ECC optimization on Sandy Bridge

The cost of cofactor $h = 1$

Daan Sprenkels

Advisor: dr. P. Schwabe
Second Reader: prof. dr. L. Batina

March 11, 2019

Abstract

The nontrivial cofactor in Curve25519 has introduced design risks in complex cryptographic protocols. While the Decaf and Ristretto point-compression schemes can be used to mitigate these risks, they can also be prevented by using traditional prime-order curves.

Arithmetic on these prime-order curves can safely be implemented using the complete addition formulas from Renes, Costello, and Batina. While Curve25519 is often presumed to be considerably faster than traditional Weierstrass curves, this has never been quantified accurately. In this thesis, we aim to evaluate the actual performance benefit that Curve25519 provides over traditional Weierstrass curves.

We select a prime order curve that is similar to Curve25519. Using the Renes-Costello-Batina addition formulas, we implement an algorithm for variable-base-point scalar multiplication. The implementation targets Intel's Sandy Bridge microarchitecture, and leverages the $4\times$ vectorized double-precision floating-point arithmetic that AVX provides.

When we compare the performance of our implementation to the performance of the Sandy2x implementation, we observe that Sandy2x is a factor 2.5 faster than our implementation. Moreover, we see that the "cost of completeness"—i.e. the general overhead of the Renes-Costello-Batina formulas—is about a factor of 1.4.

Contents

Chapter 1

Introduction

Elliptic curve cryptography (ECC) is the most used building block in modern public-key cryptography. ECC was independently introduced by [49] in 1986 and [41] in 1987. Elliptic curves provide a faster alternative to traditional integer-group based crypto primitives. In July 1999, the U.S. National Institute of Standards and Technology (NIST) published their recommendations, to be included in FIPS 186-2 [51], which stipulated the use of elliptic curves in cryptography. Included in the standard were the parameters for 5 prime-order elliptic curves, varying in security level. In the present day, these curves are still used extensively by many prominent protocols, most notably TLS.

1.1 Cofactors and completeness

Back at the turn of the century, when elliptic-curve cryptography was popularized, there existed no complete addition laws for Weierstrass curves that were deemed efficient enough to be implemented. During that time, only the work from Bosma and Lenstra [15] had been published.

As a result, implementors chose to use incomplete addition laws in their software. These implementations dealt with exceptions using branching. Over time, side-channel attacks on these algorithms were deemed relevant, while more publications were published about the insecurity of these algorithms [28].

In response, Bernstein invented Curve25519 [8]: a Montgomery curve that came with complete addition formulas. When used with the Montgomery ladder [50], these formulas allowed for scalar-multiplication algorithms that were faster than any algorithms for other Weierstrass curves of the time. Furthermore, these addition formulas—and the Montgomery ladder—have been presumed to be easier to implement in a side-channel resistant manner.

Although Curve25519's formulas are easier to implement, is has been seen that they still allow for insecure implementations, as has been demonstrated by [40] and [29].

More recently, the IETF's Crypto Forum Research Group (CFRG) initialized new efforts to standardize more modern elliptic curves. Older standardized curves, like the ones from NIST [51] and Certicom [19], only considered Weierstrass curves. Modern curve designs, like Montgomery curves and (twisted) Edwards curves [9] were never considered for inclusion. After quite some debate and a year of drafting, the CFRG accepted RFC7748 [43], which recommends the use of Curve25519 and Curve448 [32], a similar curve proposed by Hamburg in 2015.

A downside to Curve25519 is its nontrivial cofactor. This is not a problem for simple protocols, like Diffie-Hellman [24] and the Schnorr signature scheme [59], as implemented by [9]. However, for more involved protocols, a cofactor $h = 1$ is often preferable when an algorithm's simplicity is favored.

This flaw has led to a major vulnerability in the Monero cryptocurrency [53]. In 2017, it was found that the cryptocurrency was vulnerable to a small-subgroup attack, which allowed the attacker to double-spend. This attack could have been prevented by properly hashing the inputs, or checking that the input points are not in a small subgroup.[1] However, this class of attacks would have been completely invalid if a prime-order group had been used.

Regardless, the nontrivial-cofactor flaw was fixed in 2015 when Hamburg published Decaf [31]. Decaf, and its extension Ristretto, are point-compression schemes that are used to construct a prime-order group over Edwards, Twisted Edwards, and Montgomery curves. Ristretto allows us to assume a prime-order group when constructing cryptographic protocols. In other words, we do not have to fall back to more traditional (Weierstrass) curves; we can just implement the relatively simple group arithmetic of Curve25519 and encode group elements using Ristretto.

[1] While it may be tempting to dismiss the lack of input validation as a "rookie mistake", it has been highly debated whether to validate input points on Curve25519 *at all*. This debate has been held mainly in the context of the Curve25519-based key-exchange protocol *X25519*. While the validation of points is ineffective in X25519, it *can* be important in other protocols based on Curve25519 to prevent small-subgroup attacks. Extra confusion is caused, because the term "Curve25519" has often been used to refer to the "X25519" key exchange, instead of just the curve.

1.2 Complete formulas for Weierstrass curves

Nevertheless, we would like to take a step back and look at the alternative method of using more traditional prime-order curves. That is, the main benefit of using Curve25519 in the first place, is that its complete formulas are faster than any complete formulas for Weierstrass curves.[2] However in 2017, Renes, Costello, and Batina published a set of complete addition formulas for Weierstrass curves [57]. At the moment of writing, these formulas are still the fastest complete formulas for general Weierstrass curves in existence. They have been implemented in OpenSSL, and an optimized implementation has been constructed in hardware [46]. Still, no optimized software implementations of these formulas have as of yet been built.

Therefore, we feel that this presumption—that complete implementations for Weierstrass curves are considerably slower than implementations for Curve25519—is unproven. Because of the complexity of the Renes-Costello-Batina formulas, it is expected that they are indeed slower than the Curve-25519 formulas; but we are interested in a more concrete measure of the actual difference. This thesis aims to provide such a measure.

Although we think it always preferable to implement complete formulas for Weierstrass curves in software, let us articulate that by no means we recommend the use of the Renes-Costello-Batina formulas above the use of Curve25519 with Ristretto for new protocols. This thesis aims only to quantify the benefit of Curve25519 over Weierstrass curves in terms of performance.

1.3 Our contribution

In this work, we implement a Weierstrass curve with parameters similar to Curve25519's in software. To be able to make an apt comparison, we select what we consider a current state-of-the-art implementation for Curve25519: Sandy2x [21]. This implementation targets the Sandy Bridge microarchitecture, and hence our implementation will target the same microarchitecture. We will implement an algorithm for *variable-base-point scalar multiplication*, which is the most general building block for elliptic-curve computations.

In Chapter 2, we introduce some of the concepts that will be used in the rest of the thesis. Skilled cryptographers should be able to skip this chapter entirely. Before constructing the implementation, we will select a proper Weierstrass curve in Chapter 3. This curve should be similar to Curve25519 in parameters, but should be a traditional prime-order curve. After constructing the curve,

[2] For example, on one of their "SafeCurves" pages [11], Bernstein and Lange suggest that no viable complete addition laws exist for Weierstrass curves.

we will devise the finite-field arithmetic in Chapter 4. On Sandy Bridge, we can use SIMD (single instruction, multiple data) instructions to vectorize some of our computations. In particular, we will look at $2\times$ vectorization using 64-bit integers; and $4\times$ vectorization using double precision floating points. In Chapter 5, we will apply the finite-field arithmetic to implement the Renes-Costello-Batina addition formulas for our curve. In Chapter 6, we will put the pieces together in a signed-window scalar-multiplication algorithm. Then, in Chapter 7, we will look at the resulting performance of our algorithm. In the end, we hope to be able to state—as accurately as possible—the *actual* benefit of Curve25519's construction, compared to the general Weierstrass format. I.e. we hope to find the added cost of the Renes-Costello-Batina formulas, compared to the Curve25519 formulas.

Chapter 2

Preliminaries

2.1 Elliptic curve cryptography

The field of cryptography can be divided into two main categories of algorithms. *Symmetric cryptography* involves cryptographic algorithms with only one key. This key is used for both the encryption and decryption of a ciphertext. Similarly, the key can be used to generate and verify message authentication codes on messages.

With *asymmetric cryptography* we are dealing with two keys: A *public key* and a *private key*. Two different keys allow us to build much more advanced cryptographic algorithms, including *key exchange algorithms* and *digital signature algorithms*. In the next section we will look at the Diffie-Hellman key exchange as an example.

2.1.1 The discrete log problem

All crypto algorithms rely on some kind of computationally hard problem. In asymmetric crypto, the currently most prominent problem is the *discrete logarithm problem* (DLP). When considering a cyclic group G, the DLP states that, given some group elements g and h where $h = g^x$, it is hard to find x. We can use the hardness of this problem to construct more high-level asymmetric crypto primitives. These algorithms are built in such a way that if the attacker breaks the algorithm, they must have found a solution for the problem that was presumed to be hard to solve. In other words, if the problem is indeed hard to solve, then the algorithm will be at least as hard to break.

The computational Diffie-Hellman problem is similar to the discrete logarithm problem [47]. It assumes that in some mathematical group $\langle g \rangle$, it is computationally hard to find $g^{x_1 x_2}$ given g^{x_1} and g^{x_2}. This assumption is

used to construct the Diffie-Hellman key-exchange [24]. The Diffie-Hellman key exchange protocol is used to establish a shared secret between two participants over an insecure channel.

In this protocol, the participants Alice and Bob (A and B) first generate random private keys $x_A, x_B \in_R \mathbb{Z}_n^*$. Alice and Bob then compute their public shares $h_A = g^{x_A}$ and $h_B = g^{x_B}$ respectively. They exchange their public shares and keep their private keys secret. Now Alice computes $s = h_B^{x_A}$ and Bob computes $s = h_A^{x_B}$. Both end up with the same secret s, because $s = (g^{x_A})^{x_B} = (g^{x_B})^{x_A}$. By definition, the Diffie-Hellman problem states that given h_A and h_B is i hard to find s, which makes the protocol secure against any eavesdropping.

Traditionally, the group used for protocols like the Diffie-Hellman key exchange is the multiplicative group of integers modulo a prime, often denoted by $(\mathbb{Z}_p^*, \cdot)$. Obviously the Diffie-Hellman problem is not hard to compute for small values of p. One can simply iterate over all the possible values of x. Indeed, the order of g must be large enough to obtain an algorithm that is not trivial to break.

The main drawback of using this group is its subexponential security. In other words, subexponential attacks, such as the index calculus algorithm [1], break the discrete logarithm and Diffie-Hellman assumptions for these integer groups. This is mitigated by increasing the group size. For a security level of 128 bits—that is, withstanding attacks that need at least in the order of 2^{128} operations—the recommended group size is 3072 bits [2].

