414-bit multiplication on the ARM Cortex-M0

Sven Arissen

Supervisor: Peter Schwabe

Abstract

This thesis presents a low-level implementation of 414-bit multiplication on the ARM Cortex-M0, using four levels of Karatsuba and one level of schoolbook multiplication. This multiplication is an important part of Curve41417 created by D. Bernstein and T. Lange. This implementation requires 4014 clock cycles.

August 2015

Contents

1 Introduction **4**
1.1 Motivation 4
1.2 Organisation 4

2 Preliminaries **5**
2.1 Elliptic-curve cryptography 5
2.1.1 Introduction 5
2.1.2 Theory 5
2.2 Multiprecision multiplication 7
2.2.1 Schoolbook multiplication 7
2.2.2 Karatsuba multiplication 7
2.2.3 Others 8
2.3 ARM Cortex-M0 8
2.3.1 Registers 8
2.3.2 Loading and storing 8
2.3.3 Arithmetic instructions 8

3 Related work **9**

4 Implementation **10**
4.1 Representing big integers 10
4.2 Karatsuba 11
4.3 Overview 11
4.3.1 32-bit 11
4.3.2 64-bit 11
4.3.3 128-bit & 96-bit 12
4.3.4 224-bit & 192-bit 12
4.3.5 416-bit 13

5 Results **15**

6 Conclusion and further work **16**

Bibliography **17**

Chapter 1

Introduction

With the constant spying of dangerous individuals (and governments), the importance of cryptography for securing Internet communication is rising. The problem is that computers are getting more and more powerful and are able to break previously secure algorithms. At the same time we are using smaller and less powerful computers (like the Internet of Things) for a large part of our communication.
To get this strong but fast encryption, new algorithms are brought up. But it is also important for these algorithms to be implemented as efficiently as possible, to achieve good performance on weaker devices. This thesis is about the optimisation of large-integer multiplication (specifically 414-bit multiplication) which is used in Curve41417, an elliptic curve created by Daniel Bernstein [1]. The implementation is done on the ARM Cortex-M0 [2].

1.1 Motivation

Large integer multiplication is often used in cryptographic algorithms. 414-bit multiplication is specifically used in Curve41417 [1]. This curve is used in the Silent Circle apps [3]. Silent Circle has created two apps [4], one for calling (Silent Phone) and one for texting (Silent Text). They have also created an operating system based on the Android OS, which runs on their Blackphone [5]. These products are mostly aimed at businesses and governments and offer private encrypted communication. This is not just in transit, but also encrypted to prevent the provider from reading.

1.2 Organisation

This thesis starts with a basic overview of the related theory: elliptic curves and multiprecision multiplication. After that follows an overview of the choices that were made during the implementation and an overview of the multiplication implementation from the bottom (32-bit multiplication) to the top (416-bit multiplication). This is followed by an overview of the results from benchmarking the implementation and conclusion with possibilities for future work at the end.

