An Elliptic Curve Cryptography Acceleration Core for OpenVPN on an FPGA Softcore

N. J. Versluis

Technische Universiteit Delft

TU Delft Delft University of Technology

Challenge the future

An Elliptic Curve Cryptography Acceleration Core for OpenVPN on an FPGA Softcore

by

N. J. Versluis

in partial fulfillment of the requirements for the degree of

Master of Science
in Embedded Systems

at the Delft University of Technology,
to be defended publicly on Tuesday June 30th, 2020 at 14:00.

Supervisor:	Dr. ir. A. J. Van Genderen	
Thesis committee:	Dr. ir. J. S. S. M. Wong,	TU Delft, CE
	Dr. S. Picek,	TU Delft, Cyber Security
	Ir. T. van Leeuwen,	Technolution

Q&CE-CE-MS-2020-07

TU Delft Delft University of Technology

Abstract

Elliptic Curve Cryptography (ECC) performance is a major performance bottleneck when serving many VPN clients from a single server on a low-frequency FPGA softcore CPU. Using an area-efficient Elliptic Curve Point (ECP) multiplication accelerator core on the same FGPA, a much higher amount of clients can be served using the same FPGA chip.
Using the accelerator core, the obtained speedup ranges from 1.6x in a suboptimal configuration up to 7x with a configuration that maximizes the use of ECC when connecting new clients to the server. In this optimal configuration, the total amount of clients that can be served by a single OpenVPN server increases from 80 in the base case, to 350 in the accelerated case.

Contents

1 Introduction

1.1. Background Information

1.1.1. OpenVPN-NL

A Virtual Private Network (VPN) is a piece of software used to connect to a remote network from another network. It is often used to securely access business networks over the internet, so it is possible to access devices on the local network, while not being physically connected to that network. OpenVPN is a popular free, open-source VPN server and client application used to set up a VPN connection between a server and (multiple) client(s). It consists of two layers: The control path and the data path. Both paths run on the same UDP connection, but the OpenVPN software has a multiplexer inside that can distinguish between these two paths.

The control path is used to set up the connection. First, a TLS connection is established. Then, the keys to be used for the data path are exchanged using this TLS connection. After that, the connection switches to the data path, which will handle the actual network communication. In order to be able to serve a large amount of clients with a single server, both the data path and control path must be fast enough to handle the load of many simultaneous connections. The data path needs to be able to encrypt and decrypt all incoming and outgoing data fast enough, and the control channel must be able to set up connections quickly. This thesis focuses on the scenario where the data path is already sufficiently fast, but the control path can not establish connections to new clients fast enough.

A control path bottleneck could occur for the following two reasons: First, in order to achieve forward secrecy, OpenVPN encryption keys are valid for one hour maximum and are also refreshed for each new session. This means that every hour, the control path has to reconnect to each currently connected client, thus establishing a new TLS connection for each client. Therefore, on a low-performance CPU, this connection time can be a limiting factor for the maximum amount of connected clients. Second, in case of a server restart, all clients will attempt to reconnect simultaneously, causing a lot of control paths to be created at once. In this case, the control path creation needs to happen quickly in order to prevent long timeouts on the client side.

Transport Layer Security (TLS) is a combination of cryptographic protocols designed for secure communication over an insecure network. TLS mainly consists of four parts: Key exchange, block cipher, authentication and integrity checks.
Key exchange is the method used to exchange the used encryption keys.
The block cipher indicates the way in which the sent encryption keys will be used to encrypt data.
Authentication is used to make verify that the entity sending or receiving the data is actually the entity that they claim to be.
Integrity checks use a message authentication algorithm to verify that the originally sent message has not been tampered with by a third party during transmission.
All of these factors can be easily determined when looking a the name of the used TLS ciphersuite. For example, a ciphersuite used by OpenVPN is ECDHE-RSA-AES256-GCM-WITH-SHA384.

In this example, the ciphersuite indicates that ECDHE is used as the key exchange method, RSA used for authentication, AES256-GCM used as block cipher, and finally SHA384 as its message authentication algorithm.

OpenVPN-NL is a fork of the OpenVPN project, maintained by Fox-IT. Its main distinguishing features from the regular OpenVPN software suite are more strict cryptography requirements (less secure protocols and ciphers have been removed), and the use of mbedTLS instead of OpenSSL for its cryptography library.
mbedTLS is an open-source cryptography library written in C. The goal of the project is to provide efficient cryptography support to embedded platforms, where performance often comes at a premium. The supported ciphers for OpenVPN-NL include RSA for both key exchange and authentication, ECDH for key exchange and SHA for integrity checking. Authentication algorithm ECDSA is unfortunately omitted, but still tested in this thesis, since it benefits greatly from ECP acceleration and could be included in a later release of OpenVPN-NL.

1.1.2. Elliptic Curve Cryptography

Elliptic Curve Cryptography (ECC) is a cryptography method based on Elliptic Curves. An elliptic curve is an algebraic curve defined by an equation of the form $y^2 = x^3 + ax + b$. When projecting this curve onto a finite field, this mathematical system allows for operations using large, modular integers that can be used for cryptography. The base concept is that point multiplications, that is multiplying a coordinate with a scalar, on the curve are calculated with acceptable computational cost, but very difficult to reverse.
This means that one can easily calculate $Y = n * X$, but if one has X and Y, n is near impossible to retrieve within a short amount of time.
These point multiplications can take long to calculate by simply adding a point to itself n times for large n, but many algorithms aimed at reducing this computation time exist. Most of these algorithms consist of two more basic ECP calculations: ECP doubling ($Y = 2X$), and ECP adding ($Y = X + Z$).
Compared to RSA, the numbers needed for a similar amount of cryptographical strength are significantly smaller[1], making ECC popular on embedded systems, where performance often comes at a premium because of power usage, cost or heat.

ECC is performed on an elliptic curve, but many curves of such a type exist. Therefore, selecting curves that are cryptographically secure has been an important point of discussion. The North American National Institute of Standards and Technology (NIST) has released its own specifications of certain elliptic curves that are recommended by the institute. These elliptic curves are therefore known as the NIST curves.
One of these curves, using 384-bit integers is called NIST P-384. Since the largest, and therefore also most cryptographically secure curve supported by OpenVPN-NL is the NIST P-384 curve, this thesis will focus on accelerating cryptographical operations on this curve. The only other curve supported by OpenVPN-NL is the NIST P-256 curve, which is essentially the same type of curve, but with 256-bit numbers, and therefore less secure.

1.1.3. Side-channel attacks

Side-channel attacks are attacks on the cryptography that try to obtain data on the used private key by looking at indirect data, such as power usage and timing. For instance, when looking at the amount of time between a response of the cryptography suite, one might determine that the key is even if it takes a little more or less time, if the used cryptographical calculation takes a different amount of time on even or odd numbers. When designing a cryptography accelerator, it is important to keep side channel attacks in mind, because a faster cryptography calculation that is not secure, would be meaningless.

1.1.4. Large-number mathematics

Cryptography algorithms use large integer numbers. For instance, in NIST P-384, 384-bit numbers are used.
However, most modern CPUs have either 32-bit or 64-bit registers. Therefore, cryptography suites

often use Multiple-Precision Integers (MPIs). mbedTLS also makes use of MPIs for its large number representation.
These MPIs are essentially arrays of numbers that are the maximum size of the aforementioned registers, used to represent large numbers. However, since the CPU ALU can not fit these large numbers, algorithms are needed to perform basic operations on these MPIs, such as addition, multiplication and shifting.
A Field Programmable Gate Array (FPGA) does not have register size limitations imposed on them, so they have great potential for acceleration of the large-number mathematics that are often at the core of cryptography algorithms.

1.1.5. Platform

The softcore CPU platform used in this thesis is the Frenox. This is a proprietary RISC-V softcore CPU, developed by Technolution[1] that can be programmed onto an FPGA. RISC-V is a free, open-source Instruction Set Architecture (ISA) specification, that can be used to create processors that adhere to the RISC-V standard and thus can run code compiled for RISC-V.
The Frenox-S core used in this thesis runs at 50 MHz, adhering to the RV32IMA spec, which means it is 32-bit, and has support for base Integer instructions, integer Multiplication and division, and Atomic instructions. A 100 MHz version of the core also exists, but it is not compatible with the used development board.
The aforementioned development board is a Terasic Cyclone V Starter Kit, containing an Intel (formerly Altera) Cyclone-V 5CGXFC5C6F27C7N FPGA chip.

In order to estimate performance with different software and hardware configurations, a Frenox emulator called Fremu is also available. Fremu is a PC program that is made to emulate behaviour of Frenox accurately. This means that, if something runs for 10 seconds in *Fremu time*, one can expect it to also take 10 seconds on Frenox. The only exception is cache behaviour, which always assumes a cache hit, which is an unrealistic scenario, so performance might actually be a little slower than Fremu suggests.

It should be noted that Fremu time is not equal to real time. This means that, when profiling a certain program on Fremu, the profiled code might return a run time of 10 seconds, but the emulator could run for 20 seconds of real time. In this case, that means that, on a real Frenox core, this operation would have taken 10 seconds.

This means that in order to make Fremu time close to real time, the PC running Fremu determines the frequency that Fremu can maximally emulate near (but not completely at) real time.
A fast PC might emulate a Frenox running at 20 MHz in real time, while a slower PC might cap out at 5 MHz. Running close to real time is necessary, since OpenVPN communicates with the outside world, and running too slow or fast might cause connections to time out. Real time in- and output is also convenient for user interaction with the emulator.

1.2. Research question

This work is aimed at finding a way to increase the amount of simultaneous OpenVPN-NL channels that can be maintained on a 100 MHz Frenox softcore.
The assumed scenario is that the amount of clients that can be served is bottlenecked by the control channel of OpenVPN.
The maximum amount of clients that can be served simultaneously is defined as the amount of clients that can connect within five minutes, since this emulates the scenario of a full reboot where all clients have to reconnect at once, without timing out too often.

[1] https://www.technolution.eu

1.3. Related Work

While there are papers studying OpenVPN or mbedTLS performance[2][3][4], the authors of those works focus on throughput, rather than control channel performance and maximum amount of connected clients. Furthermore, the authors only assess the performance, but do not identify the bottlenecks that use the most computational time when running OpenVPN. Also, none of the authors give suggestions for increasing OpenVPN performance.