2.1.2 Weierstrass curves

Another class of groups that are believed to uphold the DLP is the groups of points on some *elliptic curves*. This was independently proposed by Miller [49] and Koblitz [41]. The elliptic curves used in cryptography are defined over finite fields. These can be described by a Weierstrass curve equation:

$$y^2 = x^3 + ax + b \tag{2.1}$$

Added to the points described by the curve equation is the *point at infinity*, symbolized by $\mathcal{O}$. Intuitively, one can regard $\mathcal{O}$ as the point that lies everywhere where $y = \pm\infty$. We will use this point later when we define addition on the curve.

One interesting property of elliptic curves is that a line drawn through any two points on the curve intersects the curve in *exactly* one other point. We can use this property to define the addition of two points on the curve.

To describe the addition $P + Q = R$, where $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$, we first consider the case where $x_P \neq x_Q$. We will take the line that intersects the points P and Q and look for the third intersection of this line with the curve; we call this point $-R$. We reflect $-R$ around the x-axis to get $R = P+Q$. An illustration of this geometric description is shown in Figure 2.1 on an elliptic curve over $\mathbb{Q}$.

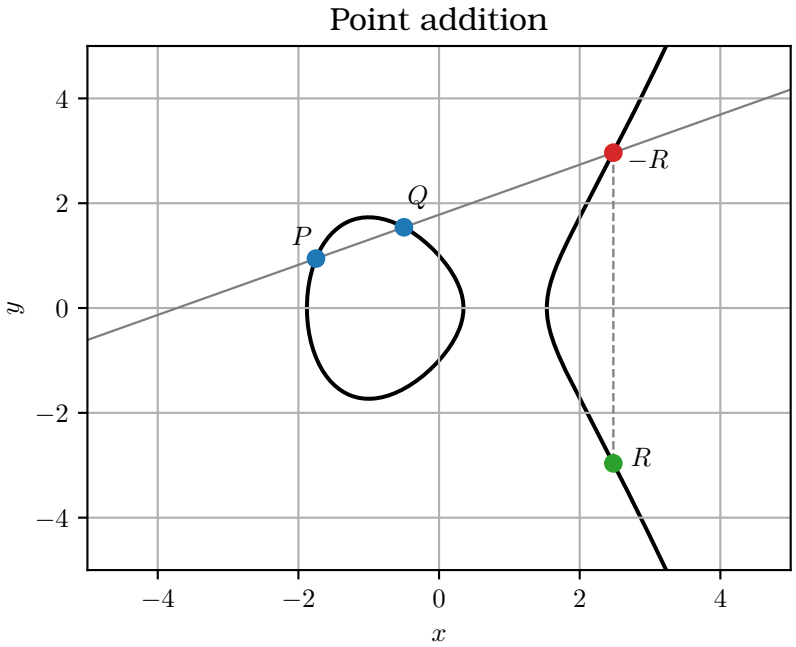

Figure 2.1: Addition of P and Q on some elliptic curve over $\mathbb{Q}$.

Of course this geometric description does not apply when $P = Q$. We cannot draw a line between P and Q, instead we draw the line tangent to the curve that goes through P. Then we take the other point that intersects the curve, and its reflected point around the x-axis is $R = 2P$.

An other exception applies whenever the constructed line is vertical. This happens when:

1. Two different points have the same x coordinate, or
2. A point is added to itself, where the tangent to the curve is vertical. This is only the case for points where $y = 0$.

In these cases, we end up with a vertical line that does not intersect the curve in any other regular point. We have previously introduced the point at infinity $\mathcal{O}$ for this exception. Whenever this situation occurs, we define the sum of the points to be $\mathcal{O}$.

2.1.3 Addition formulas

The geometric notion of point addition on elliptic curves can be put into *addition formulas*. These formulas are used to compute $R = P + Q$. They can be derived from the geometric description of addition. Recall that there are four distinct cases that can occur:

1. $x_P \neq x_Q$
2. $x_P = x_Q$ and $y_P = y_Q \neq 0$
3. $x_P = x_Q$ and $y_P = -y_Q$
4. $x_P = x_Q$ and $y_P = y_Q = 0$

Case 1. In the first case we are dealing with regular addition. To compute $R = (x_R, y_R)$, we first compute the slope λ:

$$\lambda = \frac{x_P - x_Q}{y_P - y_Q} \tag{2.2}$$

Then the resulting coordinates of $R = P + Q$ are:

$$x_R = \lambda^2 - x_P - x_Q \quad \text{and} \quad y_R = \lambda(x_P - x_R) - x_P \tag{2.3}$$

Case 2. When $P = Q$ we are doubling a point. Recall that the slope of some equation is equal to $\frac{\mathrm{d}y}{\mathrm{d}x}$ at the point (x_P, y_P). Using the curve equation, we can calculate λ in the doubling case:

$$\lambda = \frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\mathrm{d}(x^3 + ax + b)}{\mathrm{d}x}\frac{\mathrm{d}y}{\mathrm{d}(y^2)} = \frac{3x^2 + b}{2y}$$

Filling in the coordinates from P yields the slope at the point:

$$\lambda = \frac{3x_P^2 + b}{2y_P} \tag{2.4}$$

Otherwise the coordinates of $R = 2P$ are computed as in equation 2.3.

Cases 3 and 4. In the third and fourth case we already saw that we are dealing with a vertical slope. This slope intersects the curve at the point at infinity. Whenever we hit one of these cases, we define the sum of P and Q to be $\mathcal{O}$.

Up until this point, we have refrained from choosing a mathematical structure for our coordinates to exist in. Figure 2.1 suggests using a Weierstrass curve defined over the rational numbers. In cryptography, however, we work with elliptic curves defined over finite fields. Therefore, elliptic curves

actually look quite different from the intuitive depiction in Figure 2.1. They are only defined for discrete values of x and y, so the amount of points on the curve is limited. Still, the geometric description of the curve holds and the addition formulas are unchanged.

For example, let us consider the elliptic curve E defined over $\mathbb{F}_{11}$ described by the following curve equation:

$$E : y^2 = x^3 - 3x + 1 \tag{2.5}$$

Note that this is just a regular Weierstrass equation, with $a = -3$ and $b = 1$. To add the points $P = (0, -1)$ and $Q = (4, 3)$, we apply the regular addition routine: Draw a line through both points and find a third point on this line (which is $(8, -4) = -R$). After reflecting around the x-axis, we find $P + Q = (8, 4) = R$. An illustration of this example is shown in Figure 2.2.

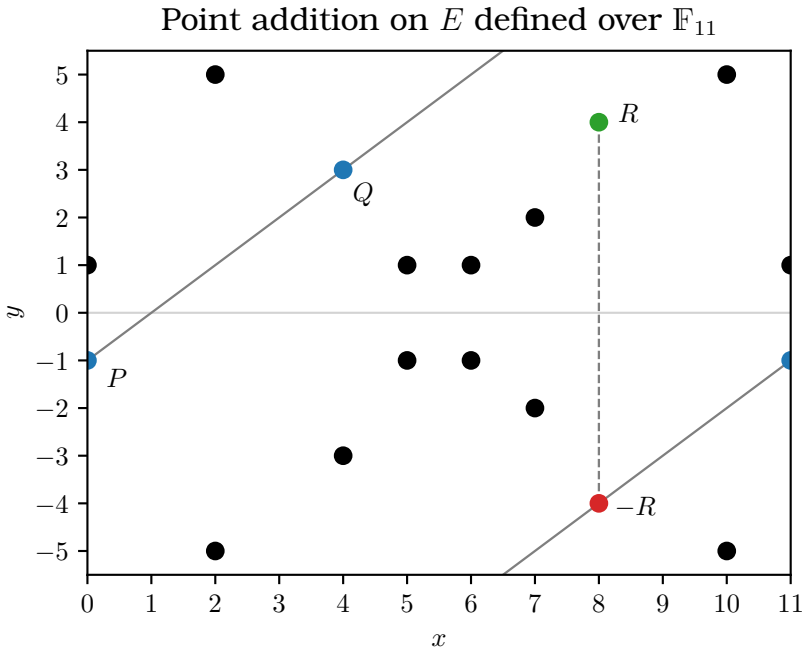

Figure 2.2: Addition of $P + Q = (0, -1) + (4, 3)$ on $E : y^2 = x^3 - 3x + 1$ defined over $\mathbb{F}_{11}$, resulting in $R = (8, 4)$. Note that because we are working in $\mathbb{F}_{11}$, we are computing everything modulo 11. Accordingly, the intersection line wraps around in the figure.

2.1.4 Using curves for cryptography

The truly useful property of elliptic curves and their addition laws is that the points on an elliptic curve form a cyclic group with the point addition

operation. As such, we have a structure that looks a lot like the groups from Subsection 2.1.1.

Where we have the DLP for integer groups, we have a new assumption for elliptic curves: The *elliptic curve discrete logarithm problem* (ECDLP). The ECDLP states that, given some points P and Q, where $Q = [k]P$, it is hard to find the scalar k. The ECDLP is not hard for *all* elliptic curves though. Only a few elliptic curves actually satisfy the properties that make the ECDLP hard. For a curve to be secure for cryptographic use, it has to meet several criteria.

Most importantly, recall from Subsection 2.1.1 that we need a group with a large order. Fortunately, choosing an elliptic curve with a large order is easy. Hasse's theorem [33] states that

$$|N - (q + 1)| \leq 2\sqrt{q},$$

which means the number of points on an elliptic curve N is of roughly the same size as the size of the underlying finite field $\mathbb{F}_q$. Effectively, if we choose a finite field $\mathbb{F}_q$ with q around 2^{256}, every curve constructed over this field will have around 2^{256} $\mathbb{F}_q$-rational points.

However, the security of the curve is not exactly determined by the total order of points on the curve, but by the order ℓ of the largest subgroup in the elliptic curve. The Pohlig-Hellman algorithm can be used to reconstruct the discrete logarithm in the large group from the discrete logarithms in the subgroups [54]. This makes breaking the DLP for the whole group only as hard as breaking the DLP for its largest subgroup.

Because of the Sylow theorems, if N is a smooth number (i.e. it has a lot of prime factors), the curve has a lot of subgroups. Then, the complexity of the attack (and thus the security of the curve) is greatly reduced. Therefore, N is often chosen to be a prime number. The curves where N is a prime are generally called *prime order curves*.

Still, in some classes of elliptic curves—like Montgomery curves and Edwards curves—N cannot be prime. In these cases, N is chosen to be "almost prime". For example, in Curve25519 [8], $N = h \cdot \ell$ where ℓ is a large prime number and $h = 8$. This additional factor h is called the *cofactor*. It is often safe to have a cofactor $h > 1$, but when a cryptographic protocol is implemented poorly, it can be vulnerable to small subgroup attacks. This attack is explained further in Subsection 3.3.

What is left to choose is the size of the subgroup order ℓ. It has to be large enough such that an attacker cannot find the solution to the ECDLP using any algorithm. Recall that with the integer groups—because of the index calculus algorithm—we need an order of around 3000 bits. However, index

calculus can only be used to solve the integer DLP, not the elliptic-curve DLP. At the moment of writing, the best general attack for solving the ECDLP is Pollard's ρ Algorithm [56]. It reduces the complexity of finding the discrete logarithm of two elements in any group to $\mathcal{O}(\sqrt{\ell})$. So to get a security of k bits, we need to choose $\ell \geq 2^{2k}$. In cryptography, an often desired security level is 128 bits, so ℓ is chosen to be around 2^{256}. For example: Curve25519's ℓ is around 2^{252}, which provides a security level of around 126 bits against Pollard's ρ attack.