Chapter 2

Preliminaries

2.1 Elliptic-curve cryptography

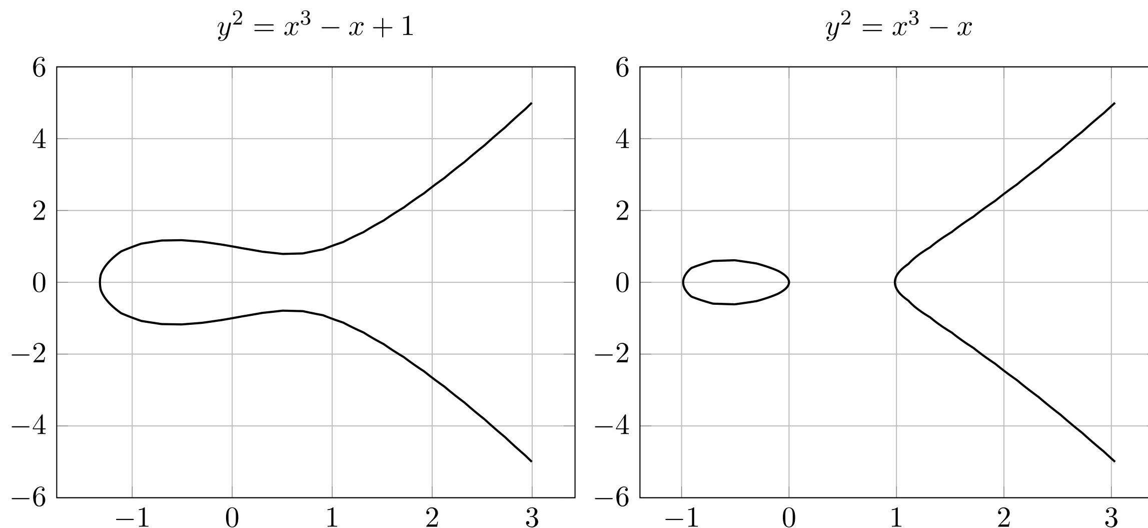
2.1.1 Introduction

Elliptic-curve cryptography (ECC) is an asymmetric cryptographic algorithm created by Koblitz [6] and Miller [7]. It is based on the properties and structure of elliptic curves, most importantly the intractability of the discrete logarithm.
ECC can be used for both key exchange (ECDH) and signing (ECDSA).
The reason that ECC is important is that many other asymmetric algorithms depend on the difficulty of finding the prime factors of an integer. However with modern technology it is starting to become easier to find these factors in a reasonable amount of time. To compensate for that the size of the keys needs to be increased, which leads to slower encryption and decryption times and also increases memory usage.
Although the idea of ECC is widely accepted by researchers, there is a lot of discussion about which curves to use. In 1999 NIST published a set of recommended standardised curves for use by governments [8], these curves where created by the NSA. Recently though there has been some discussion about the safety of these curves, with people arguing that there may be weak spots on some of them which can be exploited by the NSA [9]. This has also resulted in the creation of several other curves (of which Curve41417 is one) which are supposed to be secure.

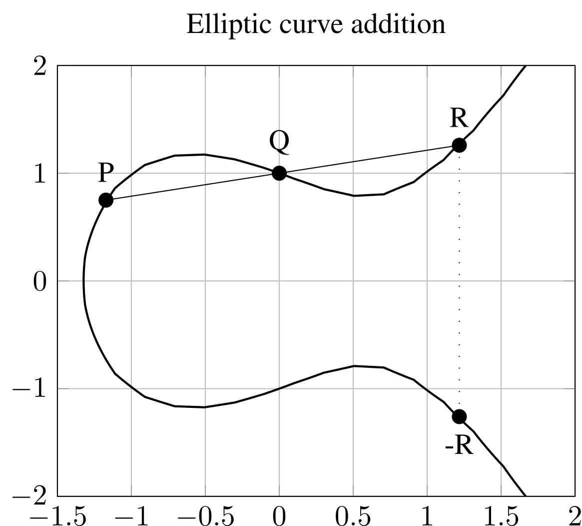
2.1.2 Theory

Real numbers

Elliptic curves over real numbers have the following equation: $y^2 = x^3 + ax + b$ Which can look like:



So the elliptic curve is the set of points satisfying the above equation together with a point at infinity: $\mathcal{O}$. This set is also a group: with $\mathcal{O}$ as identity and the inverse of a given point is the one that is symmetric on the x-axis. So the inverse of a point P with coordinates (x,y) is point -P with coordinates (x,-y) . Addition for this group is then given by: $P + Q + R = \mathcal{O}$, where P, Q and R are aligned points on the curve. This also means that: $P + Q = -R$. $-R$ is the inverse of R, since elliptic curves are symmetric in the x-axis any addition of two points, that are aligned, on the curve will give another point on the curve: the inverse of the third point.



