume that the tools inserted additional resources to meet the timing constraints. The FPGA tools create a register-tree as compensation, which increases the number of resources. The solution is to use smaller sets of computations to avoid that peak, which keeps the ALM for each additional XOR operation to 27.

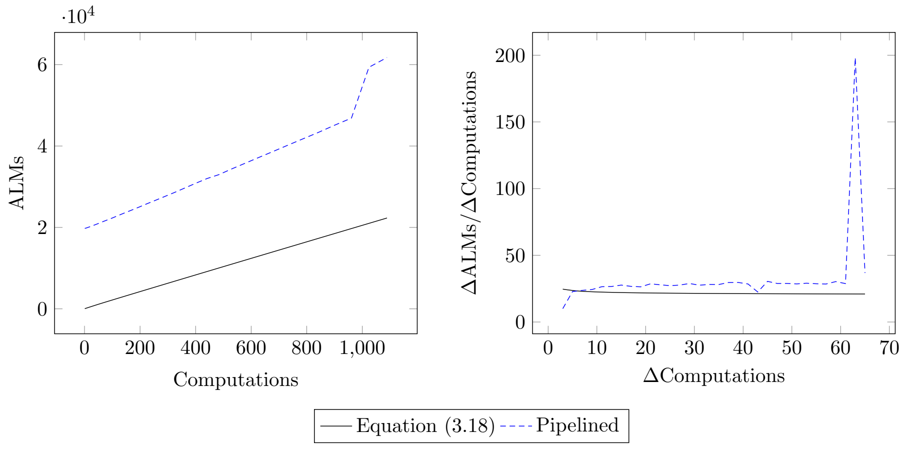


Figure 3.6: Pipelined: Theoretical vs Actual.

3.3.3.3 Reduced pattern-finding function

With the pipelined pattern-finding function, we changed the algorithm to reduce the costs of each operation. More operations are possible by removing unnecessary registers.

Architecture Figure 3.7 optimizes the pipelined pattern-finding function. It does so by performing k operations immediately, instead of in k stages.

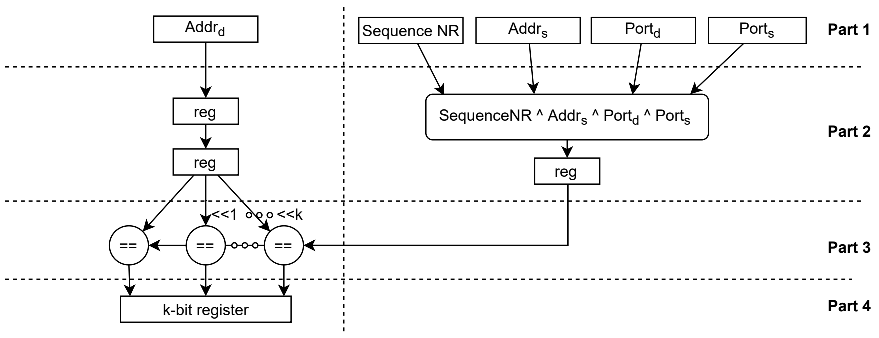


Figure 3.7: Reduced Model.

Table 3.6 the estimated resource costs for the model. Equation (3.20) and Equation (3.21), use the table to create a formula for how the operations approximately scale, where r represents the number of XOR operations peformed.

Parts	**ALMs**	**Register - bits**
1	0	128
2	$16 \times r + 8$	-
3	$8 \times k \times r$	-
4	0	k

Table 3.6: Resources reduced implementation.

$$\begin{aligned} ALM &= 16 \times r + 8 \times r \times k + 8 \\ &= 8 \times r \times (2 + k) + 1 \end{aligned} \tag{3.20}$$

$$REG - bits = 128 + k \tag{3.21}$$

Utilization Scale The average assumed ALM cost for each operation in our model Figure 3.8 is 9 ALMs, where we take $r = k$. Interestingly enough, the ALM cost of the synthesized design is lower than the initial estimation. However, similar to the pipelined pattern-finding function, a significant resource increases around the thousand operation mark. We use the same assumption that the increase occurs due to timing constraints. Using the same smaller sets, as with the pipelined pattern-finding function, gives each additional XOR operation a cost of 7 ALMs.

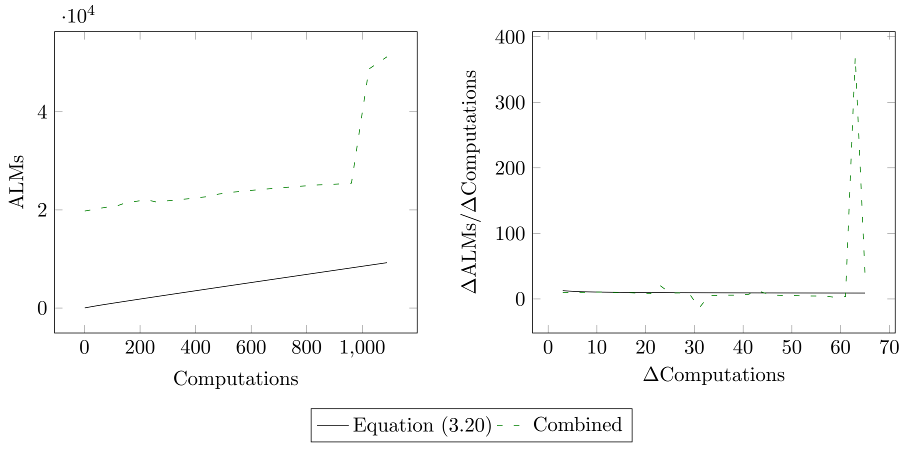


Figure 3.8: Combined: Theoretical vs Actual.

3.3.3.4 Conclusion

Figure 3.9 shows the differences between all the synthesized approaches, where Table 3.7 shows the differences between operation costs. The figure shows quite clearly that using the reduced pattern-finding function results in more operations per cycle. The total

ALMs that are available is $359,200$. With average operation costs of 7, it performs a total of $359,200/7 \approx 51,314.28$ operations each clock cycle. This is less than the $98,826.75$ operations per cycle that are required. However, there are additional operations needed on the FPGA to make a functional design. As an example, there need to be functions to parse network traffic and store the fingerprints somewhere, which reduces the number of operations.

Implementation	Assumed	Actual
Naive brute-force pattern-finding function	$20 \times x$	$31 \times x$
Pipelined brute-force pattern-finding function	$21 \times x$	$27 \times x$
Reduced brute-force pattern-finding function	$8 \times x$	$7 \times x$

Table 3.7: Resource comparison.

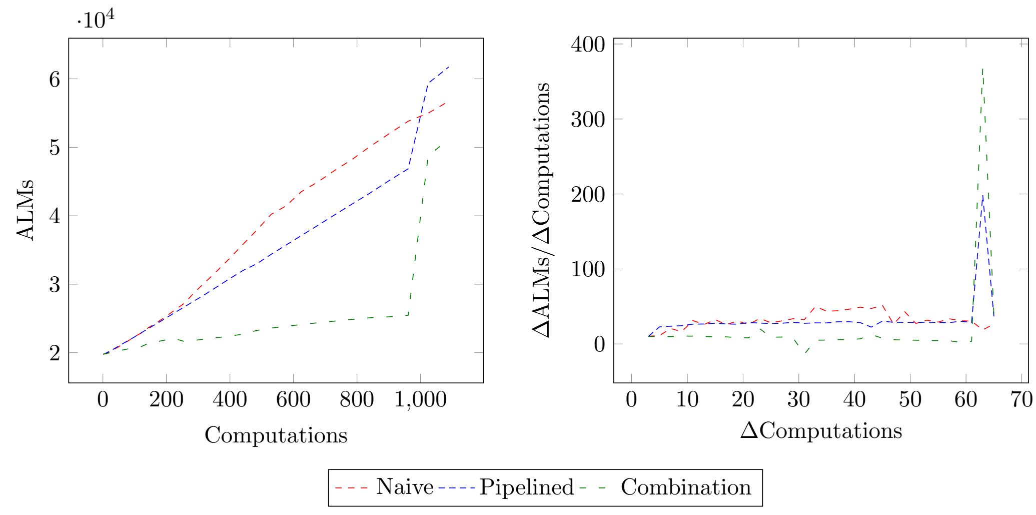


Figure 3.9: Comparing all synthesized results.

3.3.4 Adjusting our target

In Section 3.3.3.4 we found that even with the reduced brute-force pattern-finding function, it will not be possible to reach the targetted bandwidth of 10Gbit/s. We found that the system could reach half of the targeted bandwidth at most. Our problem with the used method is LUTs within the QFX-PFA-4Q system to perform all computations in time using the brute-force method.

As our system does not have enough resources, we could consider a high-performance FPGA system with a different chip, such as the Alveo U280 card, using the VU37P chip, or the VU9P FPGA chip.

Both of these chips use Xilinx' UltraScale+ technology inside their Lookup-tables (LUTs) This technology uses a 6-input 1-output LUT with a shared input mode. Here it is used as a 2x5-input 2-output LUT. [17] The Alveo U280 card contains $1,304$K of these LUTs, while the VU9P chip contains $1,182$K. [18] Our system has $359,2$K 8-input

LUTs and could perform at most half of the operations. Using the above mentioned FPGA's would effectively solve the resource issue.

The U280 card mentioned above uses High-bandwidth Memory (HBM). This type of memory uses silicon stack technologies to keep the memory and FPGA in the same chip package. It has a lot less latency due to being closer to the chip and allows for bandwidths up to 460 GByte/s. [19] Compared to the DDR3 Memory bandwidth of 12.8 GByte/s, it is $460/12.8 = 35.9$ times faster. Using HBM possibly allows for a different solution than our current one using the QDR memory. It might allow for a more hybrid solution, storing more packet data or fingerprint information than the current solution.

As we will not reach the desired bandwidth with the current system, we will change the target. Our new target is to optimize the reduced brute-force pattern-finding function and perform as many operations as our targeted system allows.

3.3.5 Storing fingerprints

After detecting a fingerprint, there needs to be a method of storing it. To accommodate this function, we decided to use the Quad Data Rate (QDR) memory, as it allows for a consistent amount of cycles for reading and writing. First, we define the number of cycles it takes between QDR requests. Afterwards, we explain the data structure we use, namely the bloom filter.

3.3.5.1 QDR memory cycles

To test the QDR memory latency, we created the design shown in Figure 3.10. In the design, when the kernel sends a command to the QDR memory, it stores a counter value inside a queue. Once we receive a response from memory, we remove the first item from the queue and calculate the clock cycles between them.

Table 3.8 shows the results at two different frequencies. Using these cycles, we know how long it takes for data to return from and to memory.

Frequency MHz	Latency Cycles
156.25	17
312.5	29

Table 3.8: Latency of QDR memory at different frequencies.

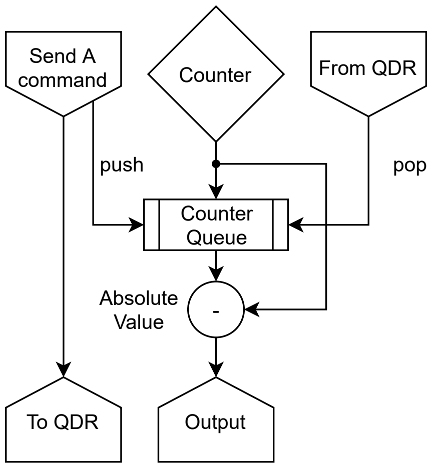


Figure 3.10: Design to calculate the latency of the QDR.

3.3.5.2 Bloom filter implementation

The bloom filter is a probabilistic memory structure, where the number of false positives increases the more data it contains. However, it does not allow for false negatives. This filter aims to store the patterns that get recognized efficiently and understand which get triggered most.

The method of inserting and validating data into the filter uses hash functions as indices. The number of hash functions determines the efficiency of the filter. The more hash functions a bloom filter users, the faster a filter gets filled. However, when a bloom filter does not use enough hash functions, the false positive rate increases. Equation (3.22) calculates the optimal amount of hash functions k for a bloom filter, where m and n are the sizes of the bit array and the number of elements inserted into the array, respectively.

$$k = (m/n)\ln(2) \tag{3.22}$$

These hash functions use the data as input. In the case of inserting data, the indices point to the bits that will turn to '1', shown in Figure 3.11. When validating data, the indices select the bits inside the filter and validates if they are both '1', it indicates a match Figure 3.12. Otherwise, the data is not in the filter Figure 3.13. A false positive occurs when data matches inside the filter without it ever being inserted. Figure 3.14 shows how a false positive can occur, which is why it is called a probabilistic data structure.

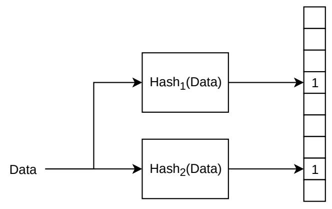


Figure 3.11: Inserting into a bloom filter.

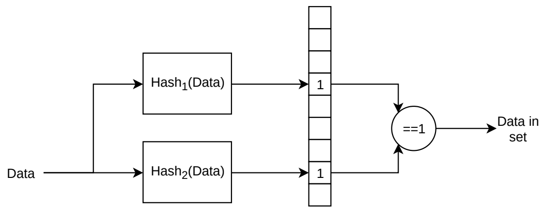


Figure 3.12: Data in the bloom filter.

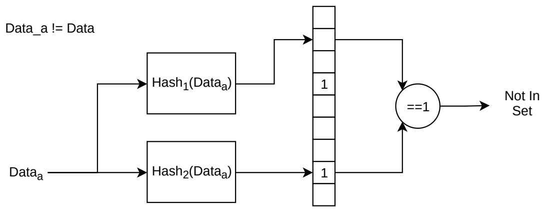


Figure 3.13: Data not in the bloom filter.

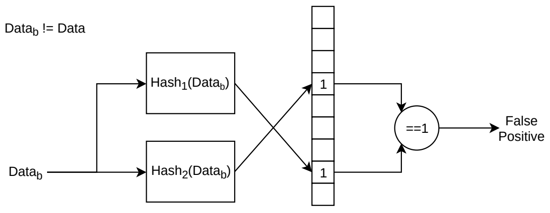


Figure 3.14: A false positive.