Elliptic Curve Cryptography has been in use for some time now, so of course there are already quite a few research papers relevant to ECC acceleration. Many area-efficient and fast designs have been proposed already:
In [5], the authors describe an ECP multiplier that is reasonably area-efficient and quite fast for 191-bit keys over a GF(191) curve. This accelerator uses a Karatsuba-Ofman multiplier for large integer multiplications, and the Montgomery point multiplication algorithm. Their results show a design using about 18k slices on a Xilinx XCV2600E platform, with about 63 us of computation time. However, this design might grow significantly larger when using 384-bit numbers for the curve.
The authors of [6] implemented an accelerator design on a Cyclone V using around 29k ALMs, and achieved a computation time of around 3.64ms with a frequency of 100MHz, using a Karatsuba-Ofman multiplier and using an RSD-based prime field for 256-bit curves. However, this design would be too large to fit on the used FPGA in this thesis. In [7], the Toom-Cook algorithm is used for multiplication, resulting in a design using 43K LUTs on a Virtex-4 FPGA.
The author of [8] explores the use of DSPs for cryptographical computations, and the modular multiplier presented in design has also been implemented for the accelerator used in this thesis.
There are many more papers still that focus on ECC acceleration [9–18]. However, none of the authors have ever tested their accelerators using actual software interfacing with the accelerator itself, which this thesis will also take into account by modifying the OpenVPN-NL source code to actually use the accelerator.
This of course means that an interface to handle communication between the FPGA, Linux and OpenVPN has to fit in the FPGA as well.
The NIST itself provides a reference document [19] with recommended algorithms for ECC computations on NIST elliptic curves.

1.4. Contents

In this thesis, first, in chapter 2, performance of OpenVPN on Frenox is analyzed, after which bottlenecks are identified. Furthermore, communication speed between Frenox and a theoretical accelerator core is examined in order to evaluate communication overhead of a potential accelerator.
Then, multiple accelerator scenarios are simulated in order to estimate the performance benefit of accelerating certain pieces of code. After that, an accelerator type is chosen. In chapter 3, the design of both the communication protocol and the accelerator core are presented. Furthermore, the actual implementation of the accelerator core is discussed. In chapter 4, the results of the implemented accelerator are presented for multiple scenarios, and the resource usage is shown.
Finally in chapter 5, the conclusion of this thesis is presented.

2

Performance Analysis

In this section, the performance of OpenVPN-NL is analyzed, with the objective of identifying bottlenecks suitable for acceleration.

2.1. Setup

In order to benchmark OpenVPN-NL performance, the following setup is used:

2.1.1. Devices

For the measurement setup, two identical PCs are used. Their specifications can be fond in table 2.1. One PC is set up to hold an OpenVPN server, running either natively, or emulated on a RISC-V platform.

Table 2.1: Server and Client PC specifications

	Server PC and Client PC
CPU	4-Core Intel i5-3570 @ 3.4 GHz (x86-64)
RAM	8GB DDR3
Storage	480GB SATA SSD
Network	Gigabit NIC, connected to Fast Ethernet (10/100) switch
OS	Ubuntu 18.04 LTS x64

The Fremu emulator is set to run at 7.5 MHz in order to make console output seem real-time, since that is the maximum frequency the Server PC can handle without losing close to real-time behaviour. Its specifications can be found in table 2.2.

Table 2.2: Fremu specifications

	Fremu
CPU	1-Core Frenox @ 7.5 MHz (RV32IMA)
RAM	512MB
Network	NAT, static IP
OS	Linux 4.15.0 RV32, Buildroot, BusyBox

A second PC is used to run the clients that will connect to the OpenVPN server. Each client will run inside its own Virtual Machine (VM), with each VM being assigned a single CPU core and 768MB of RAM. Detailed specifications of the VM can be found in table 2.3. The program used for virtualization is VirtualBox (6.0). With 1 core being reserved for the host OS, the client machine can run up to three client VMs simultaneously.

2.1.2. Network

Both the server and client PC are connected to the same internal network, on the same 100 mbit/s ethernet switch. The client VMs all have their own IP and MAC address, with the IP range being the

Table 2.3: Client VM specifications

	Client VM
CPU	1-Core Intel i5-3570 @ 3.4 GHz (x86-64)
RAM	768MB
Storage	10GB SSD
Network	Bridged adapter, unique MAC address
OS	Debian 9 Xfce x64

same as that of the client PC, because their virtual adapters are bridged.

The Fremu gets assigned an IP address in a different range from the server PC, since it uses NAT for its virtualized network adapter. In order to bypass this, the server PC is configured with IP forwarding so it can direct packets to and from devices in the server/client PC IP range.

2.1.3. Software

The following software was used in order to run the measurements:

Server: Fremu compiled for x86. A 32-bit ISA was used to guarantee compatibility with the 32-bit libraries used on Frenox.
Endianness between x86 and RISC-V is identical, so no conversion has to happen for data exchanged between the RISC-V platform and the x86 server emulating the accelerator.

Frenox/Fremux: OpenVPN-NL 2.4.7, cross-compiled for RV32IMA.

Clients: OpenVPN-NL 2.4.7 without any modifications, compiled for x64.

EasyRSA 3.0.6 was used to generate certificates for the key exchange.

The used ciphersuites for OpenVPN are ECDHE-RSA-AES256-GCM-WITH-SHA384 and ECDHE-ECDSA-AES256-GCM-SHA384. They are abbreviated as ECDHE-RSA and ECDHE-ECDSA respectively further on in this document.

2.2. Measurement method

In order to establish a baseline, performance measurements without any acceleration have to be done first. These measurements were performed as follows:

On Fremu, a bash script starts up the OpenVPN server, and then sends a command to the ClientHost PC via SSH. The ClientHost PC then connects to the VMs containing the OpenVPN clients. These client machines automatically divide the amount of requested clients between each other. For example, when requesting 9 clients from Fremu, with 3 client machines, each machine will launch 3 clients.
In order to simulate the worst-case scenario, the startup of the clients is synchronized to happen simultaneously. An option to start the clients in staggered fashion, with a customizable delay between each client launch was also added. After completing the run, the OpenVPN server and client log files are copied to the Server PC for analysis.

The built-in log entries of OpenVPN were used to determine the amount of time between the first client connection request and the last client connection confirmation.
Furthermore, when the log setting was set to most verbose, using option `–verb 9`, very detailed logging data would be output, at the cost of performance. The general shape of these log files can be found in listing 2.1.
A log parser was written in Python in order to identify large time gaps between log entries, indicating that a function used a large amount of computation time between log entries. Since the log also contained the file and line number that spawned the entry, this allowed for identification of parts of the source code that needed to be benchmarked further.

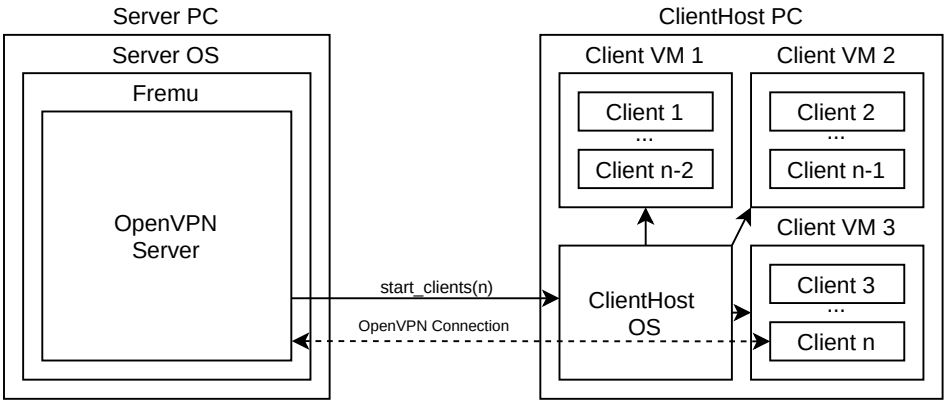

Figure 2.1: Measurement setup with Fremu

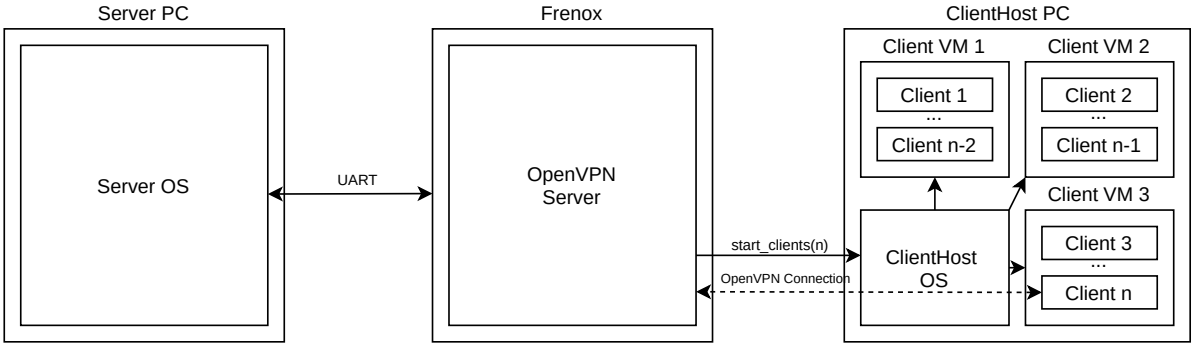

Figure 2.2: Measurement setup with Frenox

Listing 2.1: Generalized OpenVPN log file example snippet with verbosity set to level 9

```
... 15:04:42 2020 us=427794 ... /ssl_tls.c:3096): [message]
... 15:04:42 2020 us=431313 ... /ssl_tls.c:3108): [message]
... 15:04:42 2020 us=434322 ... /ssl_srv.c:3435): [message]
... 15:04:42 2020 us=437955 ... /ssl_srv.c:3084): [message]
... 15:04:44 2020 us=296745 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=301802 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=306539 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=311674 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=315423 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=319996 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=324590 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=329124 ... /ssl_srv.c:3136): [message]
... 15:04:44 2020 us=333357 ... /ssl_srv.c:3229): [message]
... 15:04:44 2020 us=338618 ... /ssl_srv.c:3280): [message]
```

These more in-depth, custom benchmarks were done by using the `clock_gettime` function, which stores the time at which the aforementioned function was called with near-nanosecond accuracy. By calling the same function again after the code to be benchmarked, and measuring the difference, the amount of time spent running the function could be measured. An example of the measurement code can be found in listing 2.2.

Listing 2.2: Benchmarking code used to measure code performance

```
#ifdef TL_BENCHMARK
    struct timespec t_start, t_end;
    clock_gettime(CLOCK_MONOTONIC, &t_start);
#endif

/* Code to be benchmarked here */

#ifdef TL_BENCHMARK
    clock_gettime(CLOCK_MONOTONIC, &t_end);
    long t_end_s = t_end.tv_sec - t_start.tv_sec;
    long t_end_ns = t_end.tv_nsec - t_start.tv_nsec;
    if (t_end_ns < 0) {
        t_end_ns += 1000000000;
        t_end_s--;
    }
    printf("--> [name]                        ------------------> = %2ld.%09ld s\n",
           t_end_s, t_start_ns - t_end_ns);
#endif
```

A snippet of output from using this method in parts of the source code that have been identified as potential bottlenecks can be found in listing 2.3. Since the total time of a function can only be measured after its subroutines have all finished, indentation was used to show which benchmarked parts are a subroutine of a different part.
Every indentation is equal to a function call, so in the example in listing 2.3, the `MPI GCD` function consists of the `MPI GCD Init + Copy`, `LSB + RSHift`, `Shift + Sub` and `LShift + Copy` subroutines.