You can also perform scalar multiplication: $Q = k \cdot P$ where P is a point on the curve and $k \in \mathbb{Z}$. The complexity of which is exponential if it is done naively: by simply adding a point to itself multiple times. But there are various algorithms, the simplest of which is $doubleandadd$ that are polynomial in complexity. However if you know P and Q and want to calculate the scalar k, you would need to solve the logarithm, which is already very difficult to calculate: this forms the trapdoor for ECC: scalar multiplication can be done relatively quickly while computing the logarithm is relatively difficult. However depending on the chosen curve it could still be pretty easy to find certain patterns which help you solve this problem. For example if you use the curve: $y^2 = x^3 - 3x + 1$, you will note that it it split in 2 parts (a left and a right

part), with some experimentation you will find that if k in $Q = k \cdot P$ is Q will be on the left part and if k is even it will be on the right part. That is why elliptic curves are generally defined in a finite field.

Finite fields

Elliptic curves are generally restricted to a finite field $\mathbb{F}_q$, when used in cryptography. This means that the formula changes to: $y^2 \equiv x^3 + ax + b \pmod{q}$ The main advantage to using finite fields are that when you use scalar multiplication: $Q = k \cdot P$. There will be a certain k where: $0 \equiv k \cdot P \pmod{q}$, after which the results from the scalar multiplication start to repeat. Which means that scalar multiplication on point on an elliptic curve on a finite field is cyclic.
Due to this instead of having to solve the logarithm, you would have to solve the discrete logarithm which is a lot harder. Currently there is no efficient way of calculating this on conventional computers, however there is in theory a way to solve this problem on a quantum computer [10].
Due to the fact that this is so much harder than for example factoring, it is possible to use far smaller keys with ECC than with for example RSA.

2.2 Multiprecision multiplication

2.2.1 Schoolbook multiplication

The idea of multiprecision multiplication is to split larger multiplicands into smaller ones. A processor generally has a multiplier that can multiply 8/16/32 bit integers, so for larger integers you need to split them, then multiply those smaller integers and add them together to get the final result. The formula for basic multiprecision multiplication of two n bit integers, a and b, is:
$(a_H \ll n/2 + a_L) * (b_H \ll n/2 + b_L) = (a_H * b_H) \ll n + (a_H * b_L) \ll n/2 + (a_L * b_H) \ll n/2 + a_L * b_L$
For a 32-bit multiplication with a 16-bit multiplier on a processor this would require four 16-bit multiplications. For a 64-bit multiplication it would require $4 * 4 = 16$, 16 bit multiplications.
In this thesis this algorithm is only used on the lowest level of the multiplication: to multiply two 32 bit numbers.

2.2.2 Karatsuba multiplication

This algorithm was created by the then 23 year Karatsuba when he was following a seminar by Kolmogorov who argued that multiplication couldn't be done any faster than n^2. Kolmogorov was so agitated because of Karatsuba's findings that he promptly cancelled the rest of the seminar. He then went on to write a paper in Karatsuba's name about his findings on the algorithm, which Karatsuba only learned about much later.
The Karatsuba algorithm [11] improves on the schoolbook multiplication by only requiring 3 half size multiplications instead of 4 at the cost of added additions and subtractions:
$H = a_H * b_H$
$L = a_L * b_L$
$M = (a_L + a_H) * (b_L + b_H)$
$(a_H \ll n/2 + a_L) * (b_H \ll n/2 + b_L) = H \ll n + (M + A - B) \ll n/2 + L$
A total of 4 levels of Karatsuba is used in this implementation.

2.2.3 Others

Other fast algorithms are Toom-Cook [12] [13] and Schönhage-Strassen [14], both of which are theoretically faster than Karatsuba, but only for even larger input sizes. These algorithms are therefore not used in this thesis.

2.3 ARM Cortex-M0

The ARM Cortex-M0 is a 32 bit microcontroller which is often used in embedded devices. It is currently the smallest ARM processor and has a very low power cost.