This difference in group size translates directly to the performance of implementations of the algorithms described thusfar. In (EC)DLP-based algorithms, public keys are always group elements. It is clear that transmitting public keys of 3072 bits takes more than ten times longer than transmitting keys of 256 bits. Moreover, computations with numbers of 3072 bits take a lot longer than computations with only 256 bits. The consequence is that elliptic-curve implementations generally need less RAM and can achieve much better performance than their integer-based counterparts.

2.2 Scalar multiplication

In the previous section we have seen the addition laws for Weierstrass curves. We have seen that elliptic-curve arithmetic can be used to construct cryptographic protocols, like the Diffie-Hellman key exchange. The most important computation that is executed in these protocols is the operation $Q = [k]P$, where $k \in_R \mathbb{Z}_N$. This operation, which multiplies a point P by a scalar k, is called *scalar multiplication*. On curve points, only addition and subtraction are actually defined, and there exists no trick to natively compute the scalar multiplication of a point. So we will have to compute the scalar multiplication by repeatedly adding the base point to itself. Such a computation is called an *addition chain*.

The most naive method of computing the scalar multiplication would be to initialize an accumulator variable to $Q := \mathcal{O}$ and then k times adding P to Q. Essentially, it computes:

$$Q = \mathcal{O} + \underbrace{P + \ldots + P}_{k \text{ times}} \tag{2.6}$$

Because k is a large number, this computation will take forever to compute.[1] Fortunately, there are other addition chains that take less operations than the naive method.

[1] Note that even the discrete logarithm problem can be solved more efficiently.

2.2.1 Double-and-add

The traditional addition chain for computing the scalar multiplication of a point on an elliptic curve is the *double-and-add algorithm* (Algorithm 2.1). In this procedure, the key k is split into its bits $k_0, \ldots, k_{n-1}$ such that $k = k_0 2^0 + \ldots + k_{n-1} 2^{n-1}$ and every k_i is in $\{0, 1\}$.

Algorithm 2.1 Double-and-add algorithm

```
1:  function DoubleAndAdd(k, P)                    ▷ Compute [k]P
2:      R ← O
3:      for i from n − 1 down to 0 do
4:          R ← [2]R                                ▷ Doubling
5:          if k_i = 1 then
6:              R ← R + P                           ▷ Addition
7:          else
8:              R ← R + O                           ▷ Addition
9:          end if
10:     end for
11:     return R
12: end function
```

Because of reasons that will be explained in Section 2.4.3, for the algorithm to be secure, we cannot use any branches in our implementations that depend on the value of k. This means that in a regular implementation, we cannot skip the addition step if $k_i = 0$. That is why, on line 8, we add the neutral element, instead of doing nothing.

We can see that the cost of the double-and-add algorithm is $n \cdot$**double**$+n \cdot$**add**. For 256-bit curves, this results in 256 doublings and 256 additions.

2.2.2 Generalized Montgomery ladder

An alternative to the double-and-add algorithm is the *generalized Montgomery ladder* algorithm from [16], based on the original Montgomery ladder from [50]. Just as the double-and-add algorithm presented in Section 2.2.1, the generalized Montgomery ladder scans k from most significant bit to least significant bit.

The generalized Montgomery ladder works by using two different accumulators instead of one. R_0 always contains the "lower bound" value and R_1 contains the "upper bound". Where R is the actual result of the computation, and m is the amount of bits left to scan, at every ladder step it holds that

$$2^m R_0 \leq R < 2^m R_1.$$

After n iterations m is equal to 0, and thus R_0 will be exactly equal to R.

Algorithm 2.2 Generalized Montgomery ladder algorithm

function GeneralizedMontgomeryLadder(k, P) ▷ Compute $[k]P$
  $R_0 \leftarrow \mathcal{O}$
  $R_1 \leftarrow P$
  for i **from** $n-1$ **down to** 0 **do**
    if $k_i = 1$ **then**
      $R_0 \leftarrow R_0 + R_1$ ▷ Doubling
      $R_1 \leftarrow [2]R_1$ ▷ Addition
    else
      $R_1 \leftarrow R_0 + R_1$ ▷ Addition
      $R_0 \leftarrow [2]R_0$ ▷ Doubling
    end if
  end for
  return R_0
end function

As an example, an illustration of the generalized Montgomery ladder computation for $k = 13$ is shown in figure 2.3. Note that the binary representation of k is `1101`.

The cost of the generalized Montgomery ladder algorithm is equal to the cost of the double-and-add algorithm: 256 doublings and 256 additions for a k value of 256 bits. The main added benefit of the Montgomery ladder is that it is very efficient for curves of which x-only addition laws exist.[2]

2.2.3 Fixed-window method

Further performance gains can be achieved by adapting the double-and-add algorithm and looking at multiple bits of k in each loop iteration. This method is called the *fixed-window* method. k is split in multiple "digits" of size w, such that every element in $k' = \text{Windows}_w(k)$ is in $\{0, \ldots, 2^w - 1\}$, instead of just $\{0, 1\}$. For example, when we split the scalar $k = 45678$ into windows of size $w = 4$, we get $(k'_3, k'_2, k'_1, k'_0) = (11, 2, 6, 14)$. This is shown in figure 2.4.

We precompute a table of all the multiples of P up to $[2^w - 1]P$ and store them in a lookup table. We use the values from the table for the addition in the double-and-add algorithm. The resulting fixed-window algorithm is listed in Algorithm 2.3.

[2]The Montgomery ladder is also proclaimed to be "more side channel resistant" (see Section 2.4) than the double-and-add algorithm. While this is true, side-channel resistance is a property of implementations, not algorithms. Indeed, some programmers will have been saved by the intrinsic side channel resistant structure of the Montgomery ladder. Still, it is a subjective argument and not particularly relevant for this thesis.

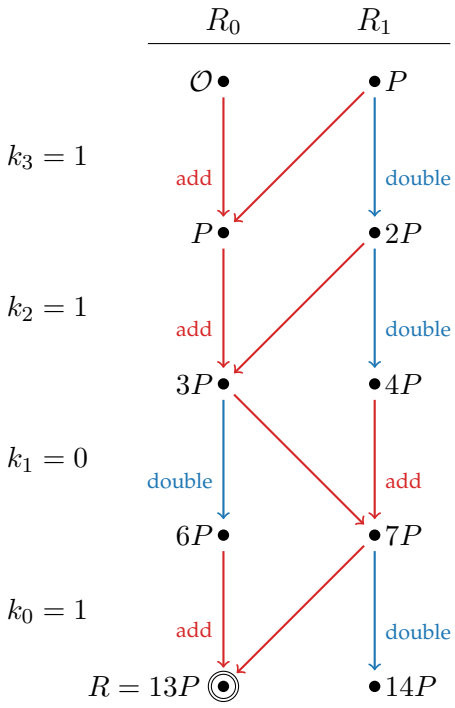

Figure 2.3: Generalized Montgomery ladder computation for $k = 13_{10} = 1101_2$.

$$k = \underbrace{\texttt{1011}}_{k'_3}\ \underbrace{\texttt{0010}}_{k'_2}\ \underbrace{\texttt{0110}}_{k'_1}\ \underbrace{\texttt{1110}}_{k'_0}$$

Figure 2.4: Splitting $k = 45678_{10}$ into widows of size $w = 4$.

In its loop, the fixed-window method computes only $\lceil \frac{n}{w} \rceil$ additions, instead of n. Although we have to additionally count the doublings and additions needed for computing the lookup table, we still end up with fewer operations. When using the fixed-window method, the amount of operations depends on the window size w:

$$\#\textbf{ops} = \underbrace{\left(2^{w-1} - 1\right)(\textbf{double} + \textbf{add})}_{\text{precomputation}} + \underbrace{n \cdot \textbf{double} + \left\lceil \frac{n}{w} \right\rceil \textbf{add}}_{\text{loop}} \tag{2.7}$$

When we take $n = 256$, as we did in the previous section, we can choose $w = 4$. Now (using Equation 2.7) we only need 262 doublings and 71 additions.

Algorithm 2.3 Fixed-window double-and-add

function FixedWindow(k, P) ▷ Compute $[k]P$
  $k' \leftarrow \text{Windows}_w(k)$
  Precompute $([2]P, \ldots, [2^w - 1]P)$
  $R \leftarrow \mathcal{O}$
  for i **from** $\frac{n}{w} - 1$ **down to** 0 **do**
    for j **from** 0 **to** $w - 1$ **do**
      $R \leftarrow [2]R$ ▷ w doublings
    end for
    if $k'_i \neq 0$ **then**
      $R \leftarrow R + [k'_i]P$ ▷ Addition
    else
      $R \leftarrow R + \mathcal{O}$ ▷ Addition
    end if
  end for
  return R
end function

2.2.4 Signed digit recoding

The last scalar multiplication optimization that is relevant for this thesis is to recode the window digits into a signed form. This method is called the *signed window method*. In this optimization, we use the feature that inverting a point on an elliptic curve is very cheap. To invert P, we only have to negate its y coordinate.

After encoding the windows into k' in the previous section, we will recode them into signed form. To compute $k'' = \text{RecodeSigned}(k')$, we first subtract w for every $k'_i \geq 2^{w-1}$ and add 1 to the next digit for every subtraction. The example k'_i digits from Section 2.2.3 (with $k = 45678$) recode to $(k''_4, k''_3, k''_2, k''_1, k''_0) = (1, -5, 2, 7, -2)$.

An illustration is shown in figure 2.5.

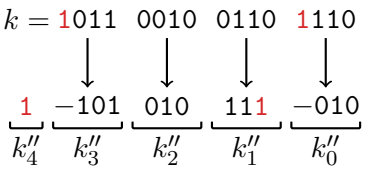

Figure 2.5: Splitting $k = 45678_{10}$ into windows of size $w = 4$ and recoding them into signed form. Carries from w-subtracted digits to the next are highlighted in red.

Algorithm 2.4 Signed double-and-add

function SignedFixedWindow(k, P) ▷ Compute $[k]P$
  $k' \leftarrow$ RecodeSigned(Windows$_w(k)$)
  Precompute $([2]P, \ldots, [2^{w-1}]P)$
  $R \leftarrow \mathcal{O}$
  for i **from** $\frac{n}{w} - 1$ **down to** 0 **do**
    for j **from** 0 **to** $w - 1$ **do**
      $R \leftarrow [2]R$ ▷ w doublings
    end for
    if $k'_i > 0$ **then**
      $R \leftarrow R + [k'_i]P$ ▷ Addition
    else if $k'_i < 0$ **then**
      $R \leftarrow R - [-k'_i]P$ ▷ Addition
    else
      $R \leftarrow R + \mathcal{O}$ ▷ Addition
    end if
  end for
  return R
end function

Through the example we see that, although w is still 4, the absolute value of all the recoded digits are at most three bits. In other words, our effective window size has been reduced to 3. After adapting Algorithm 2.3 to use the recoded k value, we end up with Algorithm 2.4. The operation count of the adapted algorithms is given by Equation 2.8.

$$\#\textbf{ops} = \underbrace{2^{w-2}\textbf{double} + \left(2^{w-2} - 1\right)\textbf{add}}_{\text{precomputation}} + \underbrace{n \cdot \textbf{double} + \left\lceil \frac{n+1}{w} \right\rceil \textbf{add}}_{\text{loop}} \quad (2.8)$$

The gain achieved by recoding into signed form is only small. The signed method eliminates about half of the precomputation operations. For a k of 256 bits and $w = 5$, this results in 264 doublings and 59 additions.