Listing 2.3: Benchmark log file example snippet. Prints indented to the right above a line indicate that they are subfunctions of that function

```
--------------> Invert Jacobian                    --------------> =  0.000254900 s
-------------------------> MPI InvMod init         -------------------------> =  0.000021600 s
----------------------------> MPI GCD Init+Copy    ----------------------------> =  0.000093500 s
----------------------------> LSB + RShift         ----------------------------> =  0.000030900 s
----------------------------> Shift+Sub( 284 loops)----------------------------> =  0.011622200 s
----------------------------> LShift + Copy        ----------------------------> =  0.000088400 s
-------------------------> MPI GCD                -------------------------> =  0.013634900 s
-------------------------> MPI ModCpyLset         -------------------------> =  0.000311800 s
----------------------------> Nested loop 1        ----------------------------> =  0.000226100 s
----------------------------> 3x MPI sub          ----------------------------> =  0.000178900 s
----------------------------> Nested loop 2        ----------------------------> =  0.000031400 s
-------------------------->  284 core loops       -------------------------> =  0.064572400 s
-------------------------> Finalize               -------------------------> =  0.000070800 s
--------------> Normalize Jacobian                --------------> =  0.081754400 s
----------> Calculate comb                        ----------> =  0.671043100 s
------> ECP mul                                  ------> =  1.804750700 s
------> ECP zero check                           ------> =  0.000032300 s
------> MPI copy                                 ------> =  0.000054900 s
--> Calculate shared secret                      --> =  1.816833600 s
Key Exchange                                     =  1.817474800 s
```

By searching for routines that take a long time, but can no longer be split up into subroutines with one subroutine also taking a long time, the bottlenecking functions can be found. For example, if a function takes 2 seconds, but consists of 4000 loops that take 500 microseconds, that function would be a prime candidate for acceleration.

In order to establish the minimum amount of time that would still be realistic to accelerate without copy overhead becoming too large, the amount of time it would take to copy data to and from the accelerator is also established further on in this report in section 2.5.

In order to make sure no important functions were missed, the total time spent on benchmarked functions was compared to the total run time. If these are near equal, except for a few milliseconds, it is safe to assume no other important functions exist.

2.3. OpenVPN Performance Bottlenecks

After identifying the routines that were most likely to cause a bottleneck in performance, using the benchmarking method described in listing 2.2, acceleration of these routines was simulated. This was done by using Fremu to simulate the resulting performance in case a perfect accelerator would be connected to Frenox.
This functionality was achieved by writing function inputs from Fremu to shared memory as would also be done with the real accelerator. The communication protocol used to do this is explained in section 3.4.2.
The emulator would take this input from memory, pause Fremu execution, calculate the result using the original mbedTLS function, but running on the Server PC instead. The communication protocol will be described more in-depth in section 3.4.2
The emulator would then place the result in the expected shared memory location, and resume Fremu execution. Fremu would then read the result of the simulated accelerator from memory and copy it back to the appropriate variables in OpenVPN.
Performance-wise, from the viewpoint of Fremu, this means that the emulated platform sees the result of the accelerator back in memory after a single clock cycle. Of course this is not a realistic scenario, but it does show the maximum potential for acceleration that could be obtained.

The results of these simulations can be found in table 2.5. In this table, the functions with the largest impact on performance can be found, as well as the amount of time that could maximally be saved per client connection by accelerating it. In table table 2.4, the speedup with all accelerators simulated at the same time can also be found. This is of course not a realistic scenario, since it would be very difficult to create an accelerator core that could accelerate all of these functions while still being area-efficient, but it does show the potential gains of accelerating just a few functions.

	Accelerator disabled	Accelerator enabled	Speedup
Connection time(s)	2.106	0.115	18.31 x

Table 2.4: Average connection time of a single client for extrapolated 100MHz Frenox server, default and accelerated

Accelerated function	None	ecp mul comb	ecp pre-compute comb	ecp mul	mpi exp mod	mpi inv mod	mpi gcd	mpi mod mpi
Connection time (s)	2.106	1.896	1.632	1.392	0.891	1.946	2.061	2.055
Improvement (s)	0	0.210	0.474	0.714	1.215	0.160	0.045	0.051
Speedup (x)	1	1.111	1.290	1.513	2.364	1.082	1.034	1.025

Table 2.5: Connection time reduction per client per accelerated function, averaged across 3 clients

Some behavioral differences were also observed between the case of all clients connecting simultaneously, and clients connecting in staggered manner, with a short amount of time in between each connection request. When all clients connected simultaneously, the TLS routine of OpenVPN would first perform all key exchanges, and then all authentication sequences. With staggered connections, OpenVPN would perform a key exchange and authentication for one client, and then move on to the next.
The resulting performance difference was measured and can be found in table 2.6. The final impact on performance appeared to be negligible, so this effect was not monitored anymore after performing

these benchmarks. The average connection time is slightly lower than in table 2.4, but this can be explained by the fact that the results in table 2.6 were averaged across more clients, which was needed to properly investigate the effect of staggering. Averaging across more clients slightly reduces certain overhead effects.

Connection time (s)	Default performance
Staggered startup	2.067
Simultaneous startup	2.042

Table 2.6: Average connection time of a single client for extrapolated 100MHz Frenox server

Keep in mind these numbers assume perfect acceleration. That is because of the use of the emulator, the acceleration result will always be ready after a single clock cycle. This is caused by the fact that memory operations are blocking calls in the emulator, and the emulator calculates the ”accelerator” result during a memory write operation.

The total speedup in table 2.4 does not match with the combined improvements of accelerating each function separately. However, this can be explained by the fact that some functions are co-dependent; by accelerating some functions, other functions will be called less often, or even not at all, because they used to be called by a function that is now accelerated.

Furthermore, `mpi_inv_mod` will have less of an effect when averaged over more clients, since its most heavy calculation is only executed during its first connection. This means that by far the largest influence on performance comes from `ecp_mul` and `mpi_exp_mod`. This is somewhat to be expected, since ECP multiplication is the main operation used in the ECC part of the ECDHE key exchange. Furthermore large modular exponentiation is the most computationally expensive part of the RSA signature algorithm. This indicates that the ECDHE and RSA parts of the TLS ciphersuite are the most computationally intensive.

2.4. Verification

Since all of these results are simulated, some verification is in order to see if these simulations match performance on the real Frenox. Since it is impossible to compare accelerator performance, since that would require the accelerators to be already implemented on Frenox, the non-accelerated case is compared.

Linear extrapolation based on the clock speed of Femu is not an ideal method to estimate performance on the real-world system.

The results obtained in the previous section were calculated by taking the performance at 7.5 MHz, and assuming computation time scales linearly with clock speed. This means that a run time of 10 seconds at 7.5 MHz would be estimated to run $10 * (7.5/100) = 0.75$s.

Although the amount of instructions per second a processor can execute generally scales close to linearly with clock speed, this is not guaranteed when moving to a different platform where other aspects that can impact performance, such as cache size and memory speed, differ as well.

In fig. 2.3, performance measurements on both Frenox and Fremu can be compared. The measured data is a five-run average, with three connecting clients. The time represented is the average connection time of a single client, so the five-run average divided by three. The amount of clients was set to three, since this meant one client per VM, so per-VM performance has no impact on the measurements. It would technically be possible to connect more clients per VM, but this would cause connection attempts to time out in the 10 MHz scenario due to limited performance when attempting to serve all clients at once.

Performance is different on Frenox than expected from Fremu. Unfortunately, real-world performance is significantly lower than performance emulated on Fremu. At higher frequencies, this difference converges to about a factor 1.8, as can be seen in table 2.8. A first suspect might be that performance does not scale linearly with clock frequency, since Frenox runs at 50 MHz instead of the

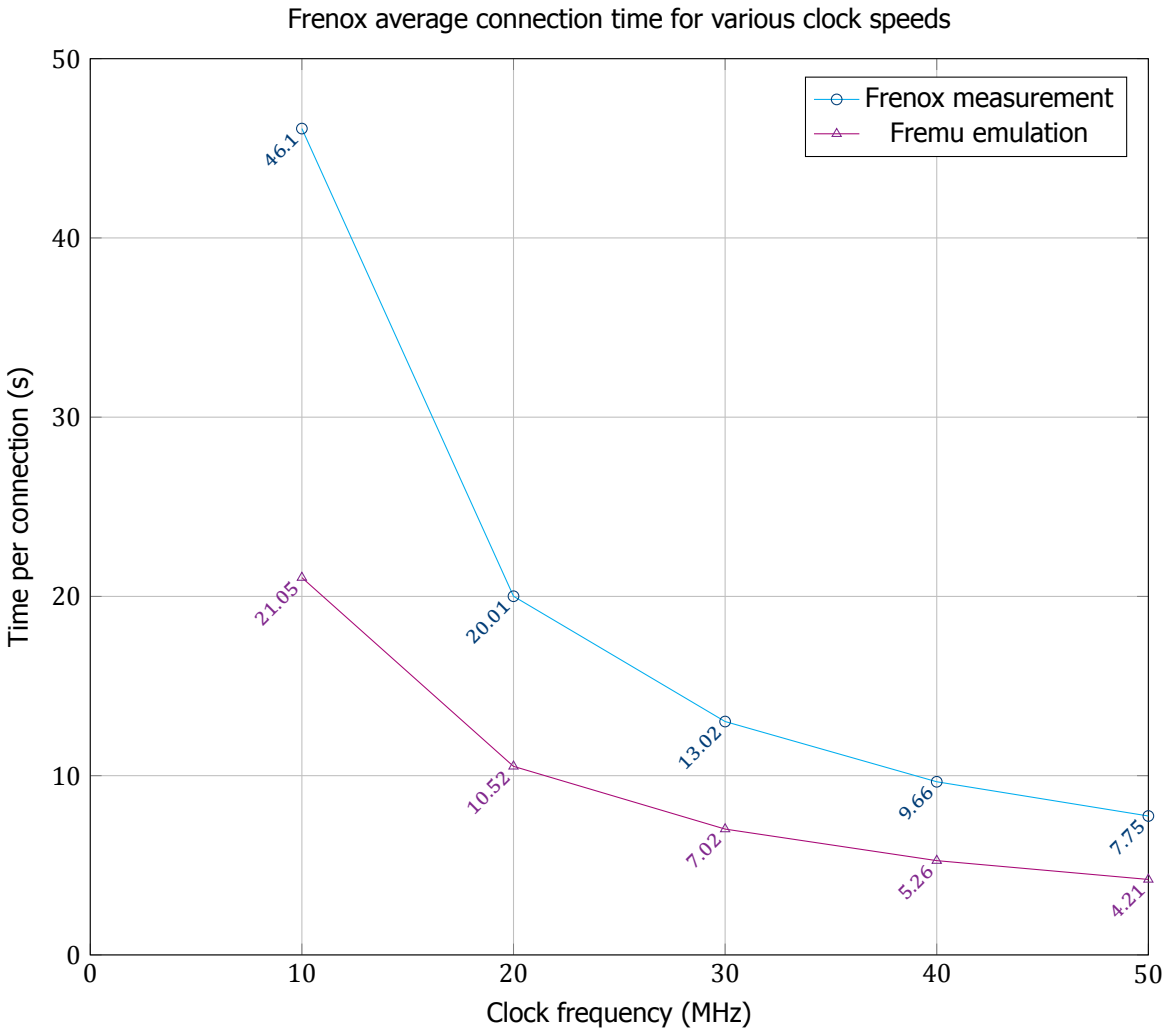

Figure 2.3: Amount of time needed to connect to a single client

7.5 MHz that Fremu runs at.

To verify this, Frenox performance was measured at 10, 20, 30, 40 and 50 MHz, after which the performance of the clock frequencies below 50 MHz was extrapolated to 50 MHz. For example, the run time of a single client at 10 MHz would be divided by 5, since that assumes linear scaling to 50 MHz.