3.3.5.3 Counting bloom filter

The counting bloom filter uses more bits for each entry, which is called a bucket. Instead of inserting a '1' into the buckets, the value inside increments. We are limited to the data size for the QDR memory, which is 288 bits in total. With a multiple of two, we can divide the total bit length into evenly sized buckets. We chose that each entry is 4-bits, which allows for a maximum of 15 values for each bucket. This amount of bits creates a total of $288/4 = 72$ buckets with a maximum value of 15 for a bucket.

A problem arises when an overflow occurs in one bucket. In that case, we chose to send that filter to the central processing unit (CPU) over the PCIe bus. In the worst-case scenario, all the buckets overflow after each other. After that, the FPGA sends those filters to the CPU. Now those buckets are empty, but after an additional 16 packets arrive, another overflow occurs, meaning an overflow occurs every $12 \times 16 = 192$ cycles. Equation (3.23) shows this bandwidth calculation using these cycles, which fits the PCIe bus.

$$\begin{aligned}
\text{Bandwidth} &= \text{Bits} \times \text{Entries} + \frac{\text{Bits} \times (\text{Frequency} - \text{Entries})}{\text{Max Value} \times \text{Cycles for overflow}} \\
&= 33^4 \times 288 + \frac{288 \times (156.25 \cdot 10^6 - 33^4)}{192} \\
&= 574,141,366.5 \\
&\approx 574 \text{ MBit/s}
\end{aligned} \tag{3.23}$$

4 Setup configuration

In the thesis we used multiple tools that are used in tendem. First, we describe the build system that contain all the tools used to build and synthesise designs. The description includes how to connect to the server and how it conntects to the other systems. Secondly, we describe the switch itself which includes how to configure it in the future with pointers to tutorials to more documentation. Finally, we describe how to run the design on the switch's VM, including how to send data to the design and information about a few maxeler utilities.

4.1 Build server configuration.

The build server contains the Altera and maxeler tools. It uses the Altera tools to synthesize the DFE designs made with the maxeler tools and create a compiled program. The host itself runs CentOS6.10. The reasoning behind this is that the Juniper server, discussed in Section 4.2, accepts GLIBC 2.12, while programs compiled with a higher version do not execute.

4.1.1 Connect to server

To connect to the server, we use SSH. Currently, the server uses the hostname 'callisto' on the 'mars' routing instance used by the TU Delft threat intelligence team. The callisto host listens to IP address 10.0.0.23 on the 'mars' instance.

The server has two users to connect with, which are 'root' and 'user', where 'user' has sudo rights. Listing 4.1 shows an example ssh configuration to connect to the build server for both Linux and Windows. There is a 'Keepass2' encrypted password database that holds the passwords for these systems. The TU Delft's threat intelligence team holds that database.

4.1.2 Interface Configuration

The build system has a specific network interface configuration. Table 4.1 shows the default network card configuration. The configuration files for these interfaces are in the '/etc/sysconfig/network-scripts' directory with an 'ifcfg-' prefix.

The em1 interface connects to the mars host using a network cable. The configuration of the 'DNS' and 'GATEWAY' enables a network connection from the build server. It uses this connection to validate the Altera license as an example.

The p55p1 and p55p2 interface belong to a dual-port QSFP+ adapter on the build system. The adapter uses Mellanox Technologies MT27520 Family technologies. P55p1 connects to the VM on the Juniper switch through the switch's management ports.

Listing 4.1: Connect to the build system

(a) Using linux

```
Host mars
    HostName 131.180.119.22
    User <mars user>

Host callisto
    HostName 10.0.0.23
    User user
    ProxyJump mars
```

(b) Using windows

```
Host mars
    HostName 131.180.119.22
    User <mars user>
    MACs hmac−sha2−512

Host callisto
    HostName 10.0.0.23
    User user
    ProxyCommand ssh.exe −q −W %h
        :%p mars
    MACs hmac−sha2−512
```

Interface	**Configuration**

em1	IPADDR=10.0.0.23
	PREFIX=24
	DNS1=8.8.8.8
	DNS2=10.0.0.1
	GATEWAY=10.0.0.1

p55p1	IPADDR=10.0.3.42
	PREFIX=24
	MACADDR=14:DA:E9:FA:02:93

p55p2	

Table 4.1: Network card configuration

Meanwhile, p55p2 connects to Port 23 of the QFX5100 switch without any additional configuration. With this network interface, we send packets to the DFE system.

4.1.3 Recovering the server

If the server stops functioning due to a hardware failure, a fully configured backup image is available. The image itself contains the Altera and maxeler tools. Additionally, all the network interfaces mentioned above are pre-configured. We used Clonezilla to create the image, which can restore the image to a disk. During the restoration process, Clonezilla fails to install the grub bootloader, which is a false positive. It fails because Clonezilla is based on the Debian distribution and expects a separate bootloader partition. CentOS6.10 installs the bootloader differently, so after the error, it still functions as intended.

4.1.4 Altera license

The Altera tools, primus, in this case, use the LM_LICENSE_FILE variable. Listing 4.2 shows the current definition of this variable. to point to the TU Delft's FlexLM license server.

Listing 4.2: Altera license environment variable

```
LM_LICENSE_FILE=27013@flexserv4.tudelft.nl
```

In this configuration, the license file variable contains 27013@flexserv4.tudelft.nl. This address points to one of the TU Delfts' FlexLM license servers. If the Altera tools cannot find the license server, the variable needs to change.

The website https://flexlm-info.tudelft.nl shows all the other license servers of the TU Delft. Look there for the license server for a Linux version of the Altera Primus software. Afterwards, change the variable in '/etc/profile.d/quartus.sh' to make it persistent.

4.2 Juniper QFX5100

The Juniper switch contains the QFX-PFA-4Q module, which holds the DFE and other off-chip resources. The switch uses a packet forwarding engine (PFE) to send data from the external server ports towards the FPGA ports. Currently, the external ports 10 and 23 of the switch are in use. Port 10 contains a loopback device, while the build server is connected to port 23 using a QSFP+ cable.

4.2.1 Current Configuration

The Juniper QFX5100 switch has some specific configuration to send data from any of its ports to the QFX-PFA-4Q module. Figure 4.1 shows a representation of the switch configuration.

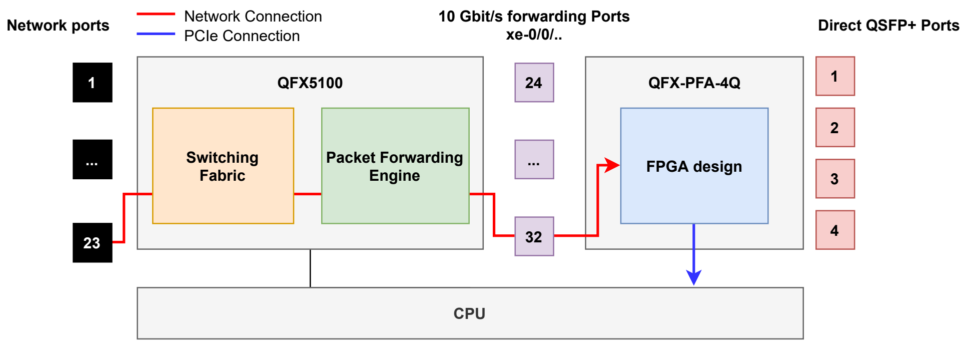


Figure 4.1: Physical frame received by the physical layer

Port 23 connects to the packet switching fabric, where it routes data from one interface to another. The same port connects to Virtual LAN (VLAN) 'v0_0' on the switch for a logical connection between network interfaces. Additionally, we connect the same VLAN to a network port understood by the QFX-PFA-4Q module.

Physical maxeler ports	**Ports represented in the Maxeler Tools**	**PFE interface**
QSFP port #0 10G sub-channel 0	JDFE_QSFP0_10G_PORT0 JDFE_XE32_10G	Xe-0/0/32
QSFP port #0 10G sub-channel 1	JDFE_QSFP0_10G_PORT1 JDFE_XE33_10G	Xe-0/0/33
QSFP port #0 10G sub-channel 2	JDFE_QSFP0_10G_PORT2 JDFE_XE34_10G	Xe-0/0/34
QSFP port #0 10G sub-channel 3	JDFE_QSFP0_10G_PORT3 JDFE_XE35_10G	Xe-0/0/35
QSFP port #1 10G sub-channel 0	JDFE_QSFP1_10G_PORT0 JDFE_XE24_10G	Xe-0/0/24
QSFP port #1 10G sub-channel 1	JDFE_QSFP1_10G_PORT1 JDFE_XE25_10G	Xe-0/0/25
QSFP port #1 10G sub-channel 2	JDFE_QSFP1_10G_PORT2 JDFE_XE26_10G	Xe-0/0/26
QSFP port #1 10G sub-channel 3	JDFE_QSFP1_10G_PORT3 JDFE_XE27_10G	Xe-0/0/27
QSFP port #2 10G sub-channel 0	JDFE_QSFP2_10G_PORT0 JDFE_XE28_10G	Xe-0/0/28
QSFP port #2 10G sub-channel 1	JDFE_QSFP2_10G_PORT1 JDFE_XE29_10G	Xe-0/0/29
QSFP port #2 10G sub-channel 2	JDFE_QSFP2_10G_PORT2 JDFE_XE30_10G	Xe-0/0/30
QSFP port #2 10G sub-channel 3	JDFE_QSFP2_10G_PORT3 JDFE_XE31_10G	Xe-0/0/31
QSFP port #3 10G sub channel 0	JDFE_QSFP3_10G_PORT0 JDFE_XE36_10G	Xe-0/0/36
QSFP port #3 10G sub channel 1	JDFE_QSFP3_10G_PORT1 JDFE_XE37_10G	Xe-0/0/37
QSFP port #3 10G sub channel 2	JDFE_QSFP3_10G_PORT2 JDFE_XE38_10G	Xe-0/0/38
QSFP port #3 10G sub channel 3	JDFE_QSFP3_10G_PORT3 JDFE_XE39_10G	Xe-0/0/39

Table 4.2: Interlinked ports on Juniper Switch [8]

Table 4.2 we show how the interfaces of the QFX-PFA-4Q module and Juniper switch interconnect. The Juniper switch represents the physical QSFP+ ports on the QFX-PFA-4Q. Additionally, the switch can configure these to use the PFE. So, in the case of Figure 4.1, we connect port 23 to Xe-0/0/32 using a VLAN. Any data send to port 23 arrives at JDFE_QSFP0_10G_PORT0, which is JDFE_XE32_10G, that the DFE design uses. Finally, the design will output data to the CPU using the PCIe connection.

4.2.2 Connect to the switch

There are two methods to make a connection to the switch for configuration. Both these operations require a connection from the build system.

The first method is a serial connection using a cable directly connected to the switch. Listing 4.3 shows how to connect with this connection. The serial connection is a failsafe if other connections do not function as intended.

Listing 4.3: Juniper switch serial connection

```
screen /dev/ttyS0 9600
```

The second method is an SSH connection from the VM running on the Juniper Switch. Listing 4.4 shows the steps to connect to the build system.

Listing 4.4: Juniper switch ssh connection

```
# Connect to the Juniper VM
ssh maxeler@10.0.3.158
# Connect to the Switch management panel
ssh root@192.178.1.2
```

4.2.3 How to change the switch configuration

Once connected to the switchs' configuration console, we can reconfigure it. Listing 4.5 gives an example of how to configure an external switch port to the FPGA. In this example, we configure the first 10Gbit channel of port 21 to one of the FPGA interfaces. The specific interface links port 21 to the FPGA's JDFE_QSFP0_10G_PORT1. Table 4.2 shows which interface port on the switch represents a specific port on the FPGA.

4.3 Maxeler Tools

The Maxeler tools that are available are maxide, which is the environment to write code. The maxelersim environment is a virtual DFE environment where the design can verify whether it functions according to the specifications. Additionally, we also received the Juniper QFX5100 Switch with the QFX-PFA-4Q module from Maxeler. For more information about the environment and how to code the DFE, look at the tutorials. These are available in the '/opt/maxeler/maxcompiler/docs' directory on the build server.

Listing 4.5: QFX5100 configuration example

```
# enter the command line interface for the network switch
    configuration
> cli
# Start configuring, which starts edit mode
> configure
# Configure port 21 for 4x10 gbit
> set chassis fpc 0 pic 0 port 21 channel-speed 10g #
    Configures port 21 on 10Gbit/s
# Connect 10Gbit interface of xe-0/0/0/21:0 to vlan v0_1
> set interfaces xe-0/0/21:0 unit 0 family ethernet-switching
    vlan members v0_1
# Connect FPGA interface to vlan v0_1
# Now those ports are connected
> set interfaces xe-0/0/33 unit 0 family ethernet-switching
    vlan members v0_1
# Give vlan v0_1 a specific vlan-id
> set vlans v0_1 vlan-id 11
# Commit changes to the switch
> commit
# exit edit mode
> exit
# exit cli mode
> exit
```

4.3.1 The DFE design bitstream

The DFE bitstream uses the 'maxcompiler' utility to transform Java code to lower level VHDL code. This section describes how to create and build with these tools.

4.3.1.1 Create a DFE design

The Maxeler IDE (Interactive Development Environment) is available on the server on the '/opt/maxeler/maxide' folder. MaxIDE is an eclipse based IDE and can be accessed using SSH with x-forwarding enabled. Once booted, it is possible to write java code to create a DFE design and the CPU implementation. We recommended creating the DFE design and CPU implementation projects with MaxIDE, as it puts most if not all of the variables in place. However, it does not automatically use the Network libraries. Adding those to the project requires adding '/opt/maxeler/maxcompilernet/libs/MaxCompiler-Net.jar' to the DFE projects' classpath.

4.3.1.2 Build the DFE design

Building the design itself can be done using the MaxIDE or the Maxeler tools with the command line.

Building the DFE using MaxIDE is the easiest method to build a design and link it to a CPU implementation. However, there are some things to keep in mind while building it. The build function in the MaxIDE uses '.dfeprojectproperties', which holds properties for both the simulation and synthesise builds. Listing 4.1 shows an example of the properties found in this directory.

Listing 4.1: DFE makefile template

```
manager=<manager to build>
jvmargs=-Xmx2G
# Set maxcompiler specifc variables.
parameters=DFEModel=JDFE target=DFE maxFileName=<name of the design>
```

Building the DFE using the command-line gives some additional problems. Namely, the Altera tools require a display server, also called X-server, to function. The X-server does not function using the command-line to build a design, for example, during an SSH connection. To solve that issue, we use a program on the server called Xcrv that creates a virtual display server. The commands in Listing 4.2 show creating and setting the display server on the server, which solves the issue.