2.3 Basic big-number arithmetic

2.3.1 Introduction

In ECC implementations we are often dealing with integers of 256 bits or larger. This thesis presents an implementation for a 64-bit system, so these integers are too large to fit into single registers. We have to split these numbers into multiple parts, which we call *limbs*. More formally, a number z is represented by a list of integers $z_0, \ldots, z_{n-1}$, where $z = z_{n-1}b^{n-1} + \ldots + z_0 b^0$. In this expression, every z_i natively fits into a register; n is the amount of limbs; and b is the *radix*, i.e. the factor between two adjacent limbs.

Let us sketch an intuitive example. Let us choose radix $b = 100$ and $n = 4$. In this example, let us say no limb can have a value larger than 99. Then we can store values up to 100.000.000.

$$z = \underbrace{12}_{z_3}\ \underbrace{34}_{z_2}\ \underbrace{56}_{z_1}\ \underbrace{78}_{z_0}$$

Figure 2.6: Representation of $z = 12.345.678$ using four limbs, where each limb $z_i < 100$.

2.3.2 Addition and subtraction

It is easy to implement addition and subtraction. To compute $x + y = z$ we add x and y limb-wise. However, the sum of some x_i and y_i may be larger than b. In this case we have to *carry* the new digits to the next limb[3].

$$\begin{array}{rcccc} x = & 12 & 34 & 56 & 78 \\ y = & 0 & 87 & 20 & 72 \\ \hline & 12 & 121 & 76 & 150 \\ \hline z = & 13 & 21 & 77 & 50 \end{array} \quad \begin{array}{l} \\ \\ + \\ \text{carry} \\ \\ \end{array}$$

Figure 2.7: Addition and single bit carry of two big integers.

Subtraction is applied roughly the same as addition, but instead of carrying an overflowed bit, we have to *borrow* an underflowed bit. Instead of adding this bit to the next limb, we subtract the bit from the next limb. Note that this carry bit always propagates from the least significant limb to the most significant limb.

[3] In the literature, the carry chain is sometimes called *coefficient reduction*.

2.3.3 Redundant representation

A sometimes (computationally) expensive element of the representation from the previous sections is the propagation of the carry bits during *every* addition and subtraction. Because carry chains execute over all limbs in a sequential order, the CPU cannot optimize the code by parallelizing instructions. To overcome this, implementors often choose to leave some headroom in each limb, such that multiple additions can be computed before a limb overflows.

This representation is called *redundant representation*[4]. In this representation, b is not the maximum value that is possible, e.g. 2^{64} for 64-bit systems. Instead we choose b some other power of two, to have $64 - \log_2 b$ bits left before overflowing.

2.3.4 Multiplication

The traditional method of multiplying two integers is—just like addition and subtraction—very intuitive. To multiply two integers $x \cdot y = z$ we compute for each destination limb z_k:

$$z_k = \sum_{i+j=k} x_i \cdot y_j \tag{2.9}$$

Equation 2.9 assumes that no overflows will occur. We can achieve this by choosing a proper radix.

We see that we must twice iterate through each of the input operands. Indeed, the complexity of the traditional multiplication method is $\mathcal{O}(n^2)$, where n is the amount of limbs. Therefore, multiplying large numbers is a lot more expensive than adding or subtracting them.

In the general case, squaring an integer is not very different from multiplying two different integers. The complexity is still $\mathcal{O}(n^2)$, but because $x_i y_j = x_j y_i$ we can eliminate almost half of the inner multiply operations.

2.3.5 Karatsuba multiplication

For getting a faster multiplication algorithm we can try using *Karatsuba multiplication* [39]. This reduces the complexity of the multiplication to $\mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.58})$. To compute the product of two integers using Karatsuba's method, we split each operand into its upper and lower parts: So the operand $x = \beta \cdot x_H + x_L$ where both x_H and x_L are smaller than the chosen base β. Now the product $z = \beta^2 z_H + \beta z_M + z_L$ is computed using the steps described by algorithm 2.5.

[4] Sometimes called *sparse representation*.

Algorithm 2.5 Karatsuba multiplication algorithm

1: **procedure** KARATSUBA(x_H, x_L, y_H, y_L) ▷ Compute $z = x \cdot y$
2: $z_H \leftarrow x_H \cdot y_H$
3: $z_L \leftarrow x_L \cdot y_L$
4: $z_M \leftarrow (x_H + x_L) \cdot (y_H + y_L) - z_H - z_L$
5: **end procedure**

Algorithm 2.5 uses only three inner multiplications, instead of the four that we would expect from equation 2.9. With one level of Karatsuba multiplications, we spare one quarter of the multiplication operations that would be needed otherwise. We can improve the complexity even further by implementing the inner multiplications in lines 2–4 recursively using the same Karatsuba trick. However, due to the amount of addition and subtraction operations added by the algorithm, adding more Karatsuba layers will not always achieve better performance.

2.4 Side-channel attacks

Any algorithm can be viewed simply as a mapping from an input to an output. The output contains only that what is inserted after the `return` statement. This intuition suggests that an algorithm does not disclose *any information* except for its output. However, while this idea holds in an abstract sense, it is simply not true for algorithms implemented on real hardware. In reality, implementations have physical properties that may depend on their inputs. Basic examples of these kinds of properties are runtime, memory usage, power consumption, etc.

These "information channels", through which we can distill pieces of information are called *side channels*. In a formal sense, side channels are unintended features of an implementation, through which an attacker can gain information about a secret input. Over the years, different types of side-channel attacks have been devised. This section will restrict itself to describing timing attacks, which are relevant for the CPU's that are targeted in this thesis.

2.4.1 Regular timing attacks

Timing attacks are a type of side-channel attack that are easy to execute. They were introduced by Kocher in 1996 [42]. Timing attacks measure the runtime of an implementation or parts of that implementation.

As an example, let's look again at the double-and-add algorithm from Subsection 2.2.1. If we look at line 9 from the the double-and-add algorithm

(Alg. 2.1 on page 15), we see that $Q \leftarrow Q + \mathcal{O}$ is computed whenever $k_i = 0$. An implementor could choose to omit this addition, because it is a no-op. Now the runtime of the algorithm depends directly on the key k. An attacker can measure this and easily figure out the Hamming weight of the key.

2.4.2 Cache timing attacks

A more modern type of attack, based on the timing attack from the previous section, is the *cache-timing attack*. These attacks exploit the cache latency of a CPU to figure out which memory regions are accessed by the cryptographic algorithms. Using this information, an attacker can deduce which code paths are being taken and which data is being accessed.

In most cases, cache-timing attacks run on the same processor as the algorithm that is being executed, which allows them to actively manipulate the contents of the CPU cache. The ability to manipulate the cache is the reason that cache-timing attacks are generally more powerful than regular timing attacks. There exist different types of strategies to perform cache timing attack, e.g., Prime+Probe [64], Cache+Evict [64], Flush+Reload [65].

The most basic attack is the Prime+Probe attack: Assume that we share the read-only memory region that holds the lookup tables that are used by some scalar-multiplication algorithm. First the attacker makes sure no RAM is loaded into the cache. Then they wait for a run of the algorithm to happen. After the crypto-algorithm is done, it will have read from a selection of memory addresses. Now the attacker reads every memory address and measures the time it takes retrieve it. By the time it takes to retrieve a value, they know whether a table entry has been loaded into the cache during the encryption. Using this information, the attacker can deduce vital information about the internal encryption values and the encryption key.

Note that we do not even need to know every bit of the key. Every bit that is correctly recovered using the attack reduces the security of the encryption by a factor 2. If we have enough key bits we can brute force the rest. When dealing with elliptic-curve cryptography, we can even employ a lattice attack to recover the bits using an efficient lattice reduction technique [5, 55] (e.g. LLL [44] or BKZ [60]).

2.4.3 Preventing side channels

Side-channels attacks are powerful and have been used to extract keys from a lot of cryptographic software over the years. A survey from 2018 by Ge, Yarom, Cock, and Heiser shows dozens of works describing side channel attacks on common cryptographic libraries [28]. It has been determined that these attacks are practical, not just academic [45].

Multiple methods exist to prevent side-channel attacks in cryptographic software. The first and foremost method is to write all cryptographic software in *constant time*[5]. As the term suggests, software that is *constant time* is implemented in such a way that the runtime of an execution does not depend on any secret inputs. This can be achieved by using standard programming tricks. For example, the programmer can replace an `if-else` expression by a piece of code that executes both blocks and selects the correct result. Listings 1 and 2 show an example of this in C.

Listing 1 Insecure code snippet.

```
// In this snippet, key_bit is some bit in the secret key (0 or 1).
int32_t result;
if (key_bit) {
    result = a();
} else {
    result = b();
}
```

In listing 2, we multiply the results by the `key_bit` value and its negation. Because the `key_bit` value is `1` or `0`, this keeps the correct value and sets the other to `0`. Both values are added to make `result`.

Listing 2 Secure version of listing 1.

```
int32_t result = (a() * key_bit) + (b() * !key_bit);
```

In practice we will not use $*$ and $+$. For efficiency reasons we rather expand the `key_bit` value to a bitmask, i.e. `0x00000000` or `0xFFFFFFFF` for 32-bit values. Then we use a bitwise AND operation to select the correct value and a bitwise OR operation to combine them into the result. An updated version of Listing 2—using bitwise operations—is shown in Listing 3.

Listing 3 More efficient version of Listing 2.

```
// Intel uses two's complement, so we can use negation here.
const uint32_t mask = -(int32_t)key_bit;
int32_t result = (a() & mask) | (b() & ~mask);
```

[5]We could write "constant time" *and* "constant lookup" here, to explicitly include cache timing attacks in our argument. However—because of the CPU cache and other similar effects—lookups from variable addresses are *always* variable in their execution time. Therefore, "constant time" implies "constant lookup". In this thesis, whenever the term "constant time" is used, both properties are implied.

Chapter 3

Choosing a curve

Recall from Chapter 1 that the goal of this thesis is to benchmark the benefit the of x-only coordinate ladder of Montgomery curves—in particular Curve25519 [8]—relative to the more complex complete addition formulas for Weierstrass curves.

Therefore we will choose a prime-order curve that is similar to Bernstein's Curve25519. Aside from being able to make a good comparison, we can furthermore build on some of the optimizations that have been developed specifically for Curve25519 [12, 21, 25, 27]. Our second priority will be to choose properties that are "straightforward", i.e. properties that are often found in other standardized curves. In general, we try to find an answer to the question: What would Curve25519 have looked like, had it been a prime-order curve?