When comparing the extrapolated connection times at 10, 20, 30 and 40, performance does scale near linearly, though not quite entirely, especially for lower frequencies like the 7.5 MHz used with Fremu. This leads to the conclusion that the scaling applied to the results from Fremu are somewhat inaccurate with regards to real-world performance. A suspected reason for this is that in Fremu, the emulator always behaves as if variables requested from memory are in cache, which is not a realistic real-world scenario that is independent of clock frequency.
This does, however, not mean that the discovered bottlenecks are invalid. In fact, they still use up the same percentage of time. This just means that real-world performance can be expected to benefit even more from acceleration than the simulated case, since the real-world measurement is actually slower than the emulated case.

Clock Frequency (MHz)	Frenox runtime (s)	50 MHz extrapolation (s)	Error (s)	Error (%)
10	46.10	9.22	1.47	15.9
20	20.01	8.00	0.25	3.1
30	13.02	7.81	0.06	0.8
40	9.66	7.72	0.03	0.4
50	7.75	7.75	0	0

Table 2.7: 50 MHz extrapolated Frenox runtime compared to actual runtime at 50 MHz

Clock Frequency (MHz)	Difference (slowdown)
10	2.19 x
20	1.90 x
30	1.85 x
40	1.84 x
50	1.84 x

Table 2.8: Factor of time extra needed to connect a Frenox client when compared to Fremu extrapolation

2.5. Copy benchmarks

A very fast accelerator is meaningless when the overhead of communicating with the accelerator is larger than the amount of execution time the accelerator would save.
In order to prevent this scenario from occurring, a series of copy benchmarks were performed.

In order to measure the performance impact of copying data to and from shared memory between Frenox and the accelerator, a benchmark program was written in C, which copies random 32-bit words to and from the shared memory area. The total time that was needed to write to the memory and then read the data back was measured. Since this amount of time is expected to be quite low for small amounts of data, the amount of time needed to create a timestamp was also taken into account, so that the measurement itself did not add to the measured time.

The benchmark program runs each test 6 times, ignores the first result and records the lowest copy time measured. The ignoring of the first result is done to remove memory assignment effects, since memory will already be assigned when running the server for multiple clients.
This is necessary since the Linux kernel does not immediately allocate memory when using the `malloc` function of C. Only when data is actually assigned for the first time, will the memory be allocated.

Taking the lowest result out of five runs is done to remove the effect of other programs taking up CPU time while the benchmark is running. This is because Frenox is still a single core CPU running a full Linux kernel, so it is quite possible that the scheduler runs a different program for a short moment while the benchmark is still running, especially for longer runs with larger amounts of data.

2.6. Measurements

The resulting copy performance measurements can be found in fig. 2.4.

While writing the aforementioned benchmarking program, a small discrepancy in measurement data was discovered when copying less than 32 words. When digging further into the kernel, it was found that the implementation of `memcpy` used in Linux for RISC-V has assembler optimizations that behave differently when copying less than 32 words, causing the write actions to no longer be word-aligned in memory. However, the accelerator expects the data to be word-aligned. Therefore, when copying less than 32 words, they are simply assigned manually in a loop. When copying larger amounts, `memcpy` is used. This is what causes the sudden fall in copy time at 32 words in the graphs.

Another option would be to support bit masking for memory access in the accelerator, since `memcpy` does supply a correct bit mask when copying data byte-wise. The resulting performance when using bit masking and always using `memcpy` can be found in fig. 2.5. A slight performance increase can

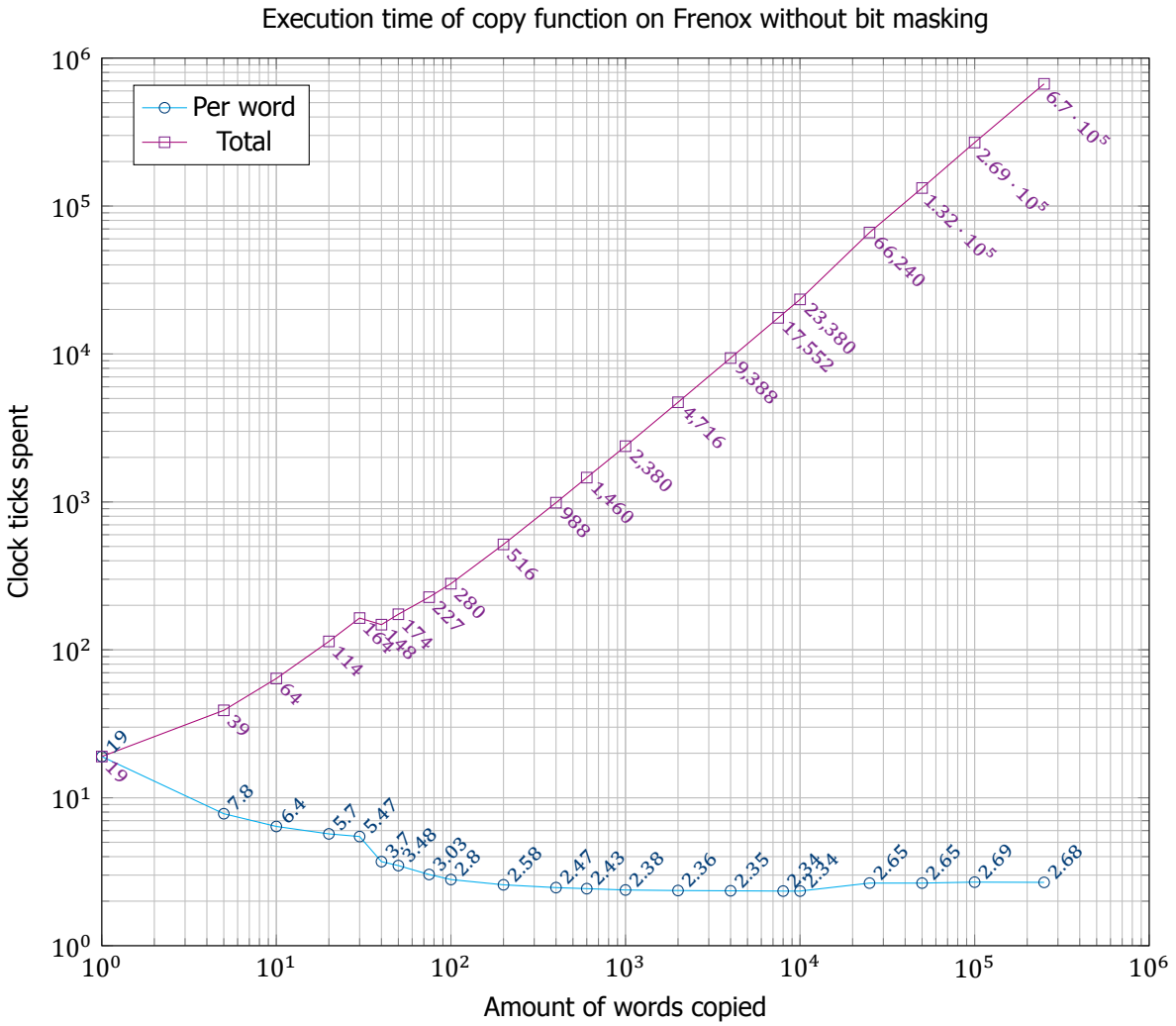

Figure 2.4: Amount of clock ticks Frenox needs to copy a word to or from memory, with manual copy if ≤32 words are copied to prevent memcpy byte-wise copy

be found when copying between 20 and 30 words. With less than 20 words, performance is actually slightly slower than copying manually, probably due to the overhead of calling the `memcpy` function instead of immediately copying the data. Since the performance impact of implementing bit masking is negligible, it was chosen to not implement it, and instead perform write operations of less than 32 words manually, word-by-word, in a loop.

For communication, the efficiency of `memmap`, the C function used to access the shared memory, needs to be evaluated for this use case. If it takes a significant amount of copy time, it might be worth it to also measure performance for Direct Memory Access (DMA)-based communication. DMA could potentially be faster since it does not use up CPU cycles, but it also takes more time and effort to implement and could cause cache coherency issues.
Furthermore, a `memmap`-based interface is already necessary for setting certain control flags (later introduced in section 3.4.2), so implementing data transfers using `memmap` comes at the cost of very little extra effort and time.
In the case of accelerating `ecp_mul`, which uses 384-bit words:
A 384-bit number consists of 12 words. When looking at fig. 2.4, the expected amount of clock cycles spent copying a word of sizes near 10 words costs about 6.4 clock cycles per word. For four numbers, this becomes $4 * 12 * 6.4 \approx 308$ clock cycles. When reading the result, only the resulting X-, Y- and Z-coordinates need to be loaded, so that would take $3 * 12 * 6.4 \approx 231$ cycles, for a total copy cost of