Listing 4.2: DFE makefile template

```
# Create a virtual x server accessible on port :1
> Xvnc :1&
# Set DISPLAY to port :1
> export DISPLAY=localhost:1.0
```

Building a design uses a 'build.sh' script and a Makefile, where 'build.sh' is a script located in the build server's '/opt/maxeler/scripts' directory. The Makefile creates a recipe that uses the 'build.sh' script to build the design. Listing 4.3 is an example Makefile, which shows how to create a custom build entry for a DFE design. Afterwards, the command 'make <NAME BUILD>' will automatically build the design and create the '.max' and '.h' files. The CPU implementation uses those files to communicate with the design. The '.max' contains the bitstream, and the '.h' file describes its interface to the programmer.

Listing 4.3: DFE makefile template

```
BUILD_SCRIPT := ./build.sh
MANAGER_LOC := # Package path where it can find the managers used by
    the maxide tools

.PHONY: <NAME_DFE_BUILD>
<NAME_DFE_BUILD>:
        $(BUILD_SCRIPT) $@ $(MANAGER_LOC) <MANAGER_NAME> DFE
```

```
.PHONY: <NAME SIM BUILD>
<NAME SIM BUILD>:
        $(BUILD_SCRIPT) $@ $(MANAGER_LOC) <MANAGER_NAME> DFE_SIM
```

4.3.2 The CPU implementation

The CPU implementation, just as the DFE design, can be created in MAXIDE. There are multiple methods to create a CPU implementation, such as using C++ or python. In the thesis, we chose C++ for more control.

By creating the CPU implementation with maxide, most variables for the project will be pre-configured. Additionally, the IDE creates a Makefile that the project uses. However, it does not enable the network libraries by default. As such, the '/opt/maxeler/maxcompilernet/include/slicnet' needs to be added to the includepath. Either use the IDE to change the include path or add it to the '.cproperties' file in the CPU implementation. The adjustments shown in Listing 4.4 need to be made to the Makefile to simulate a network design.

Listing 4.4: DFE makefile template

```
# Set the card to JDFE, which is the QFX-PFA-4Q platform
CARD ?= JDFE

# Set the port that the FPGA will listen to in the simulator.
# These ports start counting from '1', meaning PORT0 in the FPGA
    design is PORT1 in the simulator
PORT := QSFP0_10G_PORT1
# Set the IP address and netmask of the simulators TAP device.
IP_SW := 172.17.0.1
NETMASK := 255.255.255.0
# Assign the tap device
SIM_TAP := $(PORT):$(IP_SW):$(NETMASK)

# Tell maxcompilersim to use the networking functionalities on our
    TAP device.
start_sim: ; @'$(MAXCOMPILERDIR)/bin/maxcompilersim' -n $(SIMNAME) -c
     $(CARD) -e $(SIM_TAP) restart
```

The CPU build requires the generated .max file to reside in the correct directory. The rule 'MAXFILES := $(wildcard max/*.max)' must be added to the Makefile when using the command line. This rule finds all the .max files in the 'max/' directory. Otherwise, it looks for the .max files in a specific build directory of the DFE design, which the command line build does not use.

4.3.2.1 Running the CPU implementation

Once we have both the DFE design and the CPU implementation, the DFE system can run the design. We send the '.max' and CPU binary to the Juniper switch VM to run the maxeler toolset. There are some caveats with running the CPU implementation.

For example, using the ‘strings’ program on the CPU implementation will reveal all the strings inside the program. One of these strings points to the expected location of the ‘.max’ file. The tools will not load the design when the ‘.max’ file is not in that directory.

4.3.2.2 Sending Network traffic to the DFE

As mentioned in Section 4.2, the build server connects to the juniper switch on port 23, which uses a VLAN to connect to port 32. The DFE design accepts any traffic that goes through the VLAN. However, some network traffic utilities need some specific preparations. We used two utilities, where both require root permissions on the build system to send network traffic.

Definition 4.1
tcpreplay a utility that replays network traffic over a certain interface.
scapy a python library that allows creating network packets and sending them over the internet.

Scapy requires a constructed network packet from the Ethernet to a TCP layer. Otherwise, it does not function as intended and gets ignored by the DFE interface. For safety, we used the MAC address of port 32. These addresses can be configured using the Juniper switch or viewed using the ‘maxnet’ utility. Listing 4.5 shows an example output of the utility for a single interface. The utility shows the MAC addresses and the current state of the utility. However, it does not show anything when there is a DFE design loaded on the chip.

Listing 4.5: DFE makefile template

```
# Show the link of a single interface. Specifying no interface shows
    all interfaces.
> maxnet link show JDFE_XE32_10G

JDFE_XE32_10G:
  Module Present: false
         Link Up: true
     MAC address: 88:a2:5e:55:88:a0
      RX Enabled: true
       RX Frames: 0 ok
                  0 error
                  0 CRC error
                  0 invalid/errored
                  0 total
      TX Enabled: true
       TX Frames: 0 ok
                  0 error
                  0 CRC error
                  0 invalid/errored
                  0 total
```

4.3.3 Juniper VM

The Juniper VM runs on the switch and interfaces with the DFE platform that runs the designs and programs. However, some caveats need to be known before a program can run.

After synthesising, we send the compiled design and max file to the Juniper VM, where the appropriate tools are running. Section 4.2.2 explains how to connect to the VM.

4.3.3.1 Maxtop

This program shows the current state of the DFE, including the currently loaded design. In Listing 4.6 you can see an example of the output of this tool.

Listing 4.6: Useage maxtop

```
> maxtop -v -d 0

MaxTop Tool 2018.3
Found 1 card(s) running MaxelerOS 2018.3
Card 0: QFX-PFA-4Q (P/N: 241124) S/N: 000000123 Mem: 24GB PCIe: 00:0b
    .0 Net: 2

Load average: 0.00, 0.21, 0.21

DFE %BUSY  TEMP  MAXFILE        PID  USER  TIME COMMAND
 0  100.0% -     fc7b8661e2...  -    -     -    -

...
Bitstream: app_id: -1 app_rev:-1 checksum:
    fc7b8661e2a4b51140994748dcec01962c6db5d66fc8bfdd687cfb614f09e69d
...
```

A good thing to note is that it shows the current loaded design as a hash value at the ‘checksum’. The max file contains the same checksum as the one noted at the end of the file. If the checksum inside the .max file matches the one of maxtop, it loaded the correct design.

4.3.4 Maxeler Licenses

The maxeler tools, which are the build tools and the QFX-PFA-4Q module inside the Juniper switch, use licenses to function. The location of the systems’ license files is in the ‘opt/maxeler/maxcompiler/licenses/’ directory. In Table 4.3 we see the current status of these license files. Once the license expires, we can request a new one with the maxeler university program. To generate the licenses, send an email to info@maxeler.com to ask for the features shown in Table 4.3.

	Build Server	**QFX-PFA-4Q**
Expiration date	2021-08-07	2021-08-07
Features	MaxCompiler_core MaxCompiler_core_sim MaxJDFE_Runtime	MaxJDFE_Runtime
Special note	License uses MAC address 14:DA:E9:FA:02:93	

Table 4.3: Current state of the maxeler licenses.

Implementation 5

This chapter explains the design in four distinct parts. First, we give a global overview of the design that tells the placement of the computation kernels. The second part focusses on the functionality of the network parser. Afterwards, we talk about fingerprint detection. Finally, it concludes memory communication, which shows how to store the fingerprints and send them to the CPU.

5.1 Overview

The design itself consists of four different parts, as shown in Figure 5.1. Every stage of the diagram does something different; these differences are:

The Traffic parser parses network traffic from the interface and sends the fields we need to the next stage.

The Pattern-finding stage finds the fingerprints in the collected network traffic.

The Memory communication stage communicates with the memory according to the fingerprint found at the pattern-finding stage.

The QDR cleaner is a different design that clears the QDR memory at startup to remove random data.

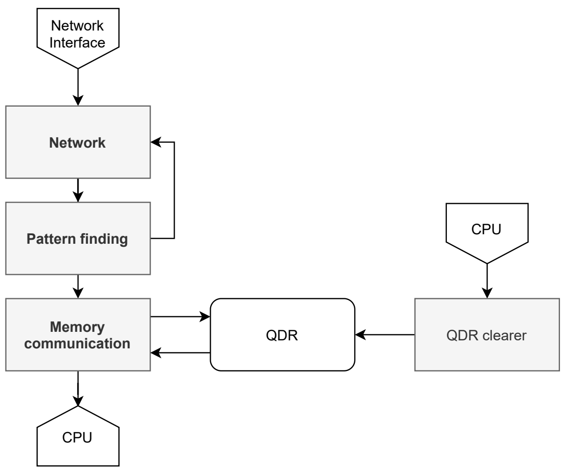


Figure 5.1: Design overview.

5.2 Traffic Parser

The kernels discussed in this section are responsible for parsing network traffic from the network interface and direct it to the pattern-finding algorithm.

5.2.1 Network & Ethernet

This section briefly explains how the design parses network traffic from the network interface. Maxeler Technologies made the code that splits the headers. The ethernet kernel itself waits for our network fields and sends the output to the remaining kernels.

5.2.1.1 Input & output

The input of the kernel is a 64-bit data stream that connects to a network interface. It starts receiving data when the network interface detects the start-of-frame, discussed in Section 2.1.2. Every cycle Afterwards, there are new bits of information.

The kernel outputs multiple streams that contain the header data of the different protocols. The ethernet kernel grabs the fields from these headers and sends them to the field cache kernel.

5.2.1.2 Kernel Function

The kernel parses the network datagram from the input. Each splitter in the kernel waits for a specific point to start parsing. These splitters communicate with each other to determine that point. For example, the IP parser only starts after it finishes parsing the Ethernet header. It separates the input data into multiple streams that wait for the header and payload for a specific protocol. This method keeps the variable data inside the headers into account. When a specific header is larger than the default, it waits some additional cycles before sending it to the next one.

Each parser uses a shift register that is large enough to hold the maximum header length for the header so the register can hold it. Afterwards, every splitter waits for cycles until the shift register holds all the required header data and parses the header. In some cases, headers contain optional fields. In those cases, the kernel puts itself on hold until those optional fields have passed.

5.2.2 Field Cache

This section discusses the field cache implementation, which sends data from the network toward the XOR algorithm kernel.

5.2.2.1 Input & output

The kernel has two inputs and one output. The inputs hold the network fields shown in tab Table 5.1. Network fields receive the parsed data from the network. Meanwhile, the cache fields receive fields that should be cached. Caching occurs when the pattern-finding algorithms detect multiple patterns in one packet. The output values, shown in

Table 5.2, is the same as the input values, but adds a one-bit flag. This flag tells the pattern-finding algorithm that an incoming packet caused a cache hit in the memory.

Field	**Bits**
Sequence Number	32
Source Address	32
Source Port	16
Destination Address	32
Destination Port	16

Table 5.1: Input values.

Field	**Bits**
Sequence Number	32
Source Address	32
Source Port	16
Destination Address	32
Destination Port	16
Cache hit	1

Table 5.2: Output values.

5.2.2.2 Kernel Function

This kernel caches network fields, where the pattern-finding algorithm detected multiple patterns. The reasoning behind this is to account for the possible *Key* value that the scanner may use, as discussed in Section 3.2.2. Figure 5.2 shows the construction of the kernel. The inputs stream through the field cache and the selector to the output.

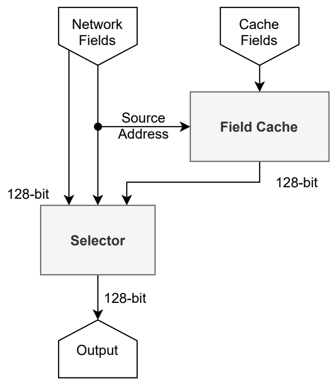


Figure 5.2: Kernel Design.

The field cache module, shown in Figure 5.3, caches data. It uses the source addresses of the inputs as the cache's memory addresses. Retrieving data from the cache uses the source address from the network fields. Storing data inside the cache uses the source address inside the cache field input. It reduces these addresses from a 32-bit number to a 10-bit number, so it fits inside the cache's 1,024 entries. In the end, the fields read from the cache are passed towards the next module.

The selection module, shown in Figure 5.4, the output of the kernel. There are two options: the input network fields; or an XORed version of the network fields. The latter performs the XOR operation on the data from the cache and the input fields. This operation aims to remove the possible *Key* value and reduce the number of operations.

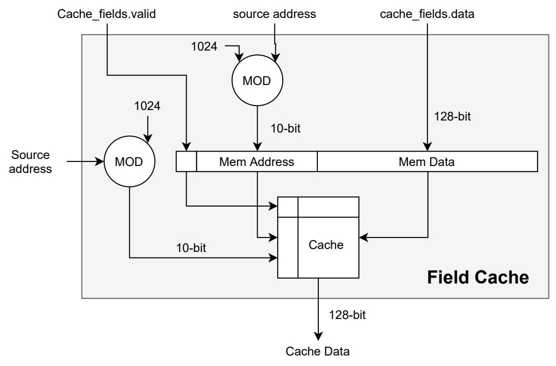


Figure 5.3: Field cache module.

Finally, it determines if the incoming packet's source address is equal to the one retrieved from the cache. In the case they are, it selects the XORed fields. Otherwise, it selects the network fields.

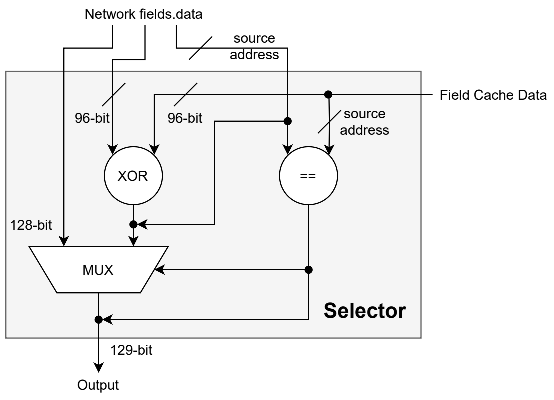


Figure 5.4: Selection module.

The XORed result excludes the source addresses, as the bloom filter kernel uses the value as its input. However, the problem is that the output source address should be zero when a cache hit occurs, as the source addresses are the same during a cache hit. The solution is setting the cache-hit flag to 1, which informs the algorithm that it should interpret the source address as 0.