2.3.1 Registers

The ARM Cortex-M0 has 16 registers (R0-R15); the upper 3 (R13-R15) are used for the stack pointer, link register and program counter. The other 13 are general-purpose registers, but there are some restrictions on registers R8-R12, which can only be used in move instructions without immediates. The reason for this is that all of the M0's instruction are encoded in 16 bit. This means that R8 to R12 are mostly used for temporary storage using the MOV instruction which only takes a single cycle.

2.3.2 Loading and storing

Aside from registers R8 to R12, temporary and final results can also be stored in memory and on the stack. Registers can be stored and retrieved from the stack using the PUSH and POP instructions which can both store/retrieve up to 8 registers at a time, the cost for this is $n + 1$ cycles, where n is the number of registers stored/retrieved. This makes it most efficient to store/retrieve as many registers as possible in a single PUSH/POP.
For loading and storing in memory the LDM and STM instruction are used. They take a 32-bit (single register) location. They store/load up to 8 registers from memory and also need $n + 1$ cycles. When using an STM instruction the base register value is written back unless an ! is placed after the location register, in which case the location in memory just after the last loaded value is stored in it.
The same applies to LDM but it is also possible to store the result of a load in the location register used for the LDM, in that case the result is stored.

2.3.3 Arithmetic instructions

The arithmetic instructions used are: MUL (multiplication), ADD/ADC (addition), SUB/SBC (subtraction), EOR (exclusive or), LSR/LSL (logical shift right/left), UXTH (zero extend halfword) and ASR (arithmetic shift right) which each require only one clock cycle. The carry from an ADD/ADC or SUB/SBC is only reset if another ADD/ADC or SUB/SBC is used and not if another instruction, that doesn't change the carry, is used in between them.
I found one exception to this rule though. Apparently using a MOV instruction in between SUB/SBC instruction automatically sets the carry flag to 1. This is most likely a fault in the assembler or chip though.

Chapter 3

Related work

Relatively little work has been done on ECC and multiplication on the ARM Cortex-M0. Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez and Schwabe worked on an implementation [15] of Curve25519 [16] on various microcontrollers in 2015. One of these microcontrollers was the ARM Cortex-M0. Curve25519 is a 256-bit curve, so they worked on a 256-bit multiplication using subtractive Karatsuba. This 414-bit implementation in this thesis is partially based on their work on the 256-bit multiplication.
In 2013 Wenger, Unterluggauer and Werner [17] published a paper looking at the implementation of the secp256r1 curve on various microcontrollers including the ARM Cortex-M0. This curve is also a 256-bit curve. However there implementation was considerably slower than the one by Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez and Schwabe [15].
The only other similar work is done De Clercq, Uhsadel, Van Herrewege and Verbauwhede in 2014 [18]. They worked on the 233-bit curve sect233k1, which was implemented on the ARM Cortex-M0+ (which is an improved version of the Cortex-M0 with the same instruction set).

Chapter 4

Implementation

The Karatsuba implementation can be implemented very nicely using recursion, however that is not done in this implementation. The reason is that using recursion would require more cycles due to the need for function calls (JUMP instructions) to other parts.
Although the goal was to create a 414-bit implementation the final result is a 416 bit multiplication. The reason for this is that at the lowest level the CPU uses a 16 bit multiplication, which means that creating a exact 414-bit implementation wouldn't be any faster than a 416-bit multiplication.
The Cortex-M0 actually has a 32-bit multiplication instruction. This instruction takes 2 32-bit registers and produces a 32-bit result. This result is only the lower 32 bits of the actual result though. Which means that, for our purpose, the multiplication instruction is only really useful for at most 16-bit multiplications.
One concession that had to be made is the additional size of the output array. Normally this would have been an 104-byte array however due to the need for additional storage of partial results some additional storage space was necessary. The output array is now 172 bytes.
The implementation is constant-time, meaning that no matter what the inputs are, the CPU will always need the same amount of clock cycles to calculate the multiplication. This is important because it prevents timing attacks which rely on differences in execution time to find weaknesses in an implementation.