The curve that we have eventually chosen was proposed by P. Barreto in May 2017 [3]. The nameless curve is defined over $\mathbb{F}_{2^{255}-19}$. It is described by equation 3.1 and a suitable generator is $G = (-7, 114)$. The Sage script that was used to verify the curve is listed in Appendix B.

$$E : y^2 = x^3 - 3x + 13318 \tag{3.1}$$

As mentioned in the introduction of this chapter, using the same field as Curve25519 allows us to build upon some of the computational tricks that have been developed for this field. Moreover, because many implementations for Curve25519 are released into the public domain, we can even copy and adapt portions of previous implementations.

As is the point of this thesis, the chosen curve is a prime-order curve. Its order is given by

$$N = \ell = 2^{255} + 32561065938887340030620144057166140515 5\,. \tag{3.2}$$

3.1 Choice of a

A reasonable alternative to Barreto's curve is the Weierstrass curve defined over $\mathbb{F}_{2^{255}-19}$ described by curve equation 3.3.

$$E : y^2 = x^3 + x + 26606 \tag{3.3}$$

Its difference with Barreto's curve is that $a = 1$, instead of $a = -3$. The first reason we choose $a = -3$ instead of 1 is that—depending on the addition formulas that are used—it results in more efficient curve arithmetic [17]. Indeed, the Renes-Costello-Batina complete addition formulas have a specialized case for $a = -3$ [57, Section 3.2]. The second reason is that various cryptographic standards have adopted these kinds of curves [26, 18, 52, 48, 20]. Our results will apply to more other commonly used curves if we mimic the standards.

3.2 Choice of b

The first prime order curve that we find for $q = 2^{255} - 19$ and $a = -3$ is the curve for which $b = 101$. However, in order to mitigate invalid curve attacks in some instances, we have opted for a curve of which the quadratic twist is also of prime order. The first curve of which the twist is also of prime order is the curve with $b = 13318$.

These mitigations will take effect when using x-only scalar-multiplication ladders. At this point in time, only [61] describes x-only exceptionless addition formulas for general Weierstrass curves. When other addition formulas are used, point validation is almost always mandatory.[1]

Other than choosing the value of b, the curve is left unchanged. In other words, we can choose this property "for free". With this in mind, it makes no sense *not* to choose this property.

3.3 Cofactor security

Bad implementations of cryptographic protocols using curves with a cofactor $h > 1$ can be vulnerable to *small-subgroup attacks* [23]. In a small-subgroup attack, the attacker uses the nontrivial cofactor to force a point P into a subgroup that is smaller than ℓ. If h is small—which it likely is, because otherwise N would be smooth—the DLP becomes very easy to solve in this

[1] An exception to this rule are the Brier-Joye addition formulas [16]. In the case of the Brier-Joye formulas, an implementor can choose to skip the point-validation step. However, if they do, the must reduce the secret scalar modulo the curve order *and* the twist order, before starting the scalar multiplication. Otherwise, exceptions may occur when computing scalar multiplications of an invalid point.

subgroup. An attacker can use this property to generate signatures that verify correctly by badly implemented protocols.

To mitigate against this attack, implementors must check if $[\ell]P = \mathcal{O}$ during the point validation stage. If this check does not pass, the algorithm must reject the point, as it is invalid.

Another approach to prevent the small-subgroup attack is to disallow users to sign arbitrary messages, and only allowing them to sign *hashes* of messages [9]. This prevents an attacker from generating malicious signatures, because they must brute force messages until they find a hash that maps to a point of order smaller than ℓ. However, a curve's cofactor is small (generally, $h \leq 8$), this means the amount of small-subgroup points on the curve is small. Therefore, the probability of hitting a small-subgroup point becomes negligible, which makes the attack infeasible. This mitigation works mainly for traditional signatures on messages, such as Schnorr signatures [59] and ECDSA [38]. It does not work for protocols where the signed data can be arbitrarily chosen by an attacker, for example in ring signatures [58].

Barreto's curve is of prime order, i.e. it has a trivial cofactor. It is therefore immune to small-subgroup attacks. By completely eliminating this attack, these curves are preferable above non-prime order curves, when constructing more complex cryptographic protocols.

3.4 Indistinguishability

Lastly, a feature that may be required from an elliptic curve is that its points can be represented as uniformly random strings.[2] To achieve this, we need a suitable point-compression algorithm for our curve. A popular option for Edwards curves is Elligator, which implements a system wherein points are compressed into uniformly random strings [10]. Elligator Squared is a less known alternative for prime order curves [63]. It can be used to compress points into indistinguishable strings. Elligator Squared works for *all* prime order curves, including Barreto's curve.

[2] This is useful when building censorship-circumvention tools.

Chapter 4

Field arithmetic

Before we can evaluate the practical performance of the Renes-Costello-Batina formulas from [57] compared to the Curve25519 formulas, we have to select a reference point first.

In this thesis we will compare the performance of our implementation to the performance of the *Sandy2x* [21] implementation. Sandy2x is an optimized implementation of Curve25519 scalar multiplication, targeted to Intel's Sandy Bridge microarchitecture.

The exact part of Sandy2x that we will try to beat is its *variable-point* scalar-multiplication algorithm. In variable-point scalar multiplication the multiplied point is different for each multiplication. In contrast, in *base-point* scalar multiplication the multiplied point is fixed. In this case the implementor can precompute a lot values at compile-time.

To get an implementation that is fast and side-channel resistant, we cannot just use a generic big-integer arithmetic library. Often, these libraries are not side-channel resistant. Furthermore, their gain in general applicability is paid in performance loss. Instead, we write the core of the algorithm—including the finite-field arithmetic—in (NASM) assembly language.

To speed up our computations we use AVX to compute some finite-field arithmetic from the addition formulas in a vectorized manner. In other words, some arithmetic operations will be computed in parallel. This chapter only describes how the finite-field arithmetic is implemented. Later—in Chapter 5—we will describe how the arithmetic routines are used in the addition formulas.

In our research we have studied two different approaches to implement the finite-field arithmetic. The first approach uses $2\times$ vectorized 64-bit integers in radix $2^{25.5}$. This approach was used by [12] and it is the method that

Sandy2x uses. The second approach uses $4\times$ vectorized double-precision floating-point numbers in radix $2^{21.25}$. Both methods will be described in depth in Sections 4.2 and 4.3, respectively. In the end, this thesis has chosen to use the floating-point method for the actual implementation.

4.1 The Sandy Bridge microarchitecture

The implementation described in this thesis targets Intel's *Sandy Bridge* microarchitecture. Sandy Bridge is a *superscalar* processor, meaning that it can execute multiple instruction pipelines in parallel. Furthermore, it is the first processor from Intel that features *Advanced Vector Extensions* (AVX). AVX is an instruction set that features various useful SIMD instructions. In particular, the architecture has instructions for 128-bit-wide packed 64-bit integer arithmetic and 256-bit-wide packed double floating-point arithmetic.

To understand what the performance of low-level algorithms—like those described later in this chapter—is composed of, we must first understand some of the components that the CPU provides. This subsection summarizes Section 2.3 from Intel's Optimization Reference Manual [37], particularly the elements that are relevant for this thesis.

The CPU's pipeline is split up in two parts: The *front end* and the *back end*[1]. The front end is differentiated from the back end in that it executes its work *in-order*, while the back end executes its work *out-of-order*[2]. The most relevant group of components in the front end are the instruction *decoders*. The decoders—Sandy Bridge has four—decode the instructions in the code into *micro operations* (µops), which are the basic operations that the CPU will execute. For instance, a `push rax` instruction will be decoded into one `mov [rsp], rax` µop and one `sub rsp, 8` µop. After decoding the instructions, the *renamer* will map the logical registers in the code to physical registers on the CPU. When a µop is *ready*, it will enter the scheduler.

The *scheduler* is part of the back end. Every cycle, the scheduler selects the µops that are ready, and dispatches them to one of the six CPU *ports* for execution. These ports perform the actual computation, and write the result back to the physical registers. Each port implements a different set of functionality, so different kinds of µops are dispatched to different ports. Furthermore, the scheduler can dispatch multiple µops per cycle, which is how the CPU executes instructions in parallel.

Most µops take a couple of cycles before their result is written back and they retire. This amount of cycles is called the *latency* of an instruction. The CPU ports execute their µops in a pipeline, so a port can start executing

[1] These are sometimes called the *fused domain* and the *unfused domain*.
[2] In the literature and manuals, this is often abbreviated as *OoO*.

new µops while other µops on that port have not finished yet. Because some functionality is implemented in multiple different ports—for example, memory loads are implemented on both port 2 and port 3—the *throughput* of an instruction can be more than one instruction per cycle.

In some sections in this thesis, an assembly code listing is provided for illustrational purposes. Appendix A lists the µops, latency and reciprocal throughput for the instructions that are relevant for this thesis. Through the coming sections, this list may help in understanding how some of the code listings translate into cycle counts.

4.2 Radix $2^{25.5}$ in integer registers

In the radix $2^{25.5}$-representation, a field element $f \in \mathbb{F}_{2^{255}-19}$ is put into 10 64-bit registers and is operated on using *unsigned*[3] operations. Each limb's base is defined by $b_i = 2^{\lceil 25.5i \rceil}$, i.e. every base is rounded up to the next power of 2. In essence, f is represented as

$$f = f_0 + 2^{26} f_1 + 2^{51} f_2 + 2^{77} f_3 + 2^{102} f_4 + 2^{128} f_5 + 2^{153} f_6 + 2^{179} f_7 + 2^{204} f_8 + 2^{230} f_9.$$

Before adding some f and g into h, we must first ensure that h will fit in its limbs. If this is not the case, we must carry one or both of the operands before computing their sum. After ensuring that no overflow will occur, we compute by limb-wise addition: $h_i = f_i + g_i$.

Subtraction works almost the same as addition. However, because we are using unsigned integer limbs, we must also ensure that no underflows can occur in the operation. In other words, when computing $h = f - g$, we must guarantee that $f_i \geq g_i$ for each limb i. An easy method to ensure this is to add a multiple of p $(= 2^{255} - 19)$ to f before computing the subtraction. The multiple n is always chosen such that $f'_i = f_i + (np)_i \geq g_i$.

4.2.1 Carry

When f nears overflowing its registers, we must execute a carry chain to reduce the limb's values to occupy only the bottom part of their registers.

[3] When we use *signed* limbs, we need an instruction that shifts packed quadwords to the right, while shifting in sign bits, in for our carry chain in Section 4.2.1. Such an arithmetic shift operation—which would be called `vpsraq`—has never been implemented for the Sandy Bridge microarchitecture. Indeed, the first implementation of the `vpsraq`-instruction was in AVX-512, in the Knight's Landing and Skylake microarchitectures.

Instead, we only have the logical shift instruction `vpsrlq` operation, which we use in Listing 4. However, for our purposes, we can only use this instruction for unsigned limbs.