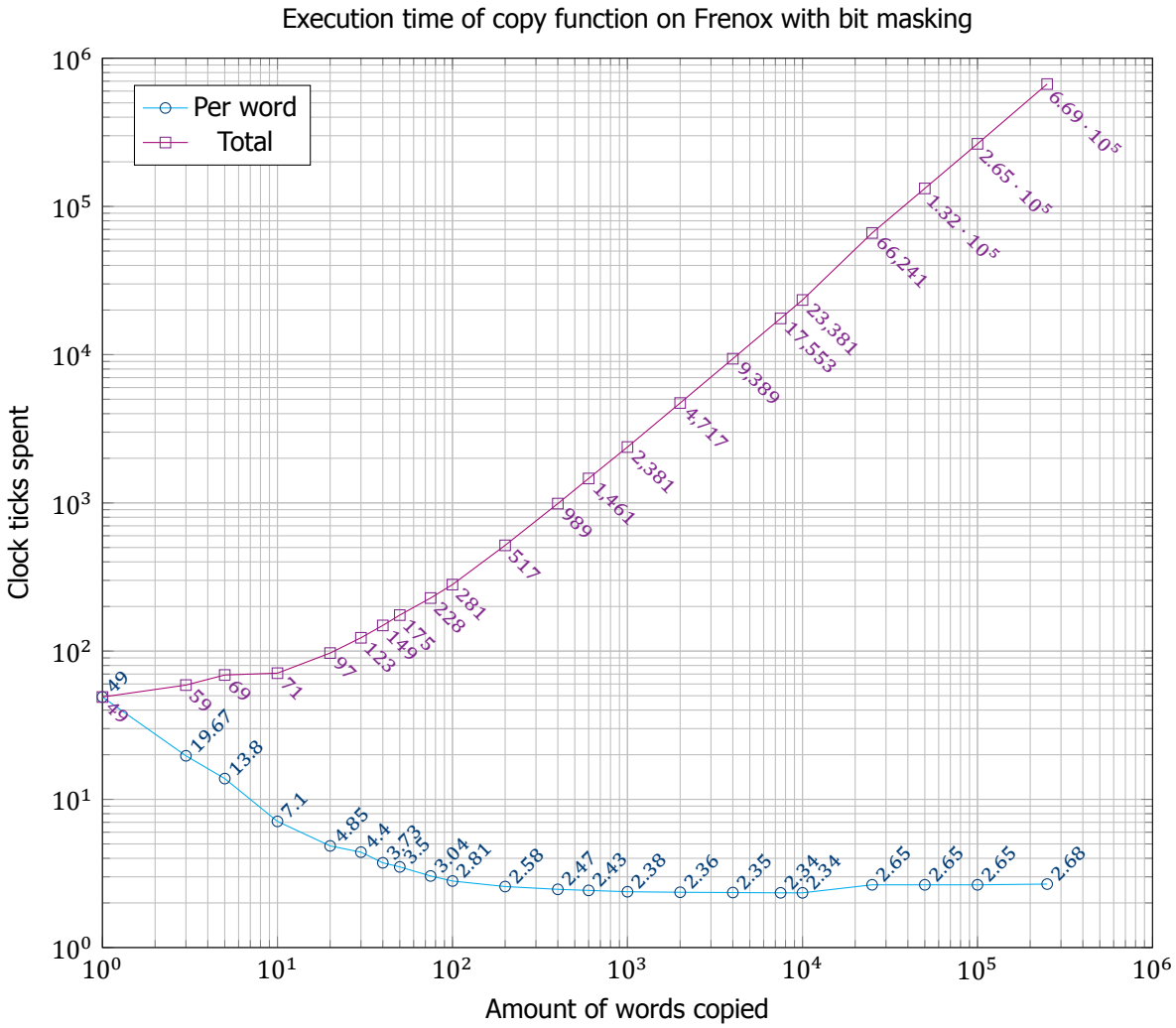

Figure 2.5: Amount of clock ticks Frenox needs to copy a word to or from memory using only memcpy

$308 + 231 = 539$ cycles per accelerator call.
When running at 100 MHz, the amount of time spent on this copy operation becomes 5.39 us, which is a negligible amount of time compared to the average time spent on an operation for OpenVPN, as was shown in section 2.3. In the case of ECDHE, the `ecp_mul` function is called twice per connecting client, so the total time per client would be 10.78 us, which is still negligible compared to the total run time of more than 700 ms.
In the case of `mpi_exp_mod` acceleration, used for RSA:
In OpenVPN-NL, RSA uses up to 2048-bit numbers, which comes down to 64-word numbers, in which case copying data costs about 3.25 cycles per word. Writing two numbers for exponentiation input and reading one back would cost approximately $3 * 64 * 3.25 \approx 624$ cycles. Since the RSA exponentiation function is ran twice per client, this comes down to about 12.48 us of copy time, which is still negligible compared to the 1.21 s of time spent on `mpi_exp_mod`.
Therefore, it was chosen not to use DMA-based communication, but to use the much simpler to implement `memmap`-based communication.

2.7. Selected function for acceleration

As can be seen from table 2.5, The largest gains are possible during the `mpi_exp_mod` computation of the RSA signature process. A second contender is the `ecp_mul` function, which is executed during

the ECDHE key exchange. However, since RSA acceleration has already been done at Technolution, the company at which this thesis is written, and considering the fact that in the future, ECDSA can be used to replace RSA, which also uses `ecp_mul`, in the end the choice was made to create an ecp_mul accelerator, or an Elliptic Curve Point (ECP) multiplier.

3

Design

3.1. Design goals

The design of the ECP multiplication accelerator was made with the following design goals in mind:

- The design has to fit next to the Frenox core on an Altera Cyclone V-5CGXFC5C6 chip. This comes down to a maximum resource utilization of around 20,000 ALMs.
- The accelerator should be resistant to side-channel attacks, especially time-based ones.
- The accelerator should be fast enough to provide a noticeable speedup when compared to the calculation time in C on the Frenox core.
- The accelerator should be usable with most server hardware and software platforms, and should not depend on CPU ISA or OS-specific properties.
- It should be feasible to implement and test the accelerator design within a few months.

The largest challenge of these constraints is the resource utilization. Multiple complex mathematical operations on large integers are needed to compute an ECP multiplication. An ALM on a Cyclone V-chip contains 4 registers and 2 logic blocks, so care has to be take in order to make the design fit and be routable.
The final constraint is a reminder that the goal of this work is not to design the most efficient or fast accelerator core, but one that shows a decent improvement in order to demonstrate the effect of an accelerator core on OpenVPN control path performance.

3.2. FPGA Acceleration Core

As mentioned in section 2.7, it was decided to accelerate the `ecp_mul` function. This function performs Elliptic Curve Point (ECP) multiplication, by adding a point P to itself a scalar n amount of times. In order to speed up computation, an ECP doubling function also exists.

In order to be able to perform ECP multiplication using an accelerator core, three layers of computations must be implemented in hardware. They are:

1. Large-number integer modular maths: addition, subtraction, multiplication and halving.
2. ECP adding and doubling algorithms.
3. ECP multiplication algorithm.

Each higher-numbered layer can be computed using the computations of the layer below it. For instance, ECP doubling requires modular addition, subtraction and multiplication. The final top-level architecture can be found in fig. 3.1. Note that the modular multiplier is shared between the ECP adder and doubler. This is needed in order to fit the design. As will later on be shown in chapter 4, the

modular multiplier is the largest component in the entire accelerator, and having two of them would make the design impossible to fit on the used FPGA.
Performance impact of this shared architecture is present, but not very high. The measured simulation time for a single ECP Multiplication at 50 MHz went from $\pm$13 ms to $\pm$15 ms, which is acceptable considering the space savings gained.

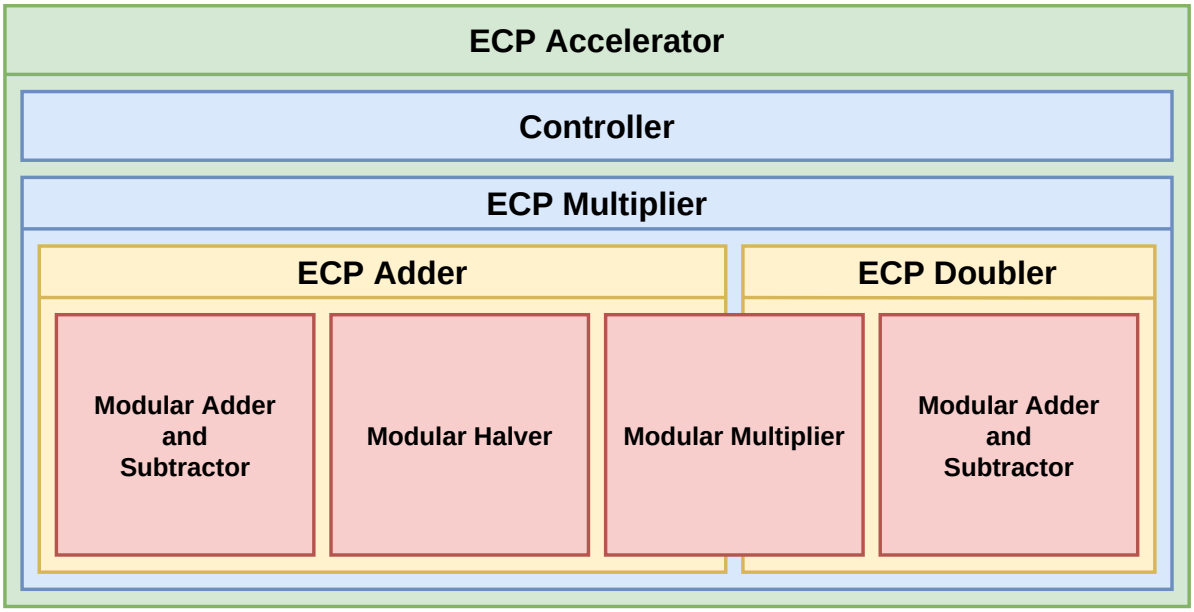

Figure 3.1: Acceleration core toplevel

An elliptic curve is a two-dimensional curve, so points on the curve are of the affine (x, y) form. However, in order to simplify the computations, the used algorithm for ECP multiplication uses *projective coordinates*. These projective coordinates also contain a z-coordinate, and are thus of the form (x, y, z).
Converting from affine to projective coordinates is very simple: $(x, y) \rightarrow (x, y, 1)$. However, converting back is not quite so simple, and requires in its computations a modular inverse to be found.

In order to save resources and development time, the output result of the accelerator will be in projective coordinates ($z \neq 1$). Therefore, the result will have to be normalized to affine coordinates in order to keep mbedTLS support. This can be done in software using the built-in `ecp_normalize_jac()` function of mbedTLS, at the cost of some performance.

3.2.1. Large-number modular mathematics

For the large-number mathematics, the design makes use of modern FPGA features, such as the fast carry chain adder and the built-in DSP blocks, which contain a multiply-accumulate (MACC) block.

Multiplication

The DSP blocks inside the Cyclone V FPGA have a maximum input size of 18x18 bits. Since our input is 384-bit, a multiplication algorithm is needed where the input is divided up into smaller parts.
Since 384 is nicely divisible by 16, the decision was made to make use of 24 DSP blocks, each with a 16x16 multiplication. The input variables were then split up into arrays of 16-bit words, and the Comba method, also known as schoolbook multiplication, was used to compute the final result. The computation of each partial product can be parallellized in such a way that every DSP is used every cycle, greatly reducing computation speed. An example of this can be seen in fig. 3.2

The resulting design can be found in algorithm 1.
First, the partial products are computed, and when a partial product is ready, it is accumulated into the final result. In order to cycle through the inputs needed for the partial products, a sliding input window was used instead of an input multiplexer in order to save on used area.