4.1 Representing big integers

The implementation is done using the Karatsuba algorithm. The 416-bit inputs are split into a lower 224-bit and an upper 192-bit.
The 224-bit integer is further split into a lower 128 and higher 96-bit. The 128-bit integer is split into two 64-bit integers and the 96-bit integer is split in a lower 64-and a higher 32-bit.
The 192-bit integer is split into 96-bit integers, which are split into a lower 64 and higher 32-bit.
The 64-bit integers are split into two 32-bit integers, which are then split into two 16-bit integers which can be directly multiplied by the Cortex-M0 multiplication instruction.
The choice to not split 416, 224 and 96 into equal parts is because of the necessity to shift the addition of the middle part by 16 bits. Since each of the 3 integers (H, L, M) in the middle is added separately, it would approximately require $3 \cdot 4 \cdot R$ (where R is the number of registers of the integer to be added) additional cycles to add the middle part.
Another problem with splitting evenly is that if the inputs are an uneven amount of registers (this is the

case with 416 for example). The high part of the inputs would need to be shifted left by 16 bits.
The 32, 64 and 128 bit implementations were made by B. Haase and A. H. Sánchez [15] and were already optimised.

4.2 Karatsuba

Instead of the normal Karatsuba algorithm, the subtractive Karatsuba algorithm is used. The reason for this is that during the calculation of the middle part (M) the result of the addition of the higher and the lower part can have a carry. That would mean that instead of multiplying $n/2$ bit integers you are multiplying $n/2+1$ bit integers. The formula for subtractive Karatsuba is:
$H = a_H * b_H$
$L = a_L * b_L$
$M = (a_L - a_H) * (b_L - b_H)$
$(a_H \ll n/2 + a_L) * (b_H \ll n/2 + b_L) = H \ll n + (L + H - M) \ll n/2 + L$ However in this implementation a variant of this formula is used instead [15], the middle part is calculated as:
$M = |a_L - a_H| * |b_H - b_L|$.

4.3 Overview

The multiplication is very modular: every level of Karatsuba splits the inputs and performs 3 smaller multiplications. Except for the 32-bit multiplication which is implemented using schoolbook multiplication and uses 4 smaller multiplications.

4.3.1 32-bit

The 32-bit implementation is implemented using schoolbook multiplication. Using Karatsuba is inefficient here due to the small size. Neither the stack nor memory is used for this multiplication as only 2 upper registers R8 and R9 are necessary as extra storage. The 32-bit multiplication starts by splitting each of the inputs in 2 16-bit parts using logical shift left (LSL) and zero extend half word (UXTH). Then all 4 of the necessary multiplications are done. After which each of the 2 middle results is split into 2 parts of 16 bits, the lower is added to the higher 16 bits of the low result register and the upper is added with carry to the lower 16 bits of the high result register.

4.3.2 64-bit

The 64-bit multiplication uses 1 level of Karatsuba. It does the calculation of the inputs for the middle multiplication first and stores them in the upper registers (R8 and R9). Then a 32-bit multiplication is done to calculate the lower part. Immediately after that a 32-bit multiplication is done to calculate the higher part. After that the lower 32 bits of the high part are added to the higher 32 bits of the low part and the carry is added to the higher 32 bits of the high part. After that the middle multiplication inputs are moved to the lower registers from the higher registers and the lower 32 bits of the low part are moved to the higher registers to create enough free registers for another 32-bit multiplication. Then the middle multiplication is done. After correction of the middle result with the negative the lower 32 bits of the

middle multiplication is added to the addition of the lower 32 bits of the high part and the higher 32 bits of the low part. And the higher 32 bits of the middle multiplication is added to the addition of the lower 32 bits of the high part and the higher 32 bits of the low part. After this the lower 32 bits of the low part are moved back to the lower registers and added to the higher 32 bits of the low part. Then the higher 32 bits of the high part are added to the lower 32 bits of the high part. Finally any possible carries are added to the highest 32 bits of the result.