Define, for this carry chain, the following functions:

$$\text{bottom}(f_i) = f_i \mod (b_{i+1}/b_i) \tag{4.1}$$

$$\text{top}(f_i) = f_i - \text{bottom}(f_i) \tag{4.2}$$

These functions respectively select the bottom part and the top part from a limb. We use them to compute a step in the carry chain. Algorithm 4.1 shows the operations that are used to carry from one limb to another.

Algorithm 4.1 Implementation of a carry step from limb f_a to f_{a+1}.

1: **procedure** CarryStep ▷ Perform a single carry step
2: $f_{a+1} \leftarrow f_{a+1} + \text{top}(f_a) \cdot (b_a/b_{a+1})$
3: $f_a \leftarrow \text{bottom}(f_a)$
4: **end procedure**

It is clear that an exception occurs when limb 9 is carried back to limb 0, because we are passing the field modulus p. To carry f_9 to f_0, let us first carry f_9 to the *virtual* limb f_{10} in the same manner as we have done with the others. We see that the value represented by f_{10} in f is $2^{255} f_{10} = (p+19) f_{10} \equiv 19 f_{10} \pmod{p}$. So to carry f_9 to f_0 we need only to multiply the carried part by 19 before adding it to f_0.

When we implement a $2\times$ vectorized carry chain for the Sandy Bridge microarchitecture, we only need three instructions per carry step. As an example, the instruction listing for the carry step from f_0 to f_1 is shown in Listing 4.

Listing 4 Single carry step for radix $2^{25.5}$.

```
1 ; Inputs:
2 ;    - xmm0: f0
3 ;    - [rel .MASK26]: times 2 dq 0x3FFFFFF
4 ; Outputs:
5 ;    - xmm0: f0
6 ;    - xmm1: f1
7 vpsrlq xmm15, xmm0, 26                    ; t ← top(f0) · 2^-26
8 vpaddq xmm1, xmm1, xmm15                  ; f1 ← f1 + t
9 vpand xmm0, xmm0, oword [rel .MASK26]     ; f0 ← bottom(f0)
```

An efficient way to implement the multiplication by 19 is to compute $2^4 c + (c + c + c)$. For this routine, we need one additional `vpsllq` and three additional `vpaddq` instructions.

The easiest method to implement the carry chain is to go through all limbs sequentially, as depicted in Figure 4.1.

$$f_0 \to f_1 \to f_2 \to f_3 \to f_4 \to f_5 \to f_6 \to f_7 \to f_8 \to f_9 \to f_0 \to f_1$$

Figure 4.1: Sequential 10-limb carry chain.

In Listing 4, all three instructions `vpsrlq`, `vpaddq` and `vpand` can be scheduled on ports 0, 1 and 5 simultaneously. Furthermore, the latency of the carry step is two cycles. This means that, with this strategy, a data hazard occurs in every step of the carry ripple, when the `vpaddq` instruction (from Listing 4) has to wait an additional cycle before the value in `xmm15` is ready. To solve this problem, [22] introduces an *interleaved carry chain*, which is also used by Sandy2x. It is displayed in Figure 4.2.

$$\begin{aligned} &f_0 \to f_1 \to f_2 \to f_3 \to f_4 \to f_5 \to f_6, \\ &f_5 \to f_6 \to f_7 \to f_8 \to f_9 \to f_0 \to f_1 \end{aligned}$$

Figure 4.2: Interleaved 10-limb carry chain.

While the interleaved chain is more expensive in the amount of steps, the two separate carry chains are computed in parallel, which almost halves the routine's latency. Because the interleaved carry chain has 12 carry steps, each with a reciprocal throughput of 1cc, we expect the lower-bound of the reciprocal throughput to be 12cc or 6cc/op[4].

4.2.2 Multiplication

Sandy2x uses the multiplication routine as described in [12]. It uses "schoolbook" multiplication, which has already been described in Subsection 2.3.4. Just as the carry chain from the previous sections, the multiplication routine is implemented $2\times$ parallel, using vectorized registers.

It manages modular arithmetic in the same manner as the carry chain from the previous section: Any product that is written into a limb h_i where $i \geq 10$ is multiplied by 19, and then added into h_{i-10}. If we use Equation 2.9 and write out the complete multiplication routine, the formulas for $h = f \cdot g$ become

[4] cycles per operation

$$
\begin{aligned}
h_0 &= f_0g_0 + 38f_1g_{11} + 19f_2g_{10} + 38f_3g_9 + 19f_4g_8 + 38f_5g_7 + 19f_6g_6 + 38f_7g_5 + 19f_8g_4 + 38f_9g_3,\\
h_1 &= f_0g_1 + f_1g_0 + 19f_2g_{11} + 19f_3g_{10} + 19f_4g_9 + 19f_5g_8 + 19f_6g_7 + 19f_7g_6 + 19f_8g_5 + 19f_9g_4,\\
h_2 &= f_0g_2 + 2f_1g_1 + f_2g_0 + 38f_3g_{11} + 19f_4g_{10} + 38f_5g_9 + 19f_6g_8 + 38f_7g_7 + 19f_8g_6 + 38f_9g_5,\\
h_3 &= f_0g_3 + f_1g_2 + f_2g_1 + f_3g_0 + 19f_4g_{11} + 19f_5g_{10} + 19f_6g_9 + 19f_7g_8 + 19f_8g_7 + 19f_9g_6,\\
h_4 &= f_0g_4 + 2f_1g_3 + f_2g_2 + 2f_3g_1 + f_4g_0 + 38f_5g_{11} + 19f_6g_{10} + 38f_7g_9 + 19f_8g_8 + 38f_9g_7,\\
h_5 &= f_0g_5 + f_1g_4 + f_2g_3 + f_3g_2 + f_4g_1 + f_5g_0 + 19f_6g_{11} + 19f_7g_{10} + 19f_8g_9 + 19f_9g_8,\\
h_6 &= f_0g_6 + 2f_1g_5 + f_2g_4 + 2f_3g_3 + f_4g_2 + 2f_5g_1 + f_6g_0 + 38f_7g_{11} + 19f_8g_{10} + 38f_9g_9,\\
h_7 &= f_0g_7 + f_1g_6 + f_2g_5 + f_3g_4 + f_4g_3 + f_5g_2 + f_6g_1 + f_7g_0 + 19f_8g_{11} + 19f_9g_{10},\\
h_8 &= f_0g_8 + 2f_1g_7 + f_2g_6 + 2f_3g_5 + f_4g_4 + 2f_5g_3 + f_6g_2 + 2f_7g_1 + f_8g_0 + 38f_9g_{11},\\
h_9 &= f_0g_9 + f_1g_8 + f_2g_7 + f_3g_6 + f_4g_5 + f_5g_4 + f_6g_3 + f_7g_2 + f_8g_1 + f_9g_0.
\end{aligned}
$$

In this implementation, 9 `vpmuludq` instructions are used to precompute a list of $19g_0, \ldots, 19g_9$ values. Furthermore, the values $2f_1, 2f_3, \ldots, 2f_9$ are precomputed using 5 `vpaddq` instructions. This results in a multiplication algorithm that uses 109 `vpmuludq` and 95 `vpaddq` instructions.

`vpaddq` can be executed on ports 1 and 5, while `vpmuludq` can only be executed on port 0. Therefore, we expect port 0 to be the bottleneck during the execution of the multiplication algorithm. This is also a good reason to implement the doubling of the f_i values using additions instead of shifts, as using the `vpshlq` instruction would increase the pressure on port 0 even more. Because in this computation we are using $2\times$ vectorized registers, we expect the lower bound cost of this algorithm to be 54.5cc/op.

We can adapt the multiplication routine to get an algorithm for squaring numbers by substituting g for f in the formulas above. This eliminates an inner multiplication for every case where $f_ig_j = f_jg_i$. After this optimization, the squaring algorithm requires only 64 `vpmuludq` instructions, which is a speedup of about 60 percent.

4.3 Radix $2^{21.25}$ in floating-point registers

The 10-limb approach from Section 4.2 provides us with arithmetic that can be $2\times$ vectorized using the instructions provided by the Intel's SSE2 instructions, which are implemented in Sandy Bridge. Apart from featuring $2\times$ vectorized integer instructions, the Sandy Bridge microarchitecture adds instructions for $4\times$ vectorized double-precision floating-point arithmetic. When using this instruction set, we can implement the finite-field arithmetic $4\times$ parallel, instead of $2\times$, yielding even better performance.

4.3.1 Floating points

Before introducing radix-$2^{21.25}$ representation, let us first briefly review the format of the floating-point numbers used in this section. Double-precision

floating points (or *doubles* for short) were introduced in [35][5].

A double consists of 52 *mantissa* m bits $(m_1, \ldots, m_{52})$ which encode the fraction of the number, 11 exponent e bits and a sign bit s. Every floating-point number is *normalized*, which means that, for some number $z = m \cdot 2^k$, the real exponent k is always selected such that $1 \leq z \cdot 2^{-k} < 2$. Because of this, the first binary digit of z is always 1, and does not need to be stored. Therefore, we have an extra "virtual" mantissa bit, which I will call m_0. With m_0 the actual precision of the mantissa is 53 bits.

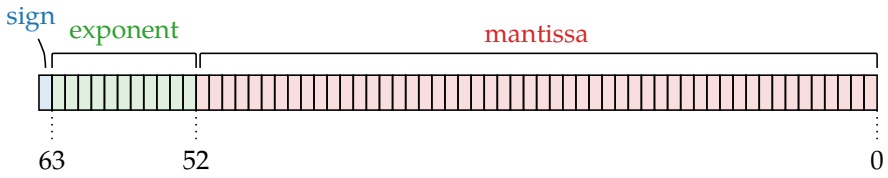

Figure 4.3: Double-precision floating-point format.

To be able to encode numbers with negative exponents, e is offset from zero by 1023, i.e. $e = k + 1023$. Furthermore, the exponent values 0 and 2047 are reserved to encoding special values. For example, a value with $e = 0$ encodes signed zero. Using this information, we see that the actual value of a double is given by

$$z = (-1)^s \left(\sum_{i=0}^{52} 2^{-i} m_i \right) 2^{e-1023}, \text{ where } m_0 = 1.$$

Using doubles with 53-bit precision, we can simulate integer registers of 53 bits. To guarantee that no rounding errors occur in the underlying floating-point arithmetic, we use the same strategy as in Section 4.2: ensure that limb values always fit in their registers, or use a carry chain to reduce the amount of bits in each register before performing operations that might overflow.

Building on this idea, [8] recommends—but does not implement—*radix* $2^{21.25}$, based on the arithmetic described in [7]. In this representation, f is put into 12 signed double-precision floating-point registers. Doubles already store their base in the exponent, which is large enough for our purposes. Therefore, we do not have to introduce a base for f's limbs. Indeed, the value is computed by just computing the sum of the limbs:

$$f = \sum_{i=0}^{11} f_i$$

[5] However, for a more general introduction to floats I instead recommend reading Goldberg's guide on floating-point arithmetic [30, Section 2].

However, because we cannot store values with more than 53 bits of mantissa precision in a register, we must restrict each limb's value. Where $b_i = 2^{\lceil 21.25i \rceil}$, for each limb f_i, it must hold that

1. $f_i \in b_i\mathbb{Z}$, and
2. $|f_i| < 2^{53} b_i$.