5.2.2.3 Resource Utilization

Table 5.3 shows the synthesized resource utilization for this kernel.

Kernel	**ALMs**	**BLOCK RAM**
Field Cache	2,879	12

Table 5.3: Resource Utilization of the field cache.

5.3 Pattern finding

The kernels discussed in this section are responsible for finding fingerprints inside the network packets. First, we discuss the XOR pattern-finding algorithm. Afterwards, we end with the fingerprint kernel, which completes the fingerprint.

5.3.1 XOR Function

This section discusses the implementation of the XOR fingerprint finding algorithm discussed in Section 3.3.3.3. This kernel receives the fields from the field cache (Section 5.2.2), finds the XOR pattern and sends it to the fingerprint kernel (Section 5.3.2).

5.3.1.1 Input & output

The kernel input is the network fields and the cache-hit flag from the field cache kernel, shown in Table 5.4. The output, shown in Table 5.5, contains the same network fields. These are two of the fingerprints' shift values, a selector and a 363-bit large data vector that holds the fingerprint's remaining part.

Field	**Bits**
Sequence Number	32
Source Address	32
Source Port	16
Destination Address	32
Destination Port	16
Cache hit	1

Table 5.4: Input of the XOR function.

Name	**Bits**
Source Address Shift	6
Destination Address Shift	6
Selector	3
Network Fields (Table 5.4)	129
Data vector	363

Table 5.5: The output of the XOR function.

5.3.1.2 Kernel Function

The kernel orchestrates multiple small XOR modules to perform as many operations per cycle as possible on independent network fields. Every XOR module, shown in Figure 5.6, performs $1,089$ comparisons for $1,089$ cycles, meaning one module performs all the required $1,185,921$ operations. The important part is to orchestrate these modules to allow them to function independently. Figure 5.5 shows the construction of the kernel. A queue holds the input, which waits until one of the modules is available for input.

The selection process uses the busy selector shown in Figure 5.7. When an XOR module starts to process input, its state becomes busy. The selector logic uses these busy bits and uses leading leading-one detection to find the first non-busy module, Equation (5.1) shows the process.

First, the busy bits are inverted. This process puts a '1' at the positions that have a free module. Afterwards, it uses leading-one detection to find the first free module.

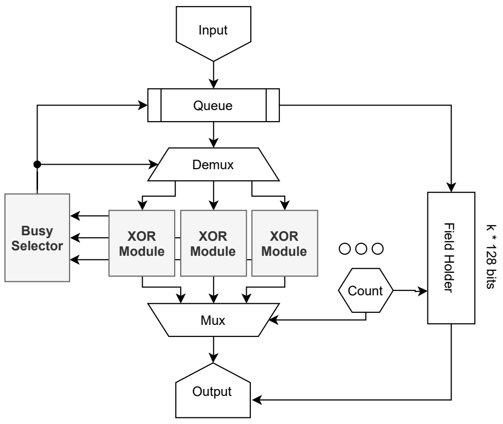


Figure 5.5: The XOR kernel.

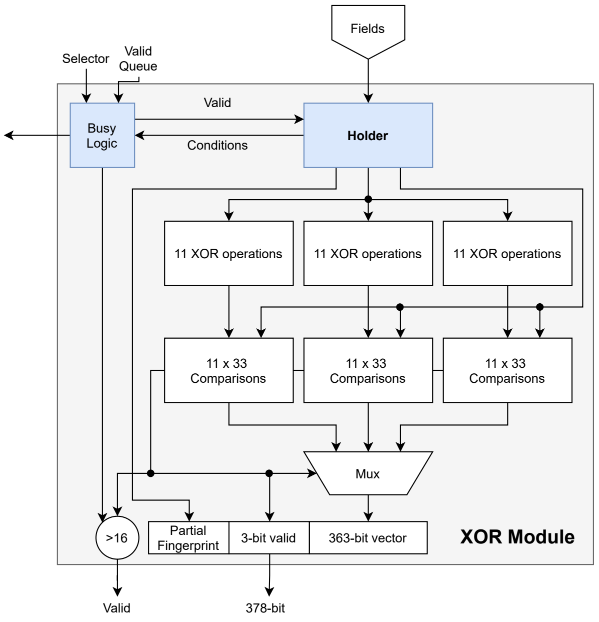


Figure 5.6: A XOR module.

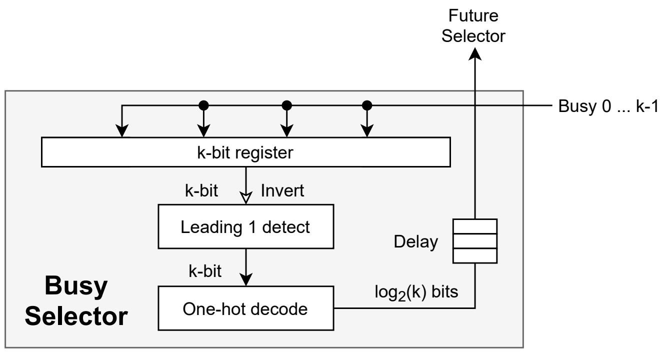


Figure 5.7: Selector Logic.

$$\begin{aligned}\text{Busy}_{\text{input}} &= 0b1100\\ \text{Busy}_{\text{inverted}} &= 0b0011\\ \text{Busy}_{\text{leading-one}} &= 0b0010\\ \text{Future Selector} &= 2\end{aligned} \tag{5.1}$$

Finally, it performs one-hot decoding on that value, which converts it to the kernel's positional value.

When a module is available, the queue sends the following fields to that module. At that time, the selector points to the module until it changes to its busy state. Additionally, it uses the same selector to store the fields from the queue into a temporary cache. That cache holds the fields that a module is currently using for computation. At the end of the computation, it retrieves these fields from the cache. Otherwise, each module had to send its input along with the output. So, we chose this method to reduce the modules' size, and now it is a simple case of retrieving the fields from the cache.

The holder function, shown in Figure 5.8, holds the input fields temporarily until new input arrives. Once the holder receives the input, the counters start counting to either

1, 089 or 33, depending on the cache-hit flag.

Every counter has a limit of 33 and is chained together. The counters continuously keep counting until count-2 reaches its max value. We use these counters to indicate that the module is busy. The module stops being busy once the counters are at their max value. Additionally, these counters shift the source address and destination port so that every computation set is unique. The values of these counters construct the partial fingerprint of that computation.

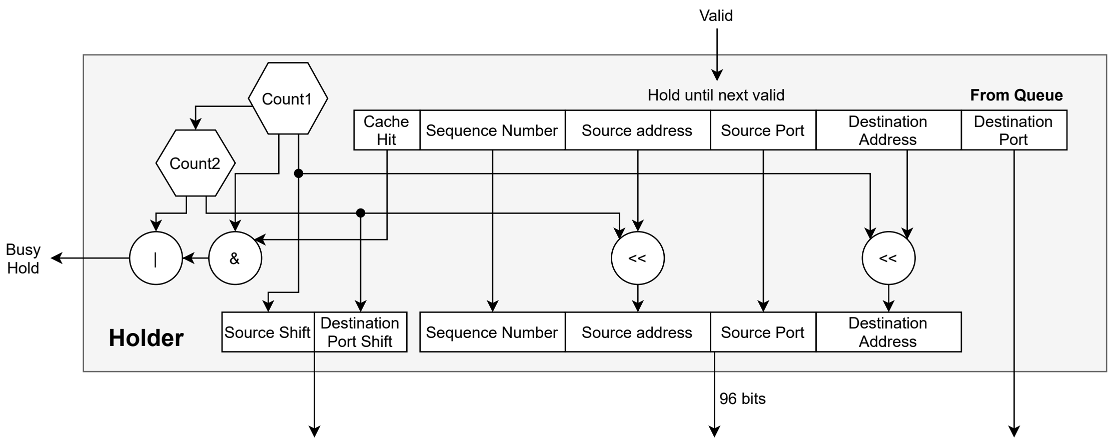


Figure 5.8: Input Holder.

The busy logic keeps the module in a busy state until it performed all computations. Figure 5.9, shows the model of the logic function. Combining the selector and the valid queue flag determines if the input is for a specific module. Equation (5.2) shows the validation process, where i is the XOR module number. It compares$i$ with the selector and queue flag combination. When it matches, the module accepts that data and puts the module in a busy state. Additionally, this is the valid flag for the holder function. Once the XOR module performed its last operation, the busy flag is set back to its original state.

The comparison phase uses the comparison functions discussed in Section 3.3.3.3. As mentioned before, an XOR module performs 1, 089 computations every cycle. To avoid the resource peak shown in Section 3.3.3.3, we choose to perform 1, 089 operations in three sets. There are three phases to the calculation process.

- The XOR stage performs 3×11 XOR operations on the input fields.
- The comparison stage performs 33 comparisons for each of the 11 results from the XOR stage. With all modules, it results in 3×363-bit vectors. Furthermore, it sets an additional flag if it found a match.
- In the selection stage, where it chooses the output of one of the three sets.

There are more intricate controls at the selection stage. Each comparison stage contains a flag that indicates it finds a match. The module combines these flags into a

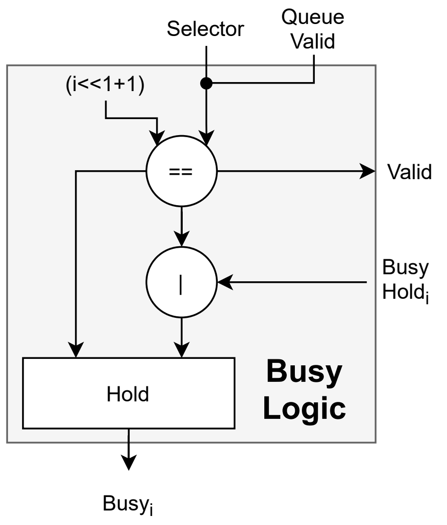


Figure 5.9: Busy Logic.

$$\begin{aligned} i &= 2 \\ &= 0b10 \\ (i \ll 1) + 1 &= 0b101 \\ \text{Selector} &= 0b10 \\ \text{Queue valid} &= 0b1 \\ \{\text{Selector}, \text{Queue valid}\} &= 0b101 \end{aligned} \tag{5.2}$$

3-bit selector, which the multiplexer uses to choose the 363-bit output. However, each set may contain a match, which means multiple matches were detected. To not lose that information, the kernel sends the selector to the next kernel. Meanwhile, the counters from the hold function form the partial fingerprint.

5.3.1.3 Optimizations

There is a small functionality update to the kernel.

Adding scalars To reduce the number of computations, we chose to add scalars to the design. These scalars control the maximum value of the module counters. We place these scalars at the counters to reduce the maximum operations on the fly. The later shift values might continuously generate the same values.

5.3.1.4 Resource Utilization

Table 5.6 shows the synthesized resource utilization for this kernel.

Kernel	**ALMs**	**BLOCK MEM**
XOR Module	10,434	0
XOR Kernel	209	8

Table 5.6: Resource Utilization of the XOR function.

5.3.2 Fingerprint Kernel

This section describes the method to extract the remaining parts of the fingerprint from the output of the XOR pattern-finding function.

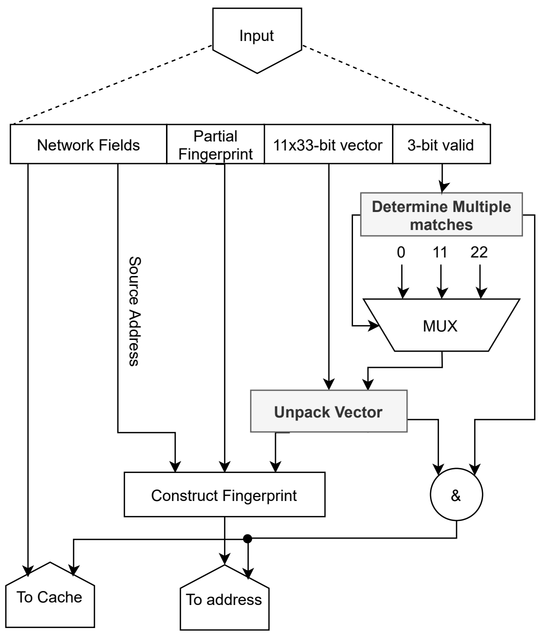


Figure 5.10: Fingerprint kernel logic.

5.3.2.1 Input & output

The input to this kernel is the data received from Section 5.3.1, shown in Table 5.7. The input contains a partial fingerprint together with a 363-bit vector that contains the remaining fingerprint data. There are two outputs of the kernel, shown in Table 5.8. The first one holds the fingerprint, while the other contains the network fields send back to the field cache.

Name	**Bits**
Source Address shift	6
Destination Address Shift	6
Selector	3
Network fields	129
Data vector	363

Table 5.7: The input of the function.

Name	**Bits**
Source Address shift	6
Destination Address Shift	6
Source Port shift	6
Destination Port shift	6
Source Address	32
Network Fields	129

Table 5.8: The output of the function.

5.3.2.2 Kernel Functionality

The kernel finds the remaining parts of the fingerprint inside the bit-vector. The function shown in Figure 5.10, finds matches by performing $\log_2$ operations on the vector. The 3-bit selector is used to see which set from the previous kernel was selected.

The find matches module, shown in Figure 5.12 uses both leading-1 and trailing-1 detection, and one-hot decoding. With leading-1 and trailing-1 detection, it checks the data bits from both sides. Meanwhile, one-hot decoding transforms the result of the leading-1 into a positional number. That number is the selector of a multiplexor (MUX), which chooses an offset for the data bits. Only when both the results of leading-1 and trailing-1 are the same, one match occurred. Equation (5.3) gives an example of how this functions with multiple 1's in its input.

$$\begin{aligned}
\text{Data} &= 1100.0000.0000.0000 \\
\\
\text{Trailing-1(Data)} &= 0100.0000.0000.0000 \\
\text{Leading-1(Data)} &= 1000.0000.0000.0000 \\
\\
\text{One-Hot Decode(Leading-1)} &= 15 \\
\text{Equal}\{\text{Leading-1,Trailing-1}\} &= 0
\end{aligned} \tag{5.3}$$

The module adds the offset from the MUX to the source port shift value. The kernel does not know the origin of the 11 × 33 bit-vectors. Thus, it adds the offset from the mux to normalize the values.