4.3.3 128-bit & 96-bit

The 128 and 96-bit multiplication use 2 levels of Karatsuba. They mostly have a similar structure, the main difference being that for 128-bit the high multiplication is a 64-bit multiplication and for 96-bit the high multiplication is a 32-bit multiplication. These multiplications start with calculating the 64-bit low multiplication. The lowest 64 bits are stored in memory (these don't need to be changed any more) and on the stack to use later in the calculation of the middle part. The higher 64 bits are stored in the higher registers. Then the middle inputs are calculated and stored on the stack. Then the 64/32 bit high multiplication is done. After that like on the other Karatsuba levels the addition of the higher 64 bits of the low part and the lower 64 bits of the high part are calculated and the carry is added to the highest 64 bits. The highest 64 bits are stored in the upper registers while the addition of the lower 64 bits of the high part and the higher 64 bits of the low part are stored in the stack. The inputs for the middle multiplication are popped from the stack and the 64-bit middle multiplication is executed. After that the final calculations are done using the information in the higher registers, on the stack and in the lower registers similar to the how it is done for the 64 bit multiplication.

4.3.4 224-bit & 192-bit

The 224 and 192-bit multiplication use 3 levels of Karatsuba. They mostly have a similar structure, the main difference being that for 224 the low and middle multiplications are 128-bit multiplications and for the 192 the low and middle multiplications are 96-bit multiplications.
First the lower 128/96-bit multiplication is done. The lower half of the result from the lower multiplication is stored in memory, while the higher half is pushed on the stack. After this the inputs for the higher multiplication are necessary again, because of this the inputs for them were store on the stack in the beginning and are now retrieved again. After this the higher 96-bit multiplication is done. Then similarly to the other multiplications the addition of the higher part of the low multiplication and the lower part of the high multiplication is calculated. This is then stored in memory, while the higher part of the high multiplication is stored on the stack. After this the middle multiplication inputs are calculated. There is not enough room to have all of these stay in the registers. So the inputs for the middle multiplication are stored in memory, directly after the addition of the higher part of the low multiplication and the lower part of the high multiplication. These can then be used in the middle 128/96-bit multiplication. After the middle multiplication is done, the result is corrected with the negative. Then the higher part of the low half and the lower part of the high half are finalised by adding the result of the middle multiplication and the higher part of the high multiplication and the lower part of the low multiplication. The results from this are stored in memory. The carries from these additions are then added to the highest registers with are then also stored in memory.

4.3.5 416-bit

The most complicated of the multiplications is the upper one (the 416 bit multiplication), this is mostly due to the sizes of the inputs and result. The inputs are both 13 registers large, which means they can't be directly loaded and the final result is a total of 26 registers large.

Lower 224-bit multiplication

So at the start only the first 128 bits of each input are loaded from memory, while the remaining 96 bits are loaded during the 224 multiplication which result in the lower 448 bits of the final result. This result is already 14 registers large which is why it is directly stored to memory at the location of the result during the calculation of the lower part. The lower 7 registers of the result of the lower multiplication (L) are immediately final and only need to be used during the calculation of the middle part. The upper 7 registers still need the addition of the lower 7 registers of the high part (H) and the lower 7 registers of the middle part (M) to be final.