4.3.2 Carry

As in Subsection 4.2.1, we define the following functions:

$$\text{bottom}(f_i) = f_i \mod b_{i+1} \tag{4.3}$$

$$\text{top}(f_i) = f_i - \text{bottom}(f_i) \tag{4.4}$$

The Intel architecture supports no native modulo operation on floating points. Instead we compute $\text{top}(f_i)$ by subsequently adding and subtracting a large constant $c_i = 3 \cdot 2^{51} b_{i+1}$, forcing the processor to throw away the lower mantissa bits. The following reasoning shows why this results in the correct carried value.[6]

Consider that the we have a limb f_i, for which $-2^{51} b_{i+1} \leq f_i \leq 2^{51} b_{i+1}$. First we add c_i to f_i yielding $z = f_i + 3 \cdot 2^{51} b_{i+1}$ and see that $2^{52} b_{i+1} \leq z \leq 2^{53} b_{i+1}$. After the value z is rounded by the CPU, we get z'.

From the bounds of z, we see that leftmost (virtual) mantissa bit $m_{z',0}$ of the rounded value z' will represent a power of two that is at least $2^{52} b_{i+1}$. This means that the rightmost bit $m_{z',52}$ represents at least b_{i+1}, i.e. we conclude that $z' \in b_{i+1}\mathbb{Z}$. We now compute $\text{top}(f_i) = z' - c_i$. Because both $z', c_i \in b_{i+1}\mathbb{Z}$, we know that $\text{top}(f_i) \in b_{i+1}\mathbb{Z}$. A visual depiction of these two steps is shown in Figure 4.4.

After having computed $\text{top}(f_i)$, computing the bottom part of f_i is easily obtained by computing $\text{bottom}(f_i) = f_i - \text{top}(f_i)$.

To carry f_{11} over to f_0—just as with radix $2^{25.5}$—we first imagine a virtual limb f_{12}. By definition, $b_{12} = 2^{255}$ and thus $f_{12} \in 2^{255}\mathbb{Z}$. Observe that $2^{255} \equiv 19 \Rightarrow 1 \equiv 19 \cdot 2^{-255} \pmod{p}$. So to carry f_{12} to f_0, we just have to multiply by $19 \cdot 2^{-255}$. We see that the limb restrictions mentioned in the introduction hold for the resulting value:

1. Because $f_{12} \in 2^{255}\mathbb{Z}$, we know that $19 \cdot 2^{-255} f_{12} \in 19\mathbb{Z}$. $b_0 = 1 \Rightarrow 19 \cdot 2^{-255} f_{12} \in b_0\mathbb{Z}$.
2. By assumption, $|f_{11}| < 2^{53} b_{11} = 2^{287}$. So the virtual limb $|f_{12}| < 2^{287}$. Then, the carried value $19 \cdot 2^{-255} f_{12} < 19 \cdot 2^{32} < 2^{53} b_0$.

[6] This argumentation is based on Theorem 2.4 from [7] and the proof that is included with it.

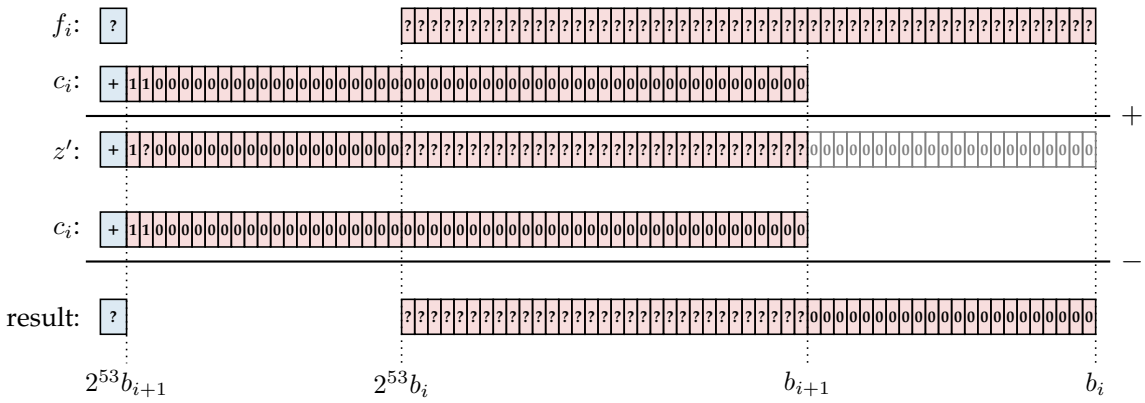

Figure 4.4: Visual representation of top using floating points with 53 bits of mantissa precision. The sign of the number is shown in blue and the mantissa is shown in red. We see that after adding c_i, the floating-point precision "window" shifts to the left. During this shift, the bits on the bottom of z are lost due to rounding (visualized in gray). After subtracting c_i from z', the window is shifted back and we get the resulting $\mathrm{top}(f_i)$, which consists of only the most significant bits of f_i.

In code, a single carry step needs 5 instructions. In Listing 5, the $4\times$-vectorized carry step from f_0 to f_1 is shown. The multiplication of f_{12} by $19 \cdot 2^{-255}$ is implemented using a regular `vmulpd ymmX, [rel .reduceconstant]` instruction.

Listing 5 Single carry step for radix $2^{21.25}$ from limb 0 to limb 1.

```
1  vmovapd ymm14, yword [rel .c_0]          ; load c_0
2  vaddpd ymm15, ymm0, ymm14                ; z' ← round(f_0 + c_0)
3  vsubpd ymm15, ymm15, ymm14               ; t ← round(z' − c_0)
4  vaddpd ymm1, ymm1, ymm15                 ; f_1 ← round(f_1 + t)
5  vsubpd ymm0, ymm0, ymm15                 ; f_0 ← round(f_0 − t)
```

In contrast to the code in Listing 4, in Listing 5 all arithmetic instructions execute on port 1. Furthermore, every `v{add,sub}pd` instruction has a latency of 3 cycles. Consequently, the latency of one carry step is the sum of the latencies of instructions 2 – 4, i.e. the latency is $3 \cdot 3 = 9$cc. Still, the reciprocal throughput is only 4cc.

In a sequential carry chain the back end is stalled most of the time due to data hazards. We expect a single carry chain to use $9 \cdot 14 = 126$cc or 31.5cc/op. Even in a twice interleaved carry chain, the latency is still 63cc, while the reciprocal throughput is still only 56cc. In other words, the twice interleaved case still has data hazards.

To overcome this, we implement a triple interleaved carry chain, as displayed

in Figure 4.5. In this case, the latency is reduced to 45cc, while the reciprocal throughput is 60cc. Conversely the bottleneck is not the latency, but the reciprocal throughput of the carry chain, of which the lower bound is 15cc/op.

$$
\begin{aligned}
f_0 &\rightarrow f_1 \rightarrow f_2 \rightarrow f_3 \rightarrow f_4 \rightarrow f_5, \\
f_4 &\rightarrow f_5 \rightarrow f_6 \rightarrow f_7 \rightarrow f_8 \rightarrow f_9, \\
f_8 &\rightarrow f_9 \rightarrow f_{10} \rightarrow f_{11} \rightarrow f_0 \rightarrow f_1
\end{aligned}
$$

Figure 4.5: Triple interleaved 12-limb carry chain.

4.3.3 Multiplication

For the 12-limb representation, we can choose to implement the multiply operation using the schoolbook method or using Karatsuba's trick. Before looking at Karatsuba multiplication in Subsection 4.3.4, we will look at the performance of the schoolbook multiplication method.

Floating points themselves store their exponents, so no additional realignment of the limbs is needed. Similar to radix $2^{25.5}$ multiplication, any product that is written into some limb h_i, where $i \geq 12$ is multiplied by $\left(2^{-255} \cdot 19\right)$ and added into h_{i-12}. In the end, for each limb, the multiplication of $h = f \cdot g$ is defined by:

$$
h_k = \sum_{i+j=k} f_i g_j + \sum_{i+j=k+12} \left(2^{-255} \cdot 19\right) f_i g_j
$$

All in all, the multiplication routine uses 144 `vmulpd` instructions for the inner multiplication operations. When we include the precomputation of $\left(2^{-255} \cdot 19\right) f_1, \ldots, \left(2^{-255} \cdot 19\right) f_{11}$, we end up with 153 `vmulpd` instructions in total. Furthermore, the routine uses 132 `vaddpd` instructions.

The execution of `vmulpd` and `vaddpd` happens on ports 0 and 1 respectively. The pressure on port 0 is the performance bottleneck, so we expect the schoolbook multiplication algorithm to take 144cc. Because we are computing 4 multiplications in parallel, this results in 36cc/op.

4.3.4 Karatsuba multiplication

As we have seen in Subsection 2.3.5, we can try Karatsuba multiplication to optimize the multiplication performance. The Karatsuba multiplication algorithm for radix $2^{21.25}$ is presented in Algorithm 4.2.

For optimization reasons, we have rewritten the computation extensively using basic math rules.[7] The majority of tweaks are inspired from [34].

We see that the Karatsuba multiplication algorithm is more complex than the schoolbook multiplication method. However, when we count up the amount of inner multiplications in Table 4.1, we see that we need slightly fewer `vmulpd` instructions. Particularly, the Karatsuba method is about 8% faster.

Table 4.1: Overview of the amount of `vmulpd` instructions needed in Algorithm 4.2.

lines	operation	count
16, 17	precompute $2^{-128} f_H$	6
16, 18	precompute $2^{-128} g_H$	6
15	compute L	36
16	compute H	36
19	compute M	36
20 – 31	multiply by 2^{-128}	5
20 – 31	multiply by 2^{128}	6
20 – 31	multiply by 38	11
Total		142

The resulting assembly snippet has a port pressure of 142 µops on port 0 and 130 µops on port 1. If we added more than 12 `vaddpd` or `vsubpd` instructions, port 1 would become the bottleneck. Therefore, we expect that adding another level of Karatsuba multiplication would not be beneficial for the performance of the multiplication algorithm. Besides, the implementation of another level of Karatsuba multiplication would need to align the limb exponents on 64-bit offsets. The result is dozens of extra multiplications by 2^{-64} and 2^{64}.

After implementing Karatsuba multiplication as a means of optimization there is only *one* optimization that we found we could do to reduce the pressure on port 0 even more. That is, we can implement some multiplications by 2^{-128} using bitwise operations. Specifically, we will implement some multiplications by 2^{-128} by setting the bit that encodes 2^{128} in the float's exponent from `1` to `0`.

Consider the function UnsetBit59(z), which sets the 59th bit of a floating-point to `0`. This function does not implement multiplication by 2^{-128} for *every* z. Instead, we will look for which inputs this function behaves correctly.

[7] The main benefit of these rewrites is the elimination of some subtractions in the accumulation in lines 20 – 31. The positive operand of the `vsubpd` instruction cannot be immediately loaded from memory and often needs an extra `vmovapd` instruction. Without these subtractions, we are able to better optimize for micro-fusion (see also [37, Section 3.4.2.1]).