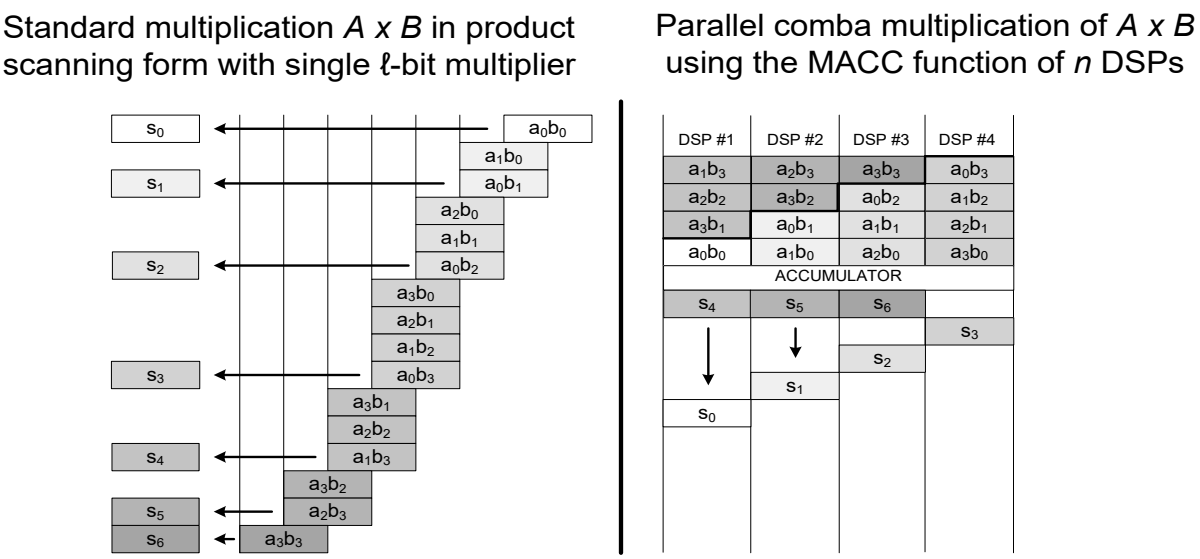

Figure 3.2: Parallel Comba multiplication using multiple DSPs. Image taken from [8]

This sliding window takes the two input integers, and rotates them so that each DSP can use a set part (for instance bits 16 to 31) of the input variables without needing a multiplexer to select the correct part of the input variables.

The DSP parallelization can be seen in a simulation of the design, found in fig. 3.3. On the horizontal axis of that image is time, while the vertical axis represents the DSP number, with DSP #1 at the top. The blue line clearly divides the two phases of partial multiplication.

However, this design outputs a 768-bit result, while modular arithmetics dictate that the end result should still be 384 bits in size. In other words, a modulo operation needs to be applied to the multiplication result.

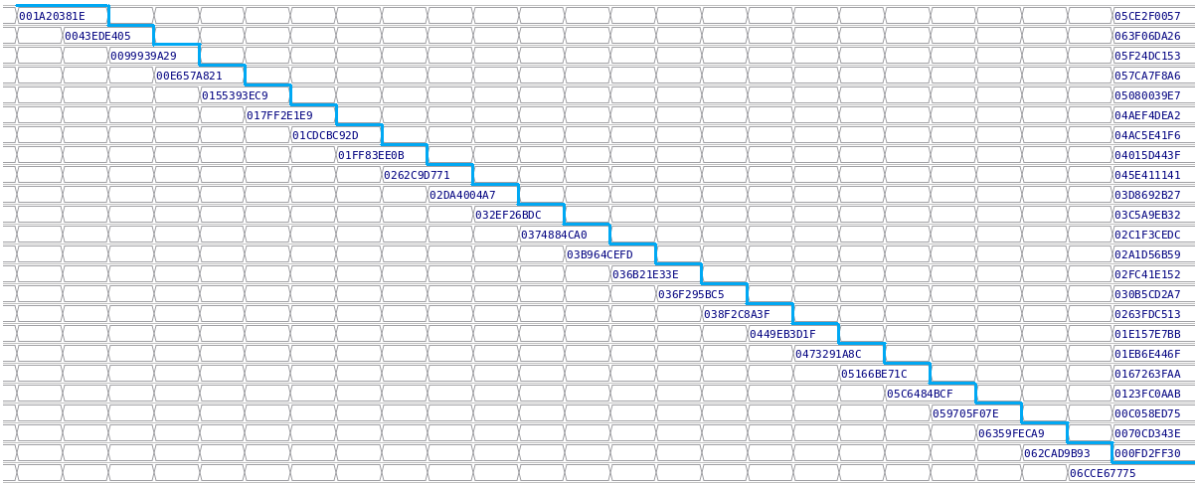

Figure 3.3: Large number Comba multiplication. The blue line represents the barrier between the two phases

Fortunately, the NIST P384 curve has a specific algorithm that can be used to quickly determine the value of a 768-bit number modulo the $p384$ prime number, which is a 384-bit prime number used by NIST to define the P-384 curve. This algorithm was provided by the NIST in [19], and the used implementation of that algorithm can be found in algorithm 2.

The result of this modulo calculator is the final result of modular multiplication.

Algorithm 1: `mul(a, b)`: 384-bit non-modular number multiplication using 24 16x16 DSPs
$r = a * b$, P is array containing 24 partial products

```
num_dsp ← 24
in_a_window_1 ← a
in_a_window_2 ← a
in_b_window_1 ← b
in_b_window_2 ← b
for i ← 0 to num_dsp do
    P(i) ← 0
for i ← 0 to num_dsp do
    for j ← 0 to num_dsp do
        if j < i then
            // First half of partial multiplications
            P(i) ← P(i) + in_a_window_1 * in_b_window_1
        else if is_done(i) = False then
            // First half of accumulation
            r ← (r_carry[15...0] || r[767...0])
            r_carry ← (0x0000 || r_carry[31...16]) + P(i)
            P(i) ← 0
            is_done(i) ← True
        if i + 1 + (num_dsp − j) < num_dsp then
            // Second half of partial multiplications
            P(i) ← P(i) + in_a_window_2 * in_b_window_2
        // Rotate input
        in_a_window_1 ← in_a_window_1 ror 16
        in_b_window_1 ← in_a_window_1 rol 16
        if j > 1 then
            in_a_window_2 ← in_a_window_2 rol 16
            in_b_window_2 ← in_a_window_2 ror 16
for i ← 0 to num_dsp do
    // Second half of accumulation
    r ← (r_carry[15...0] || r[767...0])
    r_carry ← (0x0000 || r_carry[31...16]) + P(i)
return R
```

Addition and subtraction

Addition and subtraction is done by using the fast carry chain adder built into the FPGA.
The final result should be checked to see if it has not become larger than the modulo. If it has become larger, the modulo needs to be subtracted from it once.
However, since we want execution to be constant in time, this modulo subtraction is always performed and stored to a different register. Then, the check is performed. If the result was smaller than the modulo, it will be returned, if not, the subtracted result will be returned. The resulting design can be found in algorithm 3

Halving

Since division by two can be done by a simple right-shift of the binary representation of an even number, this is exactly what is done. If the number is even, the bit representation is shifted right by one bit. If the number is uneven, we can simply add the $p384$ prime to it, since that prime is an uneven number. However, since we added the exact number that we modulate by, the represented number is still the same ($A + p384 \bmod p384 = A$). Since we want to have constant-time execution, this addition is always performed, and the either the increased or actual input will be shifted for the final result.
The resulting implementation can be found in algorithm 4

Algorithm 2: `mul_mod(a)` NIST P384 Modulo for 768-bit input:
$r = a \bmod p_{384}$, $||$ is the concatenation operator.
Input number a is divided up into 24 32-bit pieces, represented by a_n in this algorithm

$$
\begin{array}{lcl}
t & \leftarrow & (\ a_{11} \| a_{10} \| a_9 \| a_8 \| a_7 \| a_6 \| a_5 \| a_4 \| a_3 \| a_2 \| a_1 \| a_0\ ) \\
s_1 & \leftarrow & (\ 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| 0 \| 0 \| 0 \| 0\ ) \\
s_2 & \leftarrow & (\ a_{23} \| a_{22} \| a_{21} \| a_{20} \| a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12}\ ) \\
s_3 & \leftarrow & (\ a_{20} \| a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \| a_{23} \| a_{22} \| a_{21}\ ) \\
s_4 & \leftarrow & (\ a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \| a_{20} \| 0 \| a_{23} \| 0\ ) \\
s_5 & \leftarrow & (\ 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| a_{20} \| 0 \| 0 \| 0 \| 0\ ) \\
s_6 & \leftarrow & (\ 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| 0 \| 0 \| a_{20}\ ) \\
d_1 & \leftarrow & (\ a_{22} \| a_{21} \| a_{20} \| a_{19} \| a_{18} \| a_{17} \| a_{16} \| a_{15} \| a_{14} \| a_{13} \| a_{12} \| a_{23}\ ) \\
d_2 & \leftarrow & (\ 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{22} \| a_{21} \| a_{20} \| 0\ ) \\
d_3 & \leftarrow & (\ 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| 0 \| a_{23} \| a_{23} \| 0 \| 0 \| 0\ ) \\
d_1 & \leftarrow & p_{384} - d_1 \\
r & \leftarrow & t + 2 * s_1 + s_2 + s_3 + s_4 + s_5 + s_6 + d_1 - d_2 - d_3
\end{array}
$$

for $i \leftarrow 0$ **to** 3 **do**
  if $r < p_{384}$ **then**
    $result \leftarrow r$
  $r \leftarrow r - p_{384}$
return $result$

Algorithm 3: `mod_add(a, b, sub)`: Modular addition and subtraction
$r = a + b$ or $r = a - b$ if sub flag is set to $True$

if $sub = True$ **then**
  $c \leftarrow a - b$
else
  $c \leftarrow a + b$
$d \leftarrow c - p_{384}$
if $c > p_{384}$ **then**
  $r \leftarrow d$
else
  $r \leftarrow c$
return r

3.2.2. Elliptic Curve Point addition and doubling

ECP addition and doubling is done according to the ECP addition algorithm recommended by NIST in [19].
These algorithms make use of the basic mathematical operations defined in section 3.2.1.
For some steps in the algorithm, a temporary variable is needed in order to be able to perform them using only the available elementary operations.
These temporary values are marked as t_t in the implementation algorithms. The original step that was replaced with some extra temporary steps can be found between brackets in the final step using the t_t variable

For clarity reasons, the usage of the basic mathematical algorithms `mul_mod`, `add_mod` and `halve_mod` have been replaced by the $*$, $+/-$ and $/2$ operators respectively in these algorithms.

The final results can be found in algorithm 6 and algorithm 5.

3.2.3. Elliptic Curve Point multiplication

For the ECP multiplication, an algorithm called the Montgomery Ladder[20] is used. This algorithm makes use of the ECP doubling and addition operations defined in section 3.2.2. Since the amount of loops is constant, and each function is always called simultaneously, this algorithm is resistant to time-based side-channel attacks, and sufficiently resistant to power-based attacks.
The algorithm can be found in algorithm 7.

Algorithm 4: `halve_mod(a)`: Modular halving
$r = a/2$

$b \leftarrow a + p_{384}$
if $a \ mod \ 2 = 1$ **then**
  return $b >> 1$
else
  return $a >> 1$

3.3. Optimizations

For the ECP addition and doubling, intermediate results are stored to and loaded from blockram components to save additional register and LUT usage. In the case of addition, this saves seven 384-bit registers, while in the case of doubling, it saves five 384-bit registers. Performance impact of this load and store is minimal; in simulation ony 3 ms difference was measured per ECP multplication.