Higher 192-bit multiplication

After this the higher part (H) is calculated using the upper 6 registers of both inputs. This results in a 12-register result which is also directly stored in the resulting array. However the high part isn't final yet as the lower 7 registers still need the addition of the higher 7 registers of the lower part (L) and the higher 7 registers of the middle part (M). The higher 5 registers of the high part (H) still possibly need the addition of various carries and correction from the sign.
In order to save room the higher 7 registers of the low part (L) and the lower 7 registers of the high part (H) are immediately added together (since they're not needed separately). This can than be stored once and later used for both the calculation of the higher 7 registers of the low part (L) and the lower 7 registers of the high part (H). The carry of this addition is immediately added to the higher 5 registers of the high part (H). Because these 5 registers are later added to the middle this carry has been added to both of the places it needs to be added.
The 5 high registers of the high part (H) are then pushed to the stack to free up more room in memory.

Middle 224-bit multiplication

The most interesting part here is the calculation of the inputs for the middle multiplication. Using the above formula for subtractive Karatsuba, for the first input (a) the higher 6 registers are subtracted from the lower 7. This means that the seventh register of the lower part of the input is basically subtracted by 0 (so it doesn't change at all). For the second input (b) the subtraction is done the other way around the lower 7 registers are subtracted from the higher 6, this means that the final subtraction is 0 - the value of the seventh low register. In both cases the a final subtraction is done where a (unimportant) register is subtracted from itself with carry. This means that if the subtraction ended with a carry (this means the result is negative) the value in that register will be $0xFFFFFFFF$, if there was no carry it will simply be 0. This value will then be xored with the result of the accompanying subtraction, effectively taking the ones-complement if the result was negative and leaving the result unchanged if it was positive. After that the value ($0xFFFFFFFF$ or 0) will be subtracted from the result of the subtraction. These final results will then be used as inputs to the middle multiplication (M). The two registers that contain the sign will

be xored with each other to determine the sign of the middle multiplication and this sign is pushed so it can be used later. After this the middle multiplication is done, the result of which is stored in memory. This is also the reason some extra space was needed: normally 26 registers of space would have sufficed for the storage of the result, but before starting the middle multiplication there are already 14 registers stored (the lower 7 registers of the lower multiplication and the addition of the higher 7 registers of the lower multiplication and the lower 7 of the higher multiplication). The result of the middle multiplication can also be 14 registers large though, so at least 28 registers are required to store that.

Adding everything together

After the calculation of the middle part there are still 3 steps left:

- Finishing the second (step 1) and third (step 2) 7 registers by adding the middle multiplication and the lower 7 registers of the low multiplication and the higher 7 registers of the high multiplication.
- Finishing the final 5 registers by adding the necessary carries and correcting with the sign.

Before that though the final result of the middle multiplication is corrected with the sign bit by xoring them. After that the second 7 registers are finalised by adding the lower part of the middle multiplication and the lower 7 registers of the low multiplication. A temporary backup of the addition of the higher 7 register of the low multiplication and the lower 7 register of the high multiplication is stored in memory, since this is needed to calculate the third 7 registers.
These third 7 registers are then calculated from the backup with the addition of the higher part of the middle multiplication and the higher 5 registers of the high multiplication. The carries of the previous 2 additions are also added to this part.
After that all that is left is adding the 2 carries of the previous 2 additions and correcting the final 5 registers using the sign.

Chapter 5

Results

Below are the cycle counts for each of the (sub)multiplications:

Bits	Cycle count
32	17
64	81
96	290
128	332
192	1013
224	1180
416	4014

Table 5.1: Cycle results

The 32, 64 and 128-bit cycle counts are taken from Düll, Haase, Hinterwälder, Hutter, Paar, Sánchez and Schwabe's [15] paper on the 256-bit curve as the implementation was made by them. The cycle counts of the 96, 192, 224 and 416-bit multiplications were calculated using the systick counter on the development board and subtracting the function call overhead (the storage of the initial values of the registers at the start and retrieving them at the end).
All benchmarks were run on the STM32F0Discovery board with a STM32F051R8 microcontroller. This board has 64 KB of flash memory and 8 KB of RAM. The implementation can be found here: `https://github.com/GoldnEagle/CortexM0Mul414`

Chapter 6

Conclusion and further work

This thesis has presented a working version of 414 bit multiplication on the ARM Cortex-M0 using three levels of Karatsuba.