The unpacking vector module, shown in Figure 5.11 performs two $\log_2$ operations on the vector data. It interprets the 363-bit vector as a 11 × 33-bit vector or matrix. First, it searches for the column that contains the match with a logarithmic function. Afterwards, it uses another logarithmic function to find the correct row.

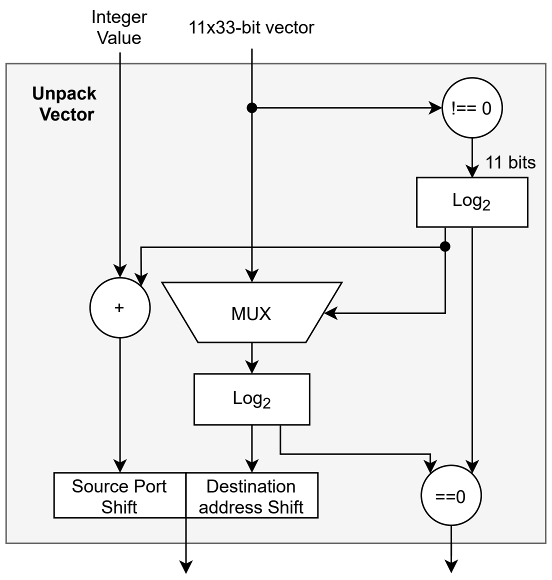


Figure 5.11: Unpacking the vector.

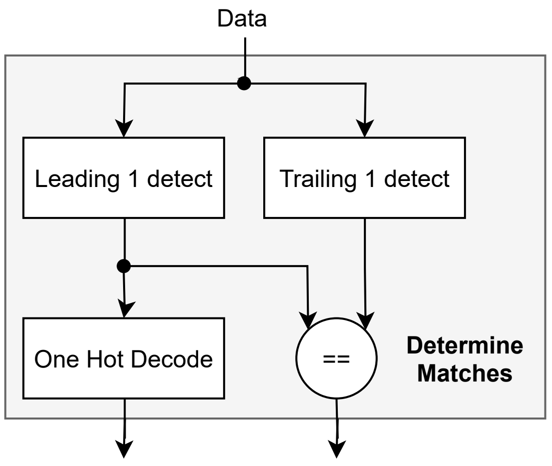


Figure 5.12: Finding multiple matches.

A $\log_2$ function creates a fixed-point result in the form of ##.##. The integer part of the result finds the column or row, while the fraction tells us whether there were other matches in the data. Meanwhile, the fractional part determines if there were other matches inside the data.

The module determines the column by looking at which of the 11 33-bit values is larger than 0. Table 5.9 shows an example of that functionality. The 11-bit result is the input to the first logarithmic function. The integer part of the output is source-port shift value. It uses that value to select the correct 33-bit row, which goes through another logarithmic function to find the destination port shift-value.

Data 4x5	0	0	0	0
	1	0	0	0
	0	0	0	0
	0	0	0	0
	1	0	0	0
>0	**1**	**0**	**0**	**0**

Table 5.9: Find data in a vector.

5.3.2.3 Optimizations

An optimization of the design, shown in Figure 5.13 , exchanges the logarithmic function with the function shown in Figure 5.12. As explained previously, when the two values approximated from both sides are equal. It found no other matches in that data. Equation (5.3) showed the case when there were multiple patterns found, and thus it sends the fields to the field cache kernel.

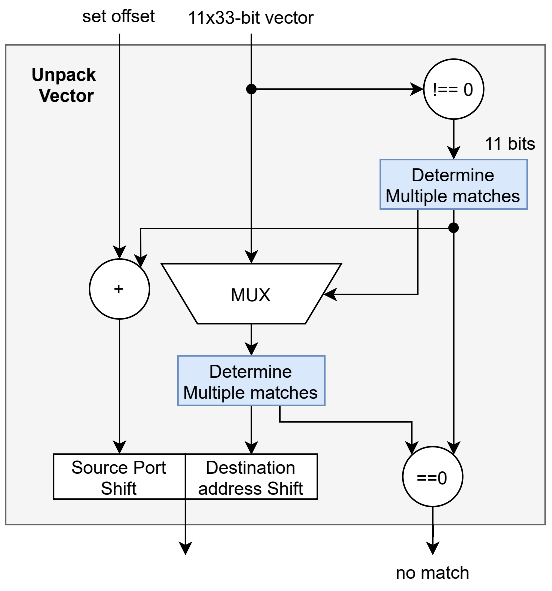


Figure 5.13: Unpacking the vector optimization.

5.3.2.4 Resource Utilization

Table 5.10 shows the synthesized resource utilization for this kernel.

Kernel	**ALMs**
Fingerprint	479

Table 5.10: Resource Utilization of the fingerprint kernel.

5.4 Memory Communication

The kernels in this section are responsible for communicating with the QDR memory The address generator creates a memory address from the fingerprint. The QDR communication kernel takes that address and requests the data located there from the QDR memory. Finally, the data reaches the bloom filter, where it gets processed and send back to memory.

5.4.1 Address Generation

In this section, we discuss the kernel that transforms a fingerprint into a QDR memory address.

5.4.1.1 Input & output

The input of this kernel, shown in Table 5.11, is the fingerprint discoverd by the pattern finding function function. The output of this kernel is the following QDR memory address to read. This is a 21-bit address field, together with a few flags for control shown in Table 5.12. The control flags inform the memory communication kernel of specific actions it may perform, such as which kernel should send the output.

Name	**Bits**
Source Address shift	6
Destination Address Shift	6
Source Port shift	6
Destination Port shift	6
Source Address	32

Table 5.11: Address Generation input.

Field	**Bits**
Wait for write	1
From Kernel	1
Source Address	32
Address	21

Table 5.12: Address Generation output.

5.4.1.2 Kernel Function

This kernel generates a unique memory read address. In the overview of the kernel, Figure 5.14, it performs two separate actions. Both of these actions result in read addresses. The Fingerprint generator generates an address from the fingerprint, while the counter generator uses a sequential counting mechanism.

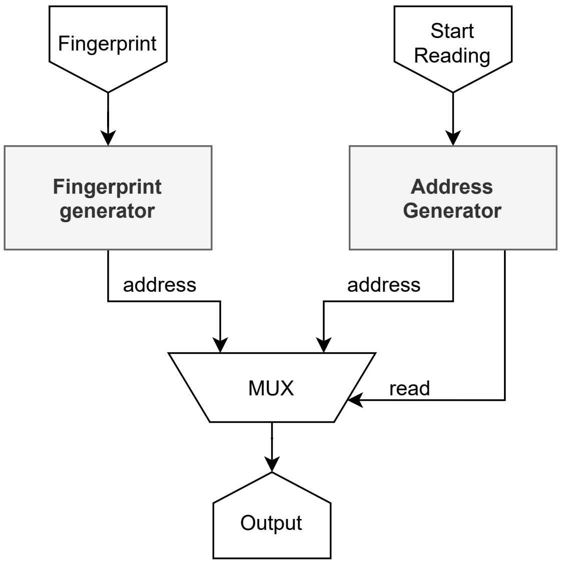


Figure 5.14: Address generator overview.

The fingerprint address module, shown in Figure 5.15, uses the shifts found in the input and combines them into one memory address. There are a total of $1,185,921$ different shift combinations fingerprints available. Equation (5.4) shows the calculation of the address from the shift values.

$$\text{Address} = Addr_s + Addr_d \times 33 + Port_d \times 1,089 + Port_s \times 35,937 \tag{5.4}$$

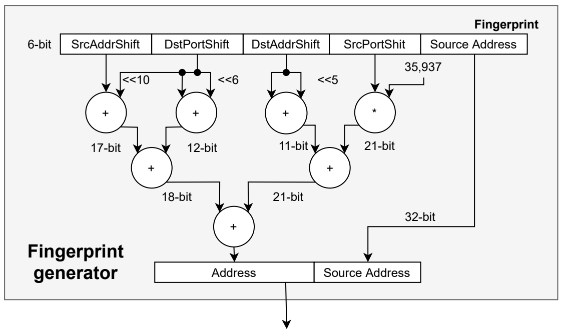


Figure 5.15: Fingerprint generator design.

The equation, while simple, is not implemented on hardware directly. The reason is that multiplication units are expensive, while other options are available. To lessen the cost, we rewrite multiplications as a set of shifts and additions. Equation (5.5) and Equation (5.6) rewrite the multiplications in small additions.

$$\begin{aligned} x \times 33 &= x \times (32 + 1) \\ &= x \times (2^5 + 1) \\ &= (x \ll 5) + x \end{aligned} \tag{5.5}$$

$$\begin{aligned} x \times 1,089 &= x \times (1,024 + 64 + 1) \\ &= x \times (2^{10} + 2^6 + 1) \\ &= (x \ll 10) + (x \ll 6) + x \end{aligned} \tag{5.6}$$

Equation (5.7) shows the result of rewriting the largest value in terms of shifts and additions. This function has too many additions, which would add more delay inside the design. For that reason, we chose to use a multiplier instead.

$$\begin{aligned} x \times 35,937 &= x \times (32,768 + 2,048 + 1,024 + 64 + 32 + 1) \\ &= x \times (2^{15} + 2^{11} + 2^{10} + 2^6 + 2^5 + 1) \\ &= (x \ll 15) + (x \ll 11) + (x \ll 10) + (x \ll 6) + (x \ll 5) + x \end{aligned} \tag{5.7}$$

The counter address module, shown in Figure 5.16, waits for a command to arrive. When it gets triggered, it starts to read all the memory addresses sequentially. This functionality is to dump the memory towards the CPU. So, in the end, there is no data loss. To keep the memory bandwidth low, one of the counters max value is reconfigurable. That counter puts a delay between read requests while still dumping all the memory.

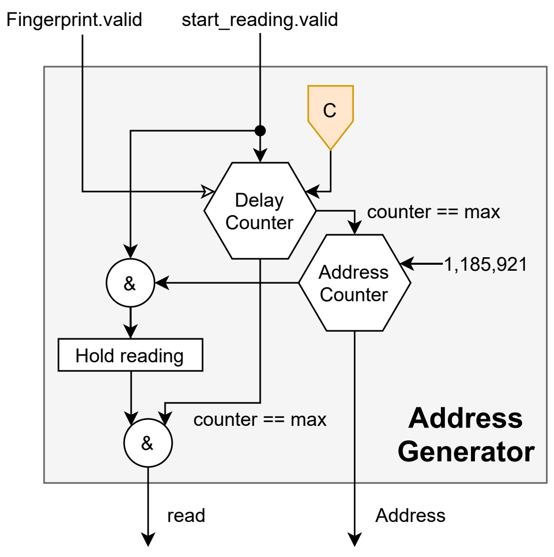


Figure 5.16: Counter address generator design.

5.4.1.3 Resource Utilization

Table 5.13 shows the cost of this kernel.

Kernel	ALMs	MULTs
Address Generator	241	1

Table 5.13: Resource Utilization of the Address generator.

5.4.2 Quad Data Rate Memory Communication

This section discusses the communication between QDR memory and FPGA, which focuses on memory consistency.

5.4.2.1 Input & output

There are three distinct inputs to the kernel, all shown in Table 5.14 A read command arrives when we want to read a specific address from memory and send it to a specific kernel. A write command arrives when there is an update to the data. The final input to the kernel receives the data directly from the QDR memory, which arrives after sending a read command.

Name	Bits
Read command	
Wait for write	1
From kernel	1
Source address	32
Address	21
Write command	
Address	21
Data	288
From QDR ×2	
Write address	21
Read address	21
Read enable	1
Write enable	1
Write byte enable	8
Write data	144

Table 5.14: QDR kernel inputs.

Name	Bits
To kernel ×2	
Address	21
Source address	32
Data	288
To QDR ×2	
Write address	21
Read address	21
Read enable	1
Write enable	1
Write byte enable	8
Write data	144

Table 5.15: QDR kernel ouputs.

There are two separate outputs of the kernel, shown in Table 5.15. The aptly named 'to kernel' sends the result from the QDR to its designated kernel. The final outputs send QDR commends to the memory.

5.4.2.2 Kernel Function

The kernel's goal is to handle all read and write requests to the memory and keep memory consistency. It achieves this by keeping track of previous read addresses. Figure 5.17 shows the overview of this kernels' functionality.

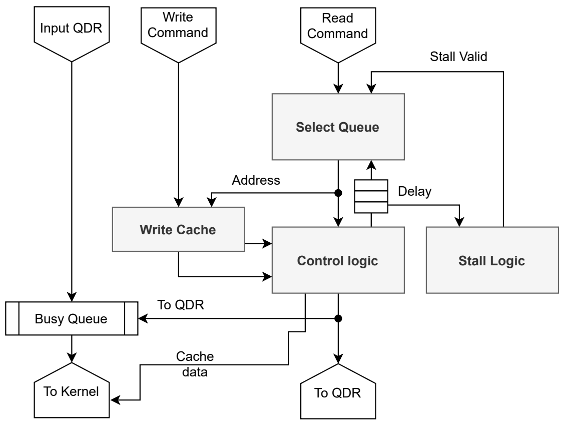


Figure 5.17: Structure of the Memory Communication.

The writing to cache module, shown in Figure 5.18, shows what happens when a write request arrives at the kernel. The kernel caches the write requests. With that method, a read request to the same memory location does not need to read from memory again. The read address is thus used as an index to the cache to retrieve the data. A cache hit occurs when the cache's data and the read command have the same address.

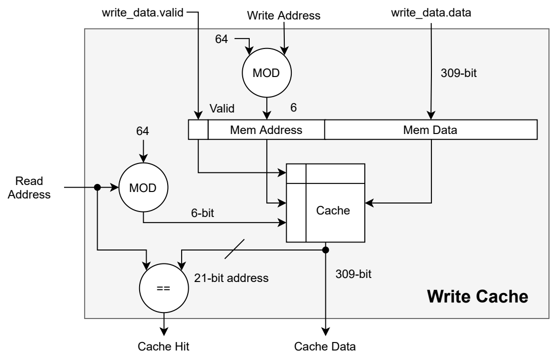


Figure 5.18: Writing to QDR data cache.

The selecting queues module, shown in Figure 5.19, stores read-requests in the stall and read queue. It inserts incoming read requests into the read queue immediately when they arrive. Meanwhile, the stall queue only holds stalled read-requests, which arrive when it detects the QDR memory is still waiting for that same address's response.