In the Montgomery Ladder algorithm, described in algorithm 7, results of ECP addition and ECP doubling are stored in intermediate values $R0$ and $R1$. However, since the implemented ECP adder and doubler already have registers in which their results are stored, their output can be directly used as input for the next cycle.

This results in the algorithm which can be found in algorithm 8, and saves 6 registers of 384 bits in size (two projected coordinates). As can be seen, this algorithm still retains the properties of being constant in execution time (the amount of loops has not changed) and being constant in power, since for every loop, both ecp_add and ecp_double are running.
The only compromise is that the doubler needs a 2-input mux to be able to select between doubler or adder output, however the size of the 2-input mux far outweighs the size of size 384-bit registers.

3.4. Communication

A communication method needs to be established in order for OpenVPN to be able to actually use the accelerator.
This section describes the memory architecture and communication protocol used to exchange data between the accelerator core on one side, and OpenVPN running on the Frenox core on the other side.

3.4.1. FPGA side of communication

Using an in-house Register Description Language (RDL) developed by Technolution, a shared memory area can be added to Frenox. Once added in the RDL file, the shared memory will show as a Userspace I/O device in Linux. It can be accessed through `/dev/uio`.
Since the UIO device used for the accelerator was the first UIO device to be added, it can be accessed at `/dev/uio0`. Using native C function `memmap`, this shared memory area can be mapped and made accessible to the C source code of OpenVPN.

This shared memory are consists of two areas: the data area, used for writing input and output data to and from the accelerator, and the control area, used for setting and reading flags.

Control memory contains three registers:

- start: Control register used to tell the accelerator that input is waiting for it in memory
- done: Status register the accelerator can set to tell the Frenox core that its output is waiting for it in memory
- debug: Status register used to check the current state of some accelerator signals in case an error occurs.

Algorithm 5: `ecp_double(S)`: NIST ECP Doubling Algorithm using only elementary operations:
Return $2S$ where $S = (S_x, S_y, S_z)$. t_t represents a temporary value.

$t_1 \leftarrow S_x$
$t_2 \leftarrow S_y$
$t_3 \leftarrow S_z$
if $t_3 = 0$ **then**
  return (1, 1, 0)
$t_4 \leftarrow t_3 * t_3$
$t_5 \leftarrow t_1 - t_4$
$t_4 \leftarrow t_1 + t_4$
$t_5 \leftarrow t_4 * t_5$
$t_t \leftarrow t_5 + t_5$
$t_4 \leftarrow t_t + t_5$ $(t_4 \leftarrow 3 * t_5)$
$t_3 \leftarrow t_3 * t_2$
$t_3 \leftarrow t_3 + t_3$
$t_2 \leftarrow t_2 * t_2$
$t_5 \leftarrow t_1 * t_2$
$t_t \leftarrow t_5 + t_5$
$t_4 \leftarrow t_t + t_t$ $(t_4 \leftarrow 4 * t_5)$
$t_1 \leftarrow t_4 * t_4$
$t_t \leftarrow t_5 + t_5$
$t_1 \leftarrow t_1 - t_t$ $(t_1 \leftarrow t_1 - 2 * t_5)$
$t_2 \leftarrow t_2 * t_2$
$t_t \leftarrow t_2 + t_2$
$t_t \leftarrow t_t + t_t$
$t_2 \leftarrow t_t + t_t$ $(t_2 \leftarrow 8 * t_2)$
$t_5 \leftarrow t_5 - t_1$
$t_5 \leftarrow t_4 * t_5$
$t_2 \leftarrow t_5 - t_2$
return (t_1, t_2, t_3)

Each register is one word in size to allow for easy reading and writing since Frenox uses the AXI4Lite[1] bus internally and AXI4L memory access is on a per-word basis.

As can be seen in fig. 3.4, control memory size is 3 words and starts at 0x00001000. Data memory is 128 32-bit words in size and starts at 0x00000000. The accelerator expects the input coordinates to be written to the memory in one continuous area, 48 words (four times a 384-bit number) in size. The output data will be stored in the next 36 words after the input words (three 384-bit numbers).

3.4.2. OpenVPN side of communication

In order to facilitate communication from OpenVPN with the FPGA accelerator core, a C library was written. This library simplifies addressing the correct memory regions with macros and contains functions to write the mbedTLS MPIs that OpenVPN uses to memory correctly, as well as reading them back. For the Fremu performance testing mentioned in chapter 2, additional functions to write more advanced classes to the shared memory region were also added, but these are unused in the final result.

Data can be copied in multiple granularities:

Bytes, words, MPIs, Elliptic Curve Points, and ECP groups. Each larger object is built out of multiple smaller objects. For example, an EC Point consists of three MPIs. An ECP group contains the full context of the elliptic curve algorithm that is being calculated. The only exception is bytes, which are first converted to words. This is because Frenox only writes aligned on a per-word basis, and writing only words means we do not have to use bit masking on the accelerator side to re-align byte-sized writes.

[1] https://www.xilinx.com/products/intellectual-property/axi_lite_ipif.html

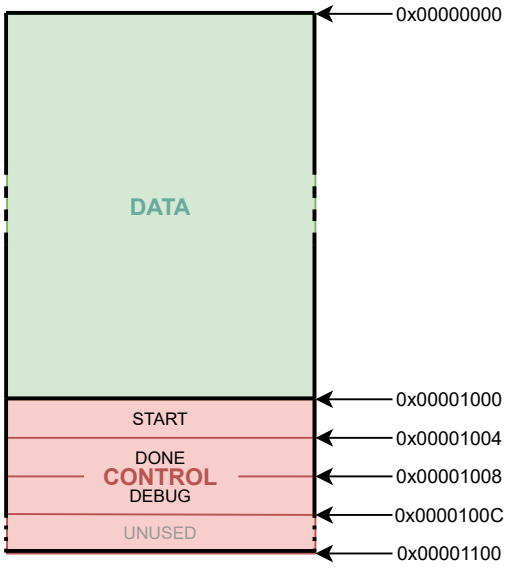

Figure 3.4: Accelerator shared memory layout (Bytes)

Copying is done manually for sizes smaller than 32 words, because the aforementioned optimized `memcpy` implementation in chapter 2 might use byte-sized writes for smaller sized copy actions. From 33 words and up `memcpy` is used, since it is significantly faster due to its architecture-specific optimizations.

The library can also be used to enable or disable acceleration of certain functions. It can also be used to enable or disable benchmarking. Furthermore, extra memory checks can be enabled at the cost of a little performance due to the extra operations needed for checking. Most of these functions were used during debugging, and are left out of the final version. The way to enable or disable the aforementioned features is by setting the correct flags during software compilation.

Locations of variables in data memory are automatically managed in the write and read functions, which increment the offset by the amount of words moved after a write or read action. This way, offset errors are not possible, as long as reading and writing of variables happens in the same order and the offset counter gets properly reset after each acceleration cycle.

The accelerator library also contains a function to run the accelerator after copying is done. It does the following:
- Flip the `start` flag to 1 and then back to 0 to send a start pulse
- Wait for the `done` flag to become 1
- Return so the read process can begin

The waiting for the `done` flag to become 1 is done via polling. When simulating the accelerator design in a VHDL simulator, the computational time for the entire accelerator was quickly found to be in the millisecond range, so the performance penalty of polling for a flag compared to the total run time of the accelerator was found to be negligible.

3.5. Validation

For validation of the accelerator output, the NIST provides a set of test values in its document containing the mathematical algorithms [19]. These test variables were used as a baseline to verify correct operation.
First, a Python prototype of the algorithm that was to be implemented in hardware was written in order

to verify correct operation of the algorithm and architecture.
After correct functionality was verified, the VHDL hardware design of each subcomponent of the ECP multiplier, and finally the ECP multiplier itself, was simulated using a VHDL simulator.
This was done using a Python program called `cocotb` [2], which makes it possible to send simulated AXI4L inputs and read AXI4L outputs to and from Python. The NIST-provided input values were sent from cocotb, and the end result was read from the simulated accelerator AXI4L output, and then compared to the expected result.
After all output variables were verified using the predetermined test variables, a random test was also added, which generated random keys using Python library `pycryptodome`.
From this key, a coordinate on the NIST P384 curve was extracted, and multiplied by a random scalar using the ECP multiplication function built into the `pycryptodome` library. The same coordinate and scalar were then sent to the accelerator, and the end result was compared. During a weekend, across a total of more than 400 VHDL simulation runs in 72 hours, no errors were found.
Finally, OpenVPN was modified to send its ECP multiplication input to the accelerator core, but also still calculate the result in software. There results were then compared in order to verify that the results were the same ones that OpenVPN-NL was expecting.

[2] https://github.com/cocotb/cocotb

Algorithm 6: `ecp_add(S, T)`: NIST ECP Addition Algorithm using only elementary operations:
Return $S + T$ where $S = (S_x, S_y, S_z)$ and $T = (T_x, T_y, T_z)$. t_t represents a temporary value.

$t_1 \leftarrow S_x$
$t_2 \leftarrow S_y$
$t_3 \leftarrow S_z$
$t_4 \leftarrow T_x$
$t_5 \leftarrow T_y$
if $T_z \neq 1$ **then**
  $t_6 \leftarrow T_z$
  $t_7 \leftarrow t_6 * t_6$
  $t_1 \leftarrow t_1 * t_7$
  $t_7 \leftarrow t_6 * t_7$
  $t_2 \leftarrow t_2 * t_7$
$t_7 \leftarrow t_3 * t_3$
$t_4 \leftarrow t_4 * t_7$
$t_7 \leftarrow t_5 * t_7$
$t_5 \leftarrow t_1 - t_4$
$t_5 \leftarrow t_2 - t_5$
if $t_4 = 0$ **then**
  if $t_5 = 0$ **then**
    return (0, 0, 0)
  else
    return (1, 1, 0)
$t_t \leftarrow t_1 + t_1$
$t_1 \leftarrow t_t - t_4$ $(t_1 \leftarrow 2 * t_1 - t_4)$
$t_t \leftarrow t_2 + t_2$
$t_2 \leftarrow t_t - t_5$ $(t_2 \leftarrow 2 * t_2 - t_5)$
if $T_z \neq 1$ **then**
  $t_3 \leftarrow t_3 * t_6$
$t_3 \leftarrow t_3 * t_4$
$t_7 \leftarrow t_4 * t_4$
$t_4 \leftarrow t_4 * t_7$
$t_7 \leftarrow t_1 * t_7$
$t_1 \leftarrow t_5 * t_5$
$t_1 \leftarrow t_1 - t_7$
$t_t \leftarrow t_2 + t_2$
$t_7 \leftarrow t_7 - t_t$ $(t_7 \leftarrow 2 * t_1)$
$t_5 \leftarrow t_5 * t_7$
$t_4 \leftarrow t_2 * t_4$
$t_2 \leftarrow t_5 - t_4$
$t_2 \leftarrow t_2/2$
return (t_1, t_2, t_3)

Algorithm 7: `ecp_mul(d, P)`: Montgomery Ladder Elliptic Point Multiplication algorithm

$R_0 \leftarrow 0$
$R_1 \leftarrow P$
for $i \leftarrow m$ **to** 0 **do**
  if $d_i = 0$ **then**
    $R_0 \leftarrow$ ecp_double(R_0)
    $R_1 \leftarrow$ ecp_add(R_0, R_1)
  else
    $R_0 \leftarrow$ ecp_add(R_0, R_1)
    $R_1 \leftarrow$ ecp_double(R_1)
return R_0

Algorithm 8: `ecp_mul(d, P)`: Optimized Montgomery Ladder Elliptic Point Multiplication algorithm

```
add_out ← ecp_add(0, P)
if d_i = 0 then
    double_out ← ecp_double(0)
else
    double_out ← ecp_double(P)
for i ← m to 0 do
    add_out ← ecp_add(add_out, double_out)
    if d_i = 0 then
        double_out ← ecp_double(add_out)
    else
        double_out ← ecp_double(double_out)
return double_out
```

4

Results

With the design fully tested and simulated, it was synthesized and programmed onto the FPGA. This section describes the obtained results, both from a resource usage and a performance standpoint.