There is still room for improvement though, relatively little time was left to work on optimisation, so there a lot could be gained from improving the current implementation especially on the upper level (the actual 416 by 416 multiplication).

There is also room for improvement on the 96, 192 and 224-bit multiplications, these are currently structured very similarly to the 128-bit multiplication (in the case of the 96-bit multiplication) and the 256-bit multiplication (in the case of the 192 and 224-bit multiplication). However due the fact that the 96, 192 and 224-bit multiplication take less register space, it is most likely possible to improve these multiplications as well.

The choice was also made to increase the size of the result array to have some additional storage space for temporary results. It would be nicer though to have these results stored in the CPU stack instead. This is not a big problem though, as in the end this multiplication would be used in an ECC implementation. This would look something like this:

```
gf_mul(gfe *r, const gfe *a, const gfe *b){
    uint32_t t[43];
    mul414(t, a, b);
    reduce(r, t);
}
```

The larger output is only used very locally during the 414-bit multiplication after which the result will need to be reduced modulo r anyway.

Bibliography

[1] D. J. Bernstein, C. Chuengsatiansup, and T. Lange. Curve41417: Karatsuba revisited. 2014. http://eprint.iacr.org/2014/526.pdf.

[2] Cortex-M0 processor - ARM. http://www.arm.com/products/processors/cortex-m/cortex-m0.php.

[3] Silent Circle: This one goes to 414. https://www.silentcircle.com/blog/this-one-goes-to-414/.

[4] Software: Silent Circle. https://www.silentcircle.com/products-and-solutions/software/.

[5] Silent OS: Silent Circle. https://www.silentcircle.com/products-and-solutions/devices/silent-os/.

[6] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, pages 203–209. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/home.html.

[7] V. Miller and S. Victor. Use of elliptic curves in cryptography. *Advances in Cryptology — CRYPTO '85 Proceedings*, 218:417–426, 1986. http://dx.doi.org/10.1007/3-540-39799-X_31.

[8] Recommended elliptic curves for federal government use, 1999. http://csrc.nist.gov/groups/ST/toolkit/documents/dss/NISTReCur.pdf.

[9] D. J. Bernstein and T. Lange. Security dangers of the NIST curves, 2013. http://cr.yp.to/talks/2013.05.31/slides-dan+tanja-20130531-4x3.pdf.

[10] P. Schor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26:1484–1509, 1997.

[11] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

[12] A. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. http://www.de.ufpe.br/~toom/my-articles/engmat/MULT-E.PDF.

[13] S. Cook. On the minimum computation time of functions. 1966. http://cr.yp.to/bib/1966/cook.html.

[14] A. Schönhage and V. Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7:281–292, 1971. http://link.springer.com/article/10.1007/BF02242355#page-1.

[15] M. Düll, B. Haase, G. Hinterwälder, M. Hutter, C. Paar, A. H. Sánchez, and P. Schwabe. High-speed Curve25519 on 8-bit, 16-bit and 32-bit microcontrollers. 2015. https://cryptojedi.org/papers/mu25519-20150417.pdf.

[16] D. J. Bernstein. Curve25519: new diffie-hellman speed records. *Proceedings of PKC 2006*, 2006. http://cr.yp.to/ecdh/curve25519-20060209.pdf.

[17] E. Wenger, T. Unterluggauer, and M. Werner. 8/16/32 shades of elliptic curve cryptography on embedded processors. *Progress in Cryptology - INDOCRYPT 2013*, pages 244–261, 2013. http://link.springer.com/chapter/10.1007/978-3-319-03515-4_16.

[18] R. De Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede. Ultra low-power implementation of ecc on the arm cortex-m0+. *DAC '14 Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014. http://dl.acm.org/citation.cfm?id=2593238.