The stall logic module, shown in Figure 5.20, controls whether to stall an incoming

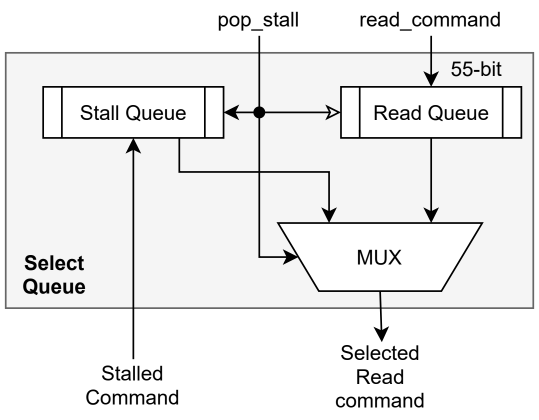


Figure 5.19: Grabbing correct request from the read.

read request. The primary function is to keep tracks of the stalled requested addresses. When a write request with the same address arrives, the module sends a request to pop data from the stall queue. The requests in the stall queue have precedence over those in the read queue. This behaviour occurs during a write request, meaning that the write request just wrote its data to the cache. For those requests, the delay is minimum, as there is no QDR interaction.

The stall module stores the stalled addresses inside a shift register. It compares the write request address to the shift register, which results in a vector of matches. The module compares both the vector and the mask. The mask tells the location of the stalled requests inside the shift register. All the bit positions that do not match the mask are removed and checked whether it found a match. It sends that match to the stall queue and back into the calculate mast module.

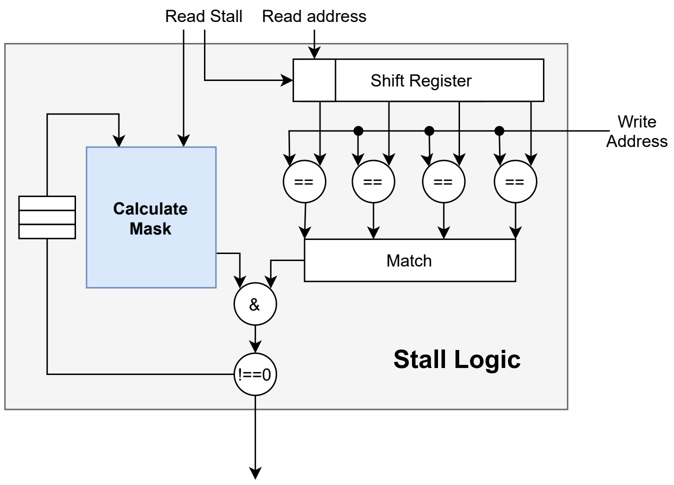


Figure 5.20: The stall logic.

The calculate mask module, shown in Figure 5.21, controls the mask calculation of the stall function. The module calculates the mask based on a correction and the stall value from a read request. It subtracts these two values because the correction value reduces the counter, while the stall increases it. Afterwards, it adds the result of the operation to the counter. The counter shifts the maximum mask value to the left, adding 0's to its right. The counter's default value is the mask's size, resulting in an all-zero mask at the start.

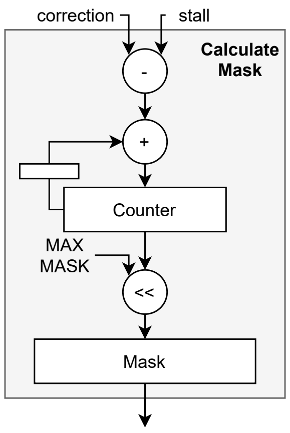


Figure 5.21: Calculate stall mask.

The control logic module, shown in Figure 5.22, shows what happens during a read-request. First, it stores the read address into a shift register, which holds previous read commands' addresses. In Section 3.3.2.5, we determined that, depending on the frequency, it takes 17 to 30 cycles to read data from QDR memory. We use this number to get a delay between sending the read-request and writing data back to memory. This delay is approximately 40 cycles in total, where it stores the read address for all those cycles. The logic in Figure 5.23 determines the following three flags:

to Kernel: The data from the cache or write-request is immediately sent to the correct kernel in the case of a cache-hit or data received from QDR memory.

To QDR: This flag determines to send a read-command to the QDR memory. Furthermore, it inserts the same command into the busy queue. This queue contains all current read-request that await data from the QDR memory. All requests return sequentially, which is why a queue is optimal in this case.

To Stall: This flag determines that the kernel should stall the current read-command. In this case, the address of the read-request previously been sent to the QDR. However, there was no response as of yet.

Figure 5.23 shows the result of how we derived the flags using the truthtable from Appendix A.1.1 and the Karnaugh maps in Appendix A.1.2.

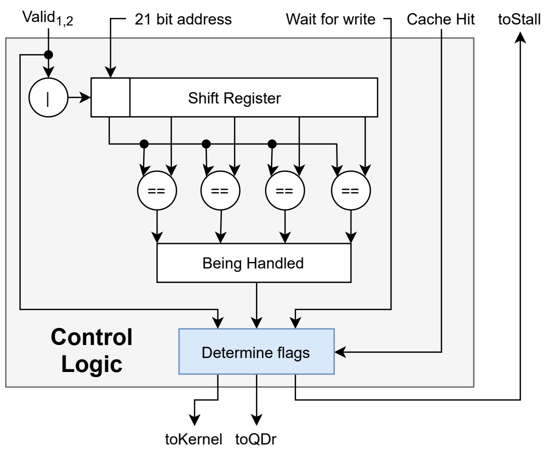


Figure 5.22: Structure of the memory communication.

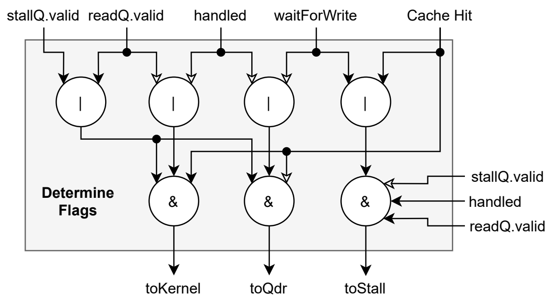


Figure 5.23: Determining flags.

5.4.2.3 Resource Utilization

Table 5.16 shows the synthesized resource utilization for this kernel.

Kernel	**ALMs**	**BLOCK MEM**
QDR Communication	1,540	14

Table 5.16: Resource Utilization of the QDR communication.

5.4.3 Bloom filter

This section describes the bloom filter's implementation discussed in Section 3.3.5.2, which stores the matched XOR patterns.

5.4.3.1 Input & output

The bloom filter input is the QDR memory kernel data (Section 5.4.2). The input holds the bloom filter that needs to be updated. Table 5.17 represents the input of the bloom filter while Table 5.18 shows the different outputs of this kernel. One of the outputs is towards the QDR memory, while the other one sends the bloom filter data to the CPU when an overflow occurs.

Field	**Bits**
Address	21
Source Address	32
Data	288

Table 5.17: Bloom filter input.

Field	**Bits**
Address	21
Source Address	32
Data	288

Table 5.18: Bloom sfilter output.

5.4.3.2 Kernel Function

The kernel updates the bloom filter inside the QDR data. The kernel segments the 288 data bits into 72 × 4-bit buckets. Figure 5.24 shows the structure of the kernel.

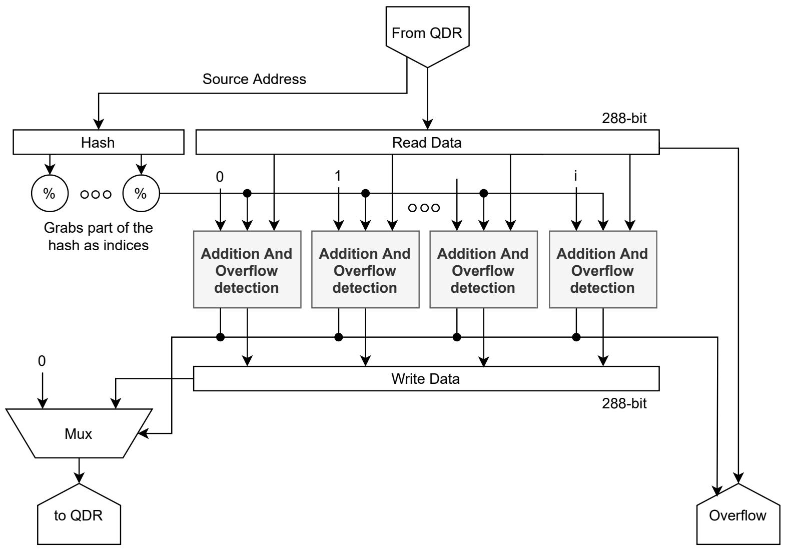

From QDR
Source Address
288-bit
Hash
Read Data
%
ooo
%
0
1
ooo
i
Grabs part of the hash as indices
Addition And Overflow detection
Addition And Overflow detection
Addition And Overflow detection
Addition And Overflow detection
0
Write Data
288-bit
Mux
to QDR
Overflow

Figure 5.24: Structure of the bloom filter.

The hashing module uses the Jenkins hash on the source address, resulting in a 32-bit random number. This number is separated into two 16-bit chunks and are the indices that select the buckets. Using the number in this way allows using one hash function instead of two, reducing complexity while maintaining randomness.

Addition and overflow detection uses the logic in Algorithm 3 shows. As previously mentioned, we first calculate the indices by taking two values, $16 - \text{bit} \mod 72$, from the hash. Each bucket gets a distinct number. It compares the indices to that number and adds that value to update the bucket. Meanwhile, a bucket overflows if the input bucket was at its maximum value before its value increased.

The data written back to the memory contains the updated values from Algorithm 3. During an overflow, the kernel sends a write-request containing zeros to the QDR memory and sends the kernel's input to the CPU to keep track of it.

5.4.3.3 Resource Utilization

Table 5.19 shows the synthesized resource utilization for this kernel.

5.4.4 QDR receiver

The QDR receiver kernel is fairly simple. After running the design for a while, we want to see what remains inside the QDR memory. The function of this kernel is to send any

```
Input: Memory Address, Source Address, Data
Result: Changed data, current data
K = Hash(Source Address);
Indices=[];
Overflow[];
Update[];
foreach 16-bits in K do
    Indices.add($k_i$ mod 72);
end
foreach i in Buckets do
    isIndice = (i in indices);
    Update[i] = data[i]+isIndice;
    Overflow[i] = isIndice & (data[i]==MAX);
end
```

Algorithm 3: Bloom filter functionality.

Kernel	**ALMs**
Bloom filter	1,437

Table 5.19: Resource Utilization of the bloom filter.

remaining data to the CPU.

5.4.4.1 Input & output

The input to this kernel, shown in Table 5.20, is data received directly from the communication kernel (Section 5.4.2). The output of the kernel is similar. However, every field corresponds to one that the CPU can understand. Table 5.21 shows the output, and it adds padding so it fits the 128-bit words required for PCIe communication defined in Section 2.4.4.3.

Field	**Bits**
Address	21
Source Address	32
Data	288

Table 5.20: The receiver input.

Field	**Bits**
Address	32
Source Address	32
Data	288
Padding	32

Table 5.21: The receiver output.

5.4.4.2 Kernel Function

The kernel checks whether the data from memory is not equal to zero. When the data contains zero bits, it means that there was no data match for that specific pattern. Otherwise, the kernel sends the data to the CPU for logging purposes.

5.4.4.3 Resource Utilization

Table 5.22 shows the synthesized resource utilization for this kernel.

Kernel	**ALMs**
QDR Receiver	62

Table 5.22: Resource Utilization of the QDR receiver.

5.5 Resource cost summary

Table 5.23 shows the combined resource cost of the individual kernels. This table only considers the individual kernel and not additional resources, such as memory FIFO buffers required to transfer data between the kernels. Furthermore, it considers only a single XOR computation module.

Kernel	**ALMs**	**BLOCK RAM**	**MULTs**
Field cache	2,879	12	0
XOR module	10,434	0	0
XOR kernel	209	8	0
Fingerprint kernel	479	0	0
Address generator kernel	241	0	1
QDR communication	1,540	14	0
Bloom filter	1,437	0	0
QDR receiver	62	0	0
Total	17,281	34	1

Table 5.23: Resource utilization summary.

5.6 Experiments

We executed these experiments in the following fashion. We send the network packets three times for each experiment to see if the data is consistent across iterations.

The sequence of the experiment was as follows:

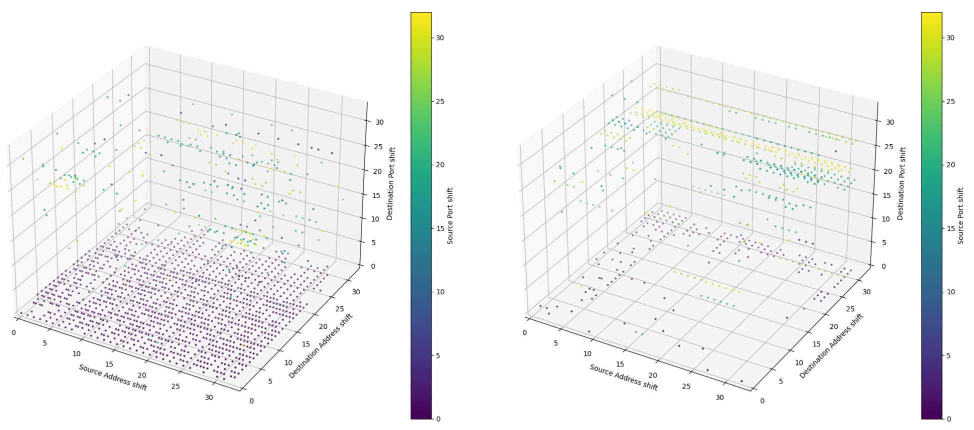
- Start the design. This phase has two parts. First, it clears the QMEM by writing zeros to it. Afterwards, it loads the network design onto the DFE.
- Reads out QMEM to see whether there was anything inside of it
- Send network data from the 'build server'.
- Reads out QMEM to get the patterns still in memory.
- Note down the metrics. These are the patterns still inside the QMEM. We write those to disk.

The design used in the experiments used 22 XOR modules instead of the 21 described in the results. However, we do not believe there was any difference in the behaviour of the design due to this.