4.1. Resource usage

The FPGA resources used by each component can be found in table 4.1. Furthermore, as expected, 24 DSP Blocks were used for the multiplier.
In this table, the "Own" values are used for components that have other components from this table as subcomponents in their design. The "Own" part shows the added resources when removing the resources used by their subcomponents in this table.
As can be seen, the largest component is the modular multiplier. This is the main reason why the multiplier was shared between both the ECP doubler and adder; the design would not fit with two multipliers. As mentioned before in chapter 3, though, performance impact of this shared multiplier was only minimal, so it was worth the trade-off. The second largest are the ECP adder and the multiplier itself, with the ECP doubler following closely.

	ALUTs	Registers	ALMs	Own ALUTs	Own Registers	Own ALMs
Full design	17865	22219	13648.6			
Control and data	1690	1642	1141.4			
ECP Multiplier	16165	50577	12506.7	2755	3896	2806.9
ECP Adder	4938	6279	3772.4	3269	3569	2539.6
ECP Doubler	2959	4730	2025.0	1681	3197	1299.1
Mod Multiplier	5513	5670	3897.7			
Mod Adder	1276	1551	729.4			
Mod Halver	393	1159	503.3			

Table 4.1: Resource usage on FPGA during synthesis and place & route

The resulting total resource usage was 85% when including Frenox, as can be seen in table 4.2. However, due to size limitations, the maximum frequency of 50 MHz could no longer be attained for both components. As a result, lowering the frequency to 40 MHz was needed. However, when synthesizing the accelerator in a standalone scenario, the 50 MHz frequency can be reached, as can be seen in table 4.3. Therefore, with a larger FPGA, this problem can be resolved.

	Available	With Frenox	Standalone
Logic utilization (ALMs)	29080	24,815	13,983
Logic utilization (%)	100	85	48

Table 4.2: Resource usage compared to available ALMs

	With Frenox	Standalone
Frequency (MHz)	42.28	50.19

Table 4.3: Maximum achievable frequency

4.2. Benchmarks

With the accelerator functional, and communication between OpenVPN and the accelerator functional, benchmarking the performance is the next logical step. The presented benchmark results were obtained by running both the Frenox core and accelerator at 40 MHz. Six clients started a connection at the same time, and their average connection time was measured across five runs.

The first tested scenario is the current one; using ECDHE-RSA for key exchange and authentication respectively. This is the scenario that would occur when using the accelerator core with the current allowed cryptography settings in OpenVPN-NL. The results can be found in table 4.4.

In the ECDHE-RSA scenario, a slight improvement of 1.6x, or a reduction of about 4.5 seconds per client can be measured, but as mentioned earlier in section chapter 2, RSA has the largest performance impact, and this has not been accelerated.

Therefore, the end result is a noticeable improvement, albeit not a very large one. If we would want to be able to reconnect all our clients within five minutes, the maximum amount of clients in the scenario has risen from $\pm$30 clients to $\pm$50 clients.

Extrapolated to 100 MHz, this would be $\pm$80 clients in the default scenario and $\pm$130 clients in the accelerated case.

	Accelerator disabled	Accelerator enabled	Speedup
Connection time (s)	9.27	5.75	1.61 x

Table 4.4: Average connection time of a single client for 40MHz Frenox server, default and accelerated, using ECDHE-RSA

However, when using ECDSA instead of RSA for the authentication step, the results are much more impressive. A speedup of more than 7 x can be achieved compared to the default performance. Even compared to the default ECDHE-RSA connection time of 9.27s, speedup is still more than 4.5 x. At 40 MHz, the amount allowed clients in order to allow for a full reconnect under five minutes is already $\pm$145, and when extrapolated to 100 MHz, this becomes a total of $\pm$350 clients.

	Accelerator disabled	Accelerator enabled	Speedup
Connection time (s)	14.30	2.03	7.04 x

Table 4.5: Average connection time of a single client for 40MHz Frenox server, default and accelerated, using ECDHE-ECDSA

4.3. Accelerator runtime

As can be seen in table 4.6, the total time it takes to run the accelerator is around 39 ms. However, since the result is still in projected coordinates, it still has to be converted to affine coordinates in software, which takes around 75ms. Therefore, the total time of the `ecp_mul` function call becomes around 114 ms. at maximum.

For ECDHE-RSA, the `ecp_mul` routine is called twice. Therefore, in the accelerated case of the 5.75 seconds for ECDHE-RSA, about 0.228 seconds are occupied by ECP multiplication, while the rest of the time is spent on other routines.

This means that further improving accelerator performance, for instance by using a larger design that also computes the conversion back to affine coordinates, would not have a significant impact on the total performance.

For ECDHE-ECDSA, the `ecp_mul` routine is called seven times, resulting in a total estimated time of 0.798s out of 2.03, which is a significant amount of time. In this case, further optimizing accelerator performance might be beneficial, especially considering the fact that coordinate transformation in hardware could save up to a theoretical maximum of 525 ms per connected client.

	Software	Hardware, without transform	Hardware + software transform
Runtime (s)	2.500	0.039	0.114

Table 4.6: Average runtime of ECP multiplication at 40 MHz in software and hardware

5

Conclusion

OpenVPN running on an FPGA softcore can greatly benefit from cryptographical acceleration when attempting to speed up the control channel, in order to increase the maximum amount of connected clients per server.
The two largest contributing functions to overall computation time are `mpi_exp_mod`, a function that performs large-number exponential modular multiplications which are used for RSA signing in the TLS protocol, and `ecp_mod`, a function that takes care of Elliptic Curve Point multiplication, which is used in the ECDHE key agreement protocol and ECDSA, which is used for key signing.
A relatively simple accelerator with low area usage can result in a speedup of up to 1.6 times when using ECDHE-RSA, and up to 7 times when using ECDHE-ECDSA, allowing the server to serve up to 350 clients, instead of the default 80. This is in the case where all clients should be able to connect within five minutes. Should this constraint be loosened, even more clients could easily be supported by the same FPGA accelerator core.
The accelerator core design could be fit on the same FPGA as the softcore CPU, which means it comes at little to no additional cost.

5.1. Future work

Since the presented accelerator core consists of modular subcomponents, performance and area usage could still be optimized by improving either individual subcomponents, or adding a subcomponent that can handle the transformation from projective coordinates back to affine coordinates. Especially in when using ECDHE-ECDSA, this could yield significant performance benefits.
Furthermore, the current version of the accelerator only supports NIST P-384 as a curve. Alternative implementations that support multiple curves might also be an interesting avenue to pursue, although this would certainly come at the cost of additional area usage.

Bibliography

[1] F. Mallouli, A. Hellal, N. Saeed, and F. Alzahrani, *A survey on cryptography: Comparative study between rsa vs ecc algorithms, and rsa vs el-gamal algorithms,* (2019) pp. 173–176.

[2] D. Ismoyo and R. Wardhani, *Block cipher and stream cipher algorithm performance comparison in a personal vpn gateway,* (2016) pp. 207–210.

[3] M. Kramer, F. Gerstmayer, and J. Hausladen, *Evaluation of libraries and typical embedded systems for ecdsa signature verification for car2x communication,* (2018) pp. 1123–1126.

[4] J. Qu, T. Li, and F. Dang, *Performance evaluation and analysis of openvpn on android,* (2012) pp. 1088–1091.

[5] L. Rodríguez Henríquez, N. A. Saqib, and A. Díaz-Pérez, *A fast parallel implementation of elliptic curve point multiplication over gf(2m),* Microprocessors and Microsystems **28**, 329 (2004).

[6] H. Marzouqi, M. Al-Qutayri, K. Salah, D. D. Schinianakis, and T. Stouraitis, *A high-speed fpga implementation of an rsd-based ecc processor,* IEEE Transactions on Very Large Scale Integration (VLSI) Systems **24**, 1 (2015).

[7] J. Ding, S. Li, and Z. Gu, *High-speed ecc processor over nist prime fields applied with toom-cook multiplication,* IEEE Transactions on Circuits and Systems I: Regular Papers **PP**, 1 (2018).

[8] T. Güneysu, *High performance ecc over nist primes on commercial fpgas,* (2008).

[9] H. Marzouqi, M. Al-Qutayri, K. Salah, and H. Saleh, *A 65nm asic based 256 nist prime field ecc processor,* (2016) pp. 1–4.

[10] K. Salah, *An fpga implementation of nist 256 prime field ecc processor,* (2013).

[11] H. Selma and H. M'hamed, *Elliptic curve cryptographic processor design using fpgas,* (2015) pp. 1–6.

[12] T. Wu and R. Wang, *Fast unified elliptic curve point multiplication for nist prime curves on fpgas,* Journal of Cryptographic Engineering (2019), 10.1007/s13389-019-00211-9.

[13] S. Shohdy, A. El-Sisi, and N. Ismail, *Fpga implementation of elliptic curve point multiplication over gf(2191),* (2009) pp. 619–634.

[14] S. Nn, V. Sridhar, and D. Patawardhan, *Fpga based efficient elliptic curve cryptosystem processor for nist 256 prime field,* (2016) pp. 194–199.

[15] S. Liu, L. Ju, X. Cai, Z. Jia, and Z. Zhang, *High performance fpga implementation of elliptic curve cryptography over binary fields,* (2014) pp. 148–155.

[16] A. Chatterjee and I. Sengupta, *High-speed unified elliptic curve cryptosystem on fpgas using binary huff curves,* (2012) pp. 243–251.

[17] K. Loi and S.-B. Ko, *Scalable elliptic curve cryptosystem fpga processor for nist prime curves,* IEEE Transactions on Very Large Scale Integration (VLSI) Systems **23**, 1 (2015).

[18] M. Hossain, E. Saeedi, and Y. Kong, *High-speed, area-efficient, fpga-based elliptic curve cryptographic processor over nist binary fields,* (2015) pp. 175–181.

[19] NIST, *Mathematical routines for the nist prime elliptic curves,* (2015).

[20] P. L. Montgomery, *Speeding the pollard and elliptic curve methods of factorization,* (1987).