When we performed the experiments, our replayed network traffic could not reach the intended 10Gbit/s speed. The highest speed we could achieve was 238 Mbits/s which is a small fraction of the speed. During the tests, the results of the output did vary. However, they did not vary significantly to suspect that some patterns did not get a turn. Therefore we did not test different network speeds.

5.6.1 Send all different XOR patterns

The first experiment was to determine what the DFE design does with multiple patterns from different sources. In this case, we send packets with all the patterns to see which fields get detected. From these results, we see which kind of pattern results in erroneous data and contemplate why. Our first packet file (pcap) tests all $1,185,921$ possible patterns from a different source address.



(a) Below threshold.

(b) Above threshold.

Figure 5.25: One pattern accross thresholds.

Figure 5.25 show the detected patterns below and above a specific threshold. We determine the threshold by the number of send patterns. As we made clear in Section 5.4.3, every match generates two points inside a bloom filter. The number of patterns we send was 1 per packet, meaning the threshold value should be 2. Figure 5.25a shows the plot for that scenario. Meanwhile, Figure 5.25b plots a graph where the matches exceeded the threshold.

To make sense of that data, we use table 5.24. This table shows the number of matches for all the number of times we experimented. From the table, we see that most found patterns were above our threshold. However, we can also safely see that not all patterns were detected. By only using the data below the threshold, our design detected $1,581$ patterns at most. As we send $1,185,921$ patterns, with $1,581$ getting detected, we have a successful pattern detection of 0.1%.

Trials # ranges	1	2	3
#Detected patterns	5,143	3,595.5	5,422.5
#Below threshold (2)	1,581	1,340	1,468
#Above threshold (2)	3,562	2,255.5	3,954.5

Table 5.24: Patterns found inside the first pcap file.

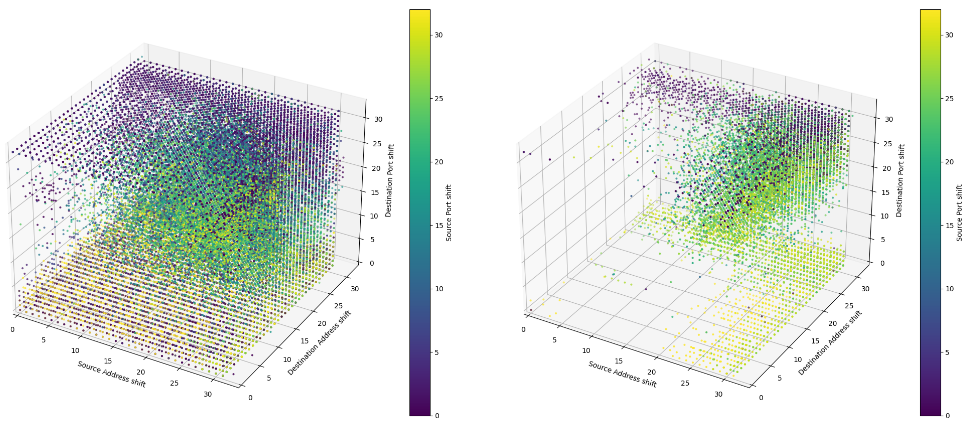
Additionally, there are many patterns detected above our threshold. To illustrate why this happens, we use Table 5.25, which groups shift values of a similar range together. From these ranges, we see that most patterns above our threshold have shift values in higher ranges. These shift values generate more patterns as the higher the shift value, the more likely an accidental match occurs due to more zero values being available.

Trials # ranges	1	2	3
Lower [0 - 11)	16.8%	25.8%	18.5%
Middle [11 - 22)	19.1%	9.3%	19.2%
Higher [22 - 33)	64.1%	64.9%	62.3%

Table 5.25: Percentage of hits on specific shifts above threshold.

5.6.2 Send all different XOR patterns 16 times.

The second experiment tests writing multiple patterns to disk. Furthermore, it also functions as a backup test to see whether a higher percentage of patterns get detected. From here, we can more easily deduce which type of pattern generates more data than expected.



(a) Patterns below threshold 32.

(b) Patterns above threshold 32.

Figure 5.26: Every pattern 16 times accross thresholds.

Figure 5.26 shows the scatterplot of data using the threshold value 32. For this plot, we used the following assumptions. Every pattern creates two additions inside the bloom filter. After the design writes a bloom filter to the disk, we should add two to the detected patterns. This addition is to keep the overflow that causes a write to disk to happen into account. Table 5.26 shows all found patterns divided by 2.

Trials	1	2	3
#Detected patterns	2,031,254.5	1,988,871.5	2,050,987.5
#In expected range	1,396,252.5	1,396,082	1,397,736
#Above expected range	635,002	592,789.5	653,251.5

Table 5.26: Percentage of hits on specific shift ranges.

Figure 5.26a shows everythin below the threshold. Comparing it to our previous experiment, we see it is a lot more densely populated. Meanwhile, Figure 5.25b shows the data points above our threshold. Using Table 5.26 we can interpret the data and see that most patterns were actually below our established threshold, meaning more patterns were accurately detected. We send 16 times more patterns than the previous experiment, which is $16 \times 33^4 = 18,974,736$ packets in total. The maximum number of packets below the threshold value is $1,397,736$. Meaning, the number of detected patterns in this scenario is $\frac{18,974,736}{1,397,736} \times 100\% \approx 7.37\%$, which is definitely higher than before.

The patterns found outside of our threshold also show some interesting information. Similar to Table 5.25, Table 5.27 shows an even higher number of high shift values. This information confirms that the higher the shift values are, the more errors there are. Furthermore, this also means that patterns with a higher shift value are less accurate. Meaning, the number of computations could get reduced.

Trials	1	2	3
Lower [0 - 11)	11.8%	12.6%	11.6%
Middle [11 - 22)	12.8%	12.9%	12.6%
Higher [22 - 33)	75.4%	74.5%	75.8%

Table 5.27: Percentage of hits on specific shifts above threshold.

5.6.3 Send actual network traffic

In Section 2.1, we analysed network traffic picked up by the TU Delft data telescope. This experiment sends this network traffic to the design to determine whether packets are detected.

Table 5.28 shows the number of patterns during the trials and the source address that caused the detection. After three runs of the experiment, there were three patterns detected. However, the pattern in the last column is not always detected and has the same source address as the first pattern. Therefore, we believe that it is due to an error.

The other patterns do not have this problem. However, the source addresses used for those patterns occur more in the pcap file with many TCP retransmissions. Therefore, we believe there might be an error in the design that limits our detection rate.

Shift Pattern	$24, 16, 18, 20$	$18, 27, 23, 16$	$20, 16, 18, 24$
Trial 1	64	62.5	17.5
Trial 2	65	63.5	0
Trial 3	64.5	40.5	0
Source address	131.180.170.231	131.180.170.37	131.180.170.231
Packets in pcap	575	475	575

Table 5.28: Recorded patterns during trials of dump-1555998151.

We tested all of the network data separately, as well as after each other. Figure 5.27 shows the patterns detected in the QMEM. In the figure we can clearly see that fig. 5.27d and fig. 5.27e look very similar. Meanwhile, there is no unique pattern from the other dumps inside Figure 5.27e. This discrepancy makes us believe there might be additional errors in the design.

5.6.3.1 Errors found during experiments

During the experiments, we noticed that there were bugs in the program. First, clearing the QMEM at the start of the experiments did not clear all memory addresses. Most of the time, there was at least one address inside the QMEM before the experiment. In rare cases, there were two addresses. These specific addresses were address 1 and 2.

Furthermore, when reading the QMEM, there were errors at times. One part of the data gets cleared, while another part still contains garbage data. We assume that data from the QMEM interfaces do not arrive in the same computing cycle in some cases. In those cases, only part of the QMEM data gets returned.

Additionally, as we have seen in table 5.28, there are a lot more packets that remained undetected with those specific source addresses, which means other errors inside the design stop the code from detecting everything.

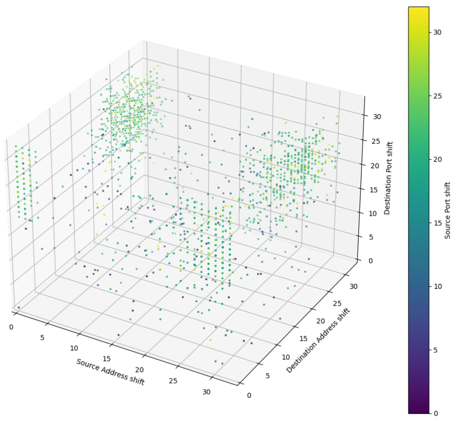
5.7 Results

This section shows the performance and utilisation of the final design.

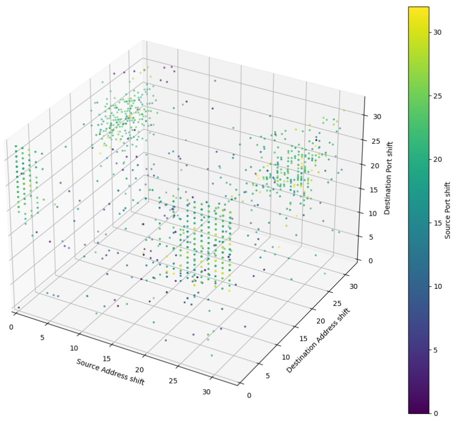
5.7.1 Final Utilisation & Performance

The final design consists of 21 XOR modules. The performance of one module is $1,089$ operations every clock-cycle at a 156.25 MHz frequency, which we found in Section 2.4.4.4. Therefore, one module performs $170,156.25$ million ($170.2 \cdot 10^9$)operations a second. With 21 modules, we can perform a total of $21 \times 170,156.25 = 3,573,281.25$ million ($3.6 \cdot 10^{12}$) operations a second. Our initial target was $98,826.75$ operations a cycle, which is $15,441,679.6875$ million ($15.4 \cdot 10^{12}$) operations a second. So, our design reaches 23.14% of the target bandwidth.

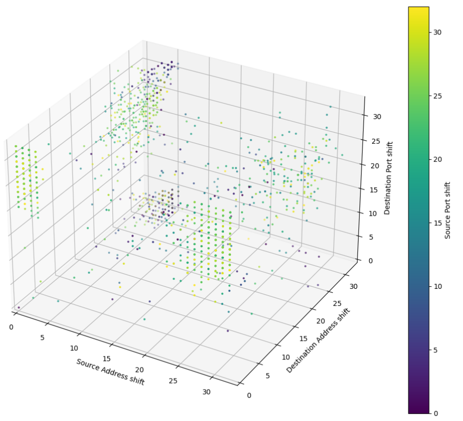
To better understand the speed of the design, we compare this design to a high-end CPU. However, it is compared purely on pattern-finding operations. As mentioned in Section 2.3.1 a CPU performs a sequence of instructions at a higher frequency. Instead



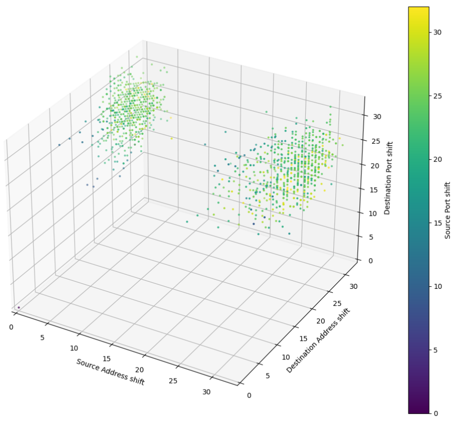
(a) Patterns in dump-1528639398.



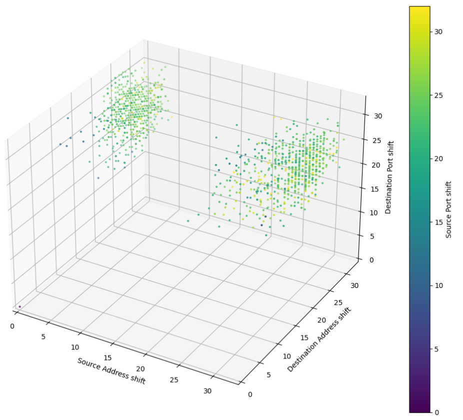
(b) Patterns in dump-1537646287.



(c) Patterns in dump-1545328787.



(d) Patterns in dump-1555998151.



(e) Patterns in all pcaps.

Figure 5.27: Patterns inside the pcap file.

of performing $1,089$ computations in one clock cycle like the FPGA, it performs multiple instructions to get the same result. The algorithm requires approximately four shift operations on a CPU, three XOR operations, and one comparison operation, totalling eight instructions. For example, a high-end Ryzen Threadripper 3990x CPU can perform 541.66 operations on each clock cycle at 4.35 GHz. This performance results in $2,356,230$ millions ($2.4 \cdot 10^{12}$) of instructions a second. Dividing those instructions by eight for the best-case scenario results in $294,528.75$ million ($0.29 \cdot 10^{12}$) pattern-finding operations a second. The performance increase based on only pattern-finding operations is approximately $3,573,281.25/294,528.75 \approx 12.13$ times as fast.

The best-case CPU scenario is not realistic, however. On a CPU-based system, there is additional latency due to memory, PCIe speed, and the hosts' network stack. The network interface needs to store the packet in the main memory, whereafter the CPU has to retrieve it from memory. Additionally, the operating system would add additional cycles where it is not performing calculations. Furthermore, this is all assuming that all instructions are executed perfectly on the parallel CPU cores.

Kernel	**ALMs**	**BLOCK MEM**	**Multipliers**
Design	194,880	326	2
Percentage	54.25	12.35	0.28

Table 5.29: Design resource utilisation with 21 XOR modules.

Conclusion & Future work 6

6.1 Conclusion

In this thesis, we investigated two research questions. In this section, we provide the answers to these questions and summarize all of our findings.

RQ1: Can FPGAs outperform conventional systems on XOR pattern detection at line rate for 10Gbit/s connections? Our original target was performing the XOR pattern detection function at 10 Gbit/s. However, during the modelling phase in Section 3.3, we found out that we can achieve only approximately 50% of the targetted bandwidth using the intended brute-force method. This is because the 359, 200 LUTs on the stratix V FPGA chip were insufficient to implement all of the necessary calculations in the available hardware. We chose to adjust our target to get as close as possible to the desired 10 Gbit/s.

In Section 5.7 we have shown how the FPGA can outperform a high-end CPU while performing the computations. From Section 3.2.2 we found that performing all our desired operations require 98, 826.75 operations a cycle. The maximum number of operations we can reach with our design is 22, 869 operations a cycle, which is $\approx 23.14\%$ of the targetted 10 GBit/s bandwidth. However, our design with 21 modules is ≈ 12.13 times faster than a CPU implementation of the same algorithm. Furthermore, our design used only $\approx 52\%$ of the available resources, so there is still potential to increase the number of calculations, increasing the bandwidth we can handle.

RQ2: What are the major limitations and opportunities of FPGA accelerators when implementing scanner detection? As mentioned above, we found out that the brute-force method could not fit on the hardware fabric of the JDFE FPGA chip. It could not because the Stratix V FPGA device does not have enough computation resources available on the chip. Our current design uses only 51% of the chip's resources and reaches $\approx 23.14\%$ of the bandwidth. However, we did not test the design at a frequency higher than 156.25 MHz. Increasing the frequency, for example, doubling it, also doubles the number of operations. If we can meet the timing without any problems, no additional resources are required. Otherwise, there is a cost for additional registers and resources to meet the timing.

The resources we utilized most in our design were the LUTs. When using an FPGA with more resources, such as the ones discussed in Section 3.3.4, it would be possible to utilize the full 10Gbit/s bandwidth. The FPGAs in that section, the Alveo U280 and VU9P chip, have more computational LUT resources with Xilinx UltraScale+ technology. The Alveo U280 card contains 1, 304K, and the VU9P has 1, 182K LUTs, compared to the 359.2K LUTs in our system. Furthermore, the U280 card uses High Bandwidth

Memory (HBM) technology, with approximately 36 times more bandwidth than our DDR3 memory. With HBM, it is possible to store more fingerprint information, and it is highly likely to give birth to hybrid solutions to finding those fingerprints, such as discussed in Section 3.2.3.

6.2 Future Work

This section discusses future possibilities for the design and how to improve it.

6.2.1 Reduce packet traffic

The whole thesis assumes that the FPGA would receive many small packets at maximum network speed. However, this is not realistic. We can assume that the TCP traffic that arrives at the FPGA uses at least two-thirds of the TCP handshake mentioned in Section 2.1. Filtering these additional packets would significantly reduce the number of received packets. For example, the filtering keeps track of the packets based on the network addresses and port numbers. When a packet with the same characteristics arrives, it gets discarded. Assuming the worst-case scenario, namely, a fast SYN scan, skipping those packets reduces the incoming TCP packets by at least half of the current solution. Due to those packets getting skipped, there is a potential to be more operations between different packets. So, in the best case situation, the packets' arrival time can be stretched to 24 clock cycles instead of 12.

6.2.2 Use onboard DDR3 memory

The current design does not utilize DDR3 memory (LMEM). One method uses a buffer for those packets that cannot be stored on the FPGA anymore as every XOR module is used. Storing the data in memory allows the design to validate the packets during less busy times. Using a two-level buffer would be the best solution. Here, the first buffer uses the fast QMEM to store a small amount of packet data. The second level uses the LMEM, where all the data removed from the QMEM gets stored. These buffers allow for reasonably fast fetching packet data from the first buffer, while older entries get fetched from the second level.

6.2.3 Using multiple FPGA's

The other method is using multiple FPGAs functioning in parallel. Additional FPGAs increase the monetary cost but allow for the FPGA to function on 10 Gbit/s. For example, we use two FPGA's that each performs $592,961$ operations. That would come down to $49,413$ operations a cycle, using the cycles between packets from Section 2.4.4.4.

6.2.4 Using more modules

In Section 5.7 we have seen our design's resource utilization does not use all of the resources for its 21 modules. Our design utilizes 54.25% the LUT resources, which are $194,880$ LUTs in total. In Section 5.5 we see the utilization of one XOR module is approximately $10,434$ LUTs, which is the main source of the costs. However, we know this does not represent the LUTs used for each module as $10,434 \cdot 21 > 194,880$ LUTs. In the best-case scenario, we could use the remaining LUTs to increase the number of these modules to two times as much. These 42 modules would then, in turn, double the number of operations.

6.2.5 Doubling the frequency

We did not test our design on frequencies higher than $156.25MHz$. However, if we double the frequency, it could theoretically perform twice the number of operations in the same amount of time. There are some caveats to this, however. A higher frequency means lower retention and propagation times for data inside the design, leading to timing issues. However, adding additional registers that result in pipeline stages in between relaxes the timing constraints. As a tradeoff, however, it increases the designs resource utilization.

6.2.6 Using a different FPGA

As previously mentioned in our conclusion, our design was ultimately LUT dependent. So using the Alveo U280 or VU9P chips would give us a lot more computational LUT resources. The Alveo U280 card contains $1,304$K, and the VU9P has $1,182$K LUTs, compared to the 359.2K LUTs in our system. Using one of these different chips would already allow the design to perform more operations in a second. Additionally, the U280 card has High Bandwith Memory (HBM) technology, giving it up to 36 times more bandwidth than our DDR3 memory. This type of memory would even allow for more hybrid solutions to mining the fingerprint.

6.2.7 Compressing the packet data

Currently, the stored packet data is 128-bits in total without any compression. By compressing the packet fields, it might be possible to store more packets in the available memory. So, there is a possibility to store more packets in memory than initially thought.

Bibliography

[1] "Ieee standard for ethernet," *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*, p. 119, Aug 2018.

[2] "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc791.txt

[3] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://rfc-editor.org/rfc/rfc793.txt

[4] C. Doerr, *Network Security in Theory and Practice*, 2018.

[5] S. Chikkagoudar, K. Wang, and M. Li, "Genie: A software package for gene-gene interaction analysis in genetic association studies using multiple gpu or cpu cores," *BMC research notes*, vol. 4, p. 158, 05 2011.

[6] "High-performance alm and interconnect," Apr 2019. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/products/fpga/features/stx-architecture.html

[7] "Jdfe." [Online]. Available: https://www.maxeler.com/products/jdfe/

[8] "Packet flow accelerator diagnostics software," Jan 2021. [Online]. Available: https://www.juniper.net/documentation/us/en/software/junos/network-mgmt/topics/topic-map/packet-flow-accelerator-diagnostics-software.html

[9] K. L. Lueth, "State of the iot 2018: Number of iot devices now at 7b – market accelerating," Aug 2018. [Online]. Available: https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/

[10] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," 2010.

[11] H.J.Griffioen, "Scanners, discovery of distributed slow scanners in telescope data," Master's thesis, Technical University Delft, 2018. [Online]. Available: http://resolver.tudelft.nl/uuid:dcb1669d-d81e-4aa3-bbd1-65049c3209c5

[12] G. Lyon, "Port scanning techniques: Nmap network scanning." [Online]. Available: https://nmap.org/book/man-port-scanning-techniques.html

[13] "Qfx5100 application acceleration switch," Oct 2015. [Online]. Available: https://www.juniper.net/assets/us/en/local/pdf/datasheets/1000532-en.pdf

[14] M. Dabbagh, A. J. Ghandour, K. Fawaz, W. E. Hajj, and H. Hajj, "Slow port scanning detection," in *2011 7th International Conference on Information Assurance and Security (IAS)*, Dec 2011, pp. 228–233.

[15] Chunmei Yin, Mingchu Li, Jianbo Ma, and Jizhou Sun, "Honeypot and scan detection in intrusion detection system," in *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No.04CH37513)*, vol. 2, May 2004, pp. 1107–1110 Vol.2.

[16] A. Dainotti, A. King, K. Claffy, F. Papale, and A. Pescapé, "Analysis of a "/0" stealth scan from a botnet," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 341–354, 2015.

[17] I. Xilinx, "Ultrascale architecture configurable logic block," Feb 2017. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf

[18] ——, "Ultrascale+ fpga product tables and product selection guide." [Online]. Available: https://www.xilinx.com/support/documentation/selection-guides/ultrascale-plus-fpga-product-selection-guide.pdf

[19] Mike Wissolik, Darren Zacher, Anthony Torza, and Brandon Day, "Virtex ultrascale+ hbm fpga: A revolutionary increase in memory performance," July 2019. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp485-hbm.pdf

Appendix A

A.1 QDR logic

A.1.1 Truth table

readQ.valid	**stallQ.valid**	**waitForWrite**	**handled**	**cache-hit**	**toKernel**	**toStall**	**toQDR**
0	0	0	0	0	**0**	**0**	**0**
0	0	0	0	1	**0**	**0**	**0**
0	0	0	1	0	**0**	**0**	**0**
0	0	0	1	1	**0**	**0**	**0**
0	0	1	0	0	**0**	**0**	**0**
0	0	1	0	1	**0**	**0**	**0**
0	0	1	1	0	**0**	**0**	**0**
0	0	1	1	1	**0**	**0**	**0**
0	1	0	0	0	**0**	**0**	**1**
0	1	0	0	1	**1**	**0**	**0**
0	1	0	1	0	**0**	**0**	**1**
0	1	0	1	1	**1**	**0**	**0**
0	1	1	0	0	**0**	**0**	**1**
0	1	1	0	1	**1**	**0**	**0**
0	1	1	1	0	**0**	**0**	**1**
0	1	1	1	1	**1**	**0**	**0**
1	0	0	0	0	**0**	**0**	**1**
1	0	0	0	1	**1**	**0**	**0**
1	0	0	1	0	**0**	**0**	**1**
1	0	0	1	1	**0**	**1**	**0**
1	0	1	0	0	**0**	**0**	**1**
1	0	1	0	1	**1**	**0**	**0**
1	0	1	1	0	**0**	**1**	**0**
1	0	1	1	1	**0**	**1**	**0**
1	1	0	0	0	**x**	**x**	**x**
1	1	0	0	1	**x**	**x**	**x**
1	1	0	1	0	**x**	**x**	**x**
1	1	0	1	1	**x**	**x**	**x**

1	1	1	1	1	x	x	x
1	1	1	1	0	x	x	x
1	1	1	0	1	x	x	x
1	1	1	0	0	x	x	x

Table A.1: QDR logic truth table.

A.1.2 Karnaugh Maps

To get the most optimal function, we used a truth table which can be found in Appendix A.1.1. In the karnaugh maps, we use the follwoing symbols for the flags:

X_0 is handled

X_1 is cache-hit

X_2 is stallQ.valid

X_3 is waitForWrite

X_4 is readQ.valid

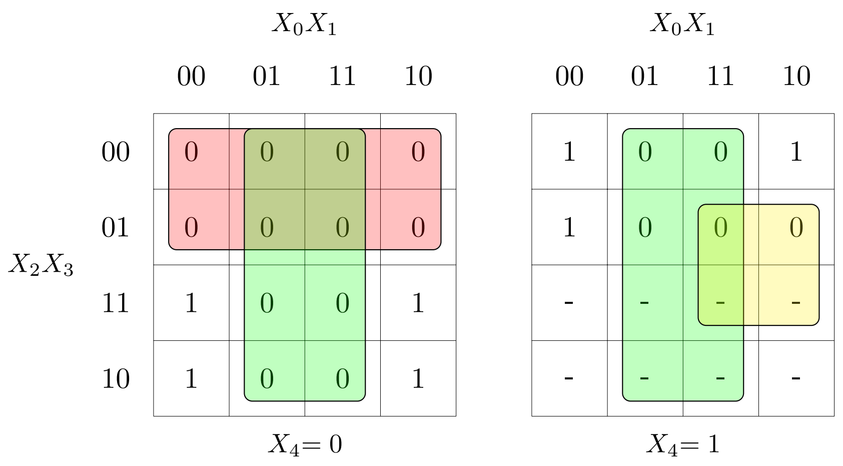


Figure A.1: To QDR.

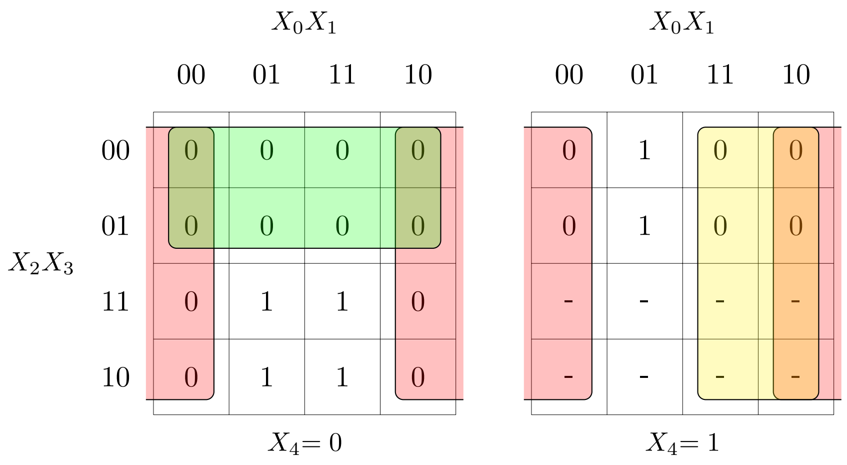
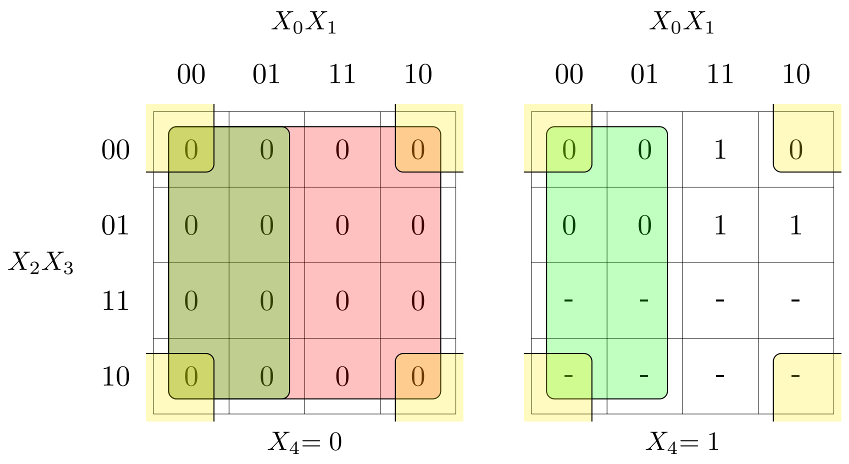


Figure A.2: To Kernel.



X_2X_3 \ X_0X_1	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

$X_4 = 0$

X_2X_3 \ X_0X_1	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	-	-	-	-
10	-	-	-	-

$X_4 = 1$

Figure A.3: To Stall.