Algorithm 4.2 Karatsuba multiplication algorithm

function MULTIPLY6(a, b) ▷ 6-limb schoolbook multiplication
  $(c_0, \ldots, c_{10}) \leftarrow (0, \ldots, 0)$
  for i **from** 0 **to** 5 **do**
    for j **from** 0 **to** 5 **do**
      $c_{i+j} = c_{i+j} + a_i b_j$
    end for
  end for
  return $(c_0, \ldots, c_{10})$
end function

procedure KARATSUBA(f, g) ▷ Compute $h = f \cdot g$
  Write $f_L = (f_0, \ldots, f_5)$
  Write $f_H = (f_6, \ldots, f_{11})$
  Write $g_L = (g_0, \ldots, g_5)$
  Write $g_H = (g_6, \ldots, g_{11})$

  $(L_0, \ldots, L_{10}) \leftarrow$ MULTIPLY6(f_L, g_L) ▷ Compute L
  $(H_0, \ldots, H_{10}) \leftarrow$ MULTIPLY6$(2^{-128} f_H, 2^{-128} g_H)$ ▷ Compute H

  $(f_{M,0}, \ldots, f_{M,5}) \leftarrow f_0 - \left(2^{-128} f_6\right), \ldots, f_5 - \left(2^{-128} f_{11}\right)$
  $(g_{M,0}, \ldots, g_{M,5}) \leftarrow \left(2^{-128} g_6\right) - g_0, \ldots, \left(2^{-128} g_{11}\right) - g_5$
  $(M_0, \ldots, M_{10}) \leftarrow$ MULTIPLY6(f_M, g_M) ▷ Compute M

  $h_0 \leftarrow L_0 + 38H_0 + 38 \cdot 2^{-128}(M_6 + L_6 + H_6)$
  $h_1 \leftarrow L_1 + 38H_1 + 38 \cdot 2^{-128}(M_7 + L_7 + H_7)$
  $h_2 \leftarrow L_2 + 38H_2 + 38 \cdot 2^{-128}(M_8 + L_8 + H_8)$
  $h_3 \leftarrow L_3 + 38H_3 + 38 \cdot 2^{-128}(M_9 + L_9 + H_9)$
  $h_4 \leftarrow L_4 + 38H_4 + 38 \cdot 2^{-128}(M_{10} + L_{10} + H_{10})$
  $h_5 \leftarrow L_5 + 38H_5$
  $h_6 \leftarrow L_6 + 38H_6 + 2^{128}(M_0 + L_0 + H_0)$
  $h_7 \leftarrow L_7 + 38H_7 + 2^{128}(M_1 + L_1 + H_1)$
  $h_8 \leftarrow L_8 + 38H_8 + 2^{128}(M_2 + L_2 + H_2)$
  $h_9 \leftarrow L_9 + 38H_9 + 2^{128}(M_3 + L_3 + H_3)$
  $h_{10} \leftarrow L_{10} + 38H_{10} + 2^{128}(M_4 + L_4 + H_4)$
  $h_{11} \leftarrow 2^{128}(M_5 + L_5 + H_5)$
end procedure

We will see that UnsetBit59$(z) = 2^{-128}z$ for all z where $|z| < 2^{257}$ and $x \in 2^{129}\mathbb{Z}$.

Let us first look at the case that $|z| \geq 2^{129}$. By assumption, $|z| < 2^{257}$. We see that the real exponent k is bounded by $129 \leq k < 257$. As such, the floating-point's exponents bits are bounded by $1152 \leq e < 1280$. Within these bounds, the 7th bit of e is always set. Thus, we conclude that for all $2^{129} \leq |z| < 2^{257}$, the UnsetBit59 function is equal to UnsetBit59$(x) = 2^{-128}x$.

The other case concerns $x = 0$. In this case, the exponent part $e = 0$. When we set the 7th exponent bit to 0, the value e remains zero. So for the value $x = 0$, UnsetBit59 is equal to the identity: UnsetBit59$(0) = 0 = 2^{-128} \cdot 0$.

Because of the restrictions we put on floating-point limbs, this means that we can use UnsetBit59 for all multiplications by 2^{-128} on limbs f_i where $2^{129} \leq b_i < 2^{257-53} = 2^{231}$. This is the case for $b_7 = 2^{145}$, $b_8 = 2^{170}$, $b_9 = 2^{192}$. Futhermore we can use UnsetBit59 on the 11th limb from the input operands to Algorithm 4.2. For all input operands it is guarantueed that $|f_i| < 2^{25}b_i$, so $2^{129} \leq b_{10} = 2^{213} < 2^{257-25} = 2^{232}$.

Using UnsetBit59, we replace 11 multiplications in Algorithm 4.2 by the instruction `vandpd ymmX, ymmX, [rel .unset59mask]`, where `.unset59mask` holds the value `0xF7FFFFFFFFFFFFFF`. We have moved a little bit of pressure from port 0 to port 5, which was completely idle before this point.

In the end, we have a multiplication routine with 131 executions (`vmulpd`) on port 0 and 130 executions on port 1 (`v{add,sub}pd`). Assuming the best case scenario, we will need $131/4 = 32.75$cc/op. The $4\times$ vectorized multiplication routine using radix $2^{21.25}$ is expected to be 66% faster than Sandy2x's $2\times$ vectorized multiplication in radix $2^{25.5}$. This is why—in this thesis—we have chosen to implement the finite-field arithmetic in double-precision floating points, using radix $2^{21.25}$. In Chapter 7 we will see the tragic repercussions of this choice[8].

Note that we have not implemented custom arithmetic for squaring and inversion. As we will see in the next chapter, we will not be able to put any $4\times$ vectorized squaring routines in our addition formulas. Futhermore, we will be reusing the inversion routines from another implementation.

[8]Spoiler: We underestimated the abysmal performance of the radix $2^{21.25}$ carry chain.

Chapter 5

Point doubling and addition

After having constructed our routines for arithmetic in $\mathbb{F}_{2^{255}-19}$, it is time to put these routines to work in the curve addition formulas. As we have mentioned a couple of times before, we are employing the addition formulas that have been devised by Renes, Costello, and Batina in 2016. Specifically, we will be using the formulas for the special case where $a = -3$. In [57], the relevant algorithms are Algorithm 6 for point doubling, and Algorithm 4 for point addition.

These formulas use homogeneous projective coordinates. That is, every coordinate (x, y) is represented as a tuple $(X : Y : Z) = (\lambda x : \lambda y : \lambda)$, where $\lambda \in \mathbb{Z}$. To map a projective coordinate back to its affine representation, we have to divide X and Y by λ $(= Z)$. Then $(x, y) = (\frac{X}{Z}, \frac{Y}{Z})$. The benefit of using a projective coordinate system is that any inversions in addition formulas can be eliminated, because for all c_1, c_2 it holds that $(\frac{x}{c_1}, \frac{y}{c_2}) = (X : Y : c_1 c_2 Z)$. An inversion operation in $\mathbb{F}_{2^{255}-19}$ involves hundreds of field multiplications and squarings. So eliminating these inversions leaves us with a scalar-multiplication algorithm that is much faster.

The Renes-Costello-Batina formulas are written with the presumption that memory space is scarce. However, the Sandy Bridge microarchitecture has 32KB of L1 data cache, so we do not need to worry about this. For the sake of clarity, we will therefore rewrite the algorithms from [57] into static single assignment (SSA) form. This will make it easier to rewrite the algorithms and eliminate operations.

In this chapter, we will construct routines for doubling and addition of points on Barreto's curve, which we will use in the next chapter. To this end, we have constructed subroutines that allow us to make use of the $4\times$ vectorized multiplications in Chapter 4. Note that the optimizations devised in this chapter need not specifically use radix $2^{21.25}$ arithmetic. Whenever $4\times$

vectorized multiplication routines are provided, these optimizations apply. For instance, Intel's Haswell microarchitecture supports $4\times$ packed 64-bit integer arithmetic, so these optimizations could be implemented on Haswell using radix $2^{25.5}$.

5.1 Doubling

The literal Renes-Costello-Batina formulas for doubling a point on Barreto's curve are

$$\begin{aligned} X_3 &= 2XY(Y^2 + 3(2XZ - bZ^2)) - 6XZ(2bXZ - X^2 - 3Z^2), \\ Y_3 &= (Y^2 - 3(2XZ - bZ^2))(Y^2 + 3(2XZ - bZ^2)) \\ &\quad + 3(3X^2 - 3Z^2)(2bXZ - X^2 - 3Z^2), \\ Z_3 &= 8Y^3Z. \end{aligned}$$

In their paper, these formulas are implemented using 34 distinct operations of which $8\mathbf{M} + 3\mathbf{S} + 2\mathbf{m} + 21\mathbf{a}$.[1] These operations are listed—in SSA form—in algorithm C.1 in appendix C. In this listing, every computation of v_i corresponds to the computation on line i in algorithm 6 of [57].

5.1.1 Batching multiplications

We will first look at how to batch the 11 multiplication and squaring operations in $4\times$ vectorized algorithms. Because computing squarings is cheaper than computing multiplications, it would be wonderful if we could manage to group all 3 squarings into one batched squaring operation, and all 8 multiplications into two batched multiplications.

We see, however, that this is not possible. Consider an approach putting all 3 squarings into a single batch. Observe that from the multiplication operations that are left, every value from $\{v_{14}, v_{15}, v_{26}, v_{30}, v_{32}\}$ depends on at least one value from $\{v_4, v_6, v_{28}\}$ to be ready. In other words, the latter group of values must be computed before the former group of values can be computed. This means we are forced to use three batched multiplication steps for the 8 multiplications *along* with the batched squaring.

Instead we opt to compute 3 batched multiplications, which *include* the squaring operations. We found that the multiplications and squarings that

[1] Where $\mathbf{M}$ stands for multiplication, $\mathbf{S}$ for squaring, $\mathbf{m}$ for multiplication by a small constant and $\mathbf{a}$ for addition/subtraction.

result in the most structured dependency graph are when

batch 1 computes $\{v_1, v_3, v_6, v_{28}\}$;
batch 2 computes $\{v_2, v_4, v_{26}, v_{30}\}$; and
batch 3 computes $\{v_{14}, v_{15}, v_{32}\}$.

5.1.2 Eliminating chained additions

Apart from the assumption that register space is limited, the Renes-Costello-Batina formulas also consider multiplications by small constants to be more expensive than multiple additions. They have therefore opted to implement all multiplications by small constants (except for b) using additions alone.

However, on our platform, the reciprocal throughputs of the `vmulsd`, `vmulpd`, `vaddsd`, and `vaddpd` instructions are all 1 cycle. This means we can reduce the amount of operations by replacing these chained additions by multiplications by their respective small constants. In the doubling formulas, we have devised these new assignments:

New assignment	**Eliminated ops**
$v_{17} \leftarrow 3v_3$	v_{16}
$v_{22} \leftarrow 3v_{20}$	v_{21}
$v_{24} \leftarrow 3v_1$	v_{23}
$v_{34} \leftarrow 8v_{28}$	v_{33}
$v_{11} \leftarrow -6v'_9$, where $v'_9 \leftarrow v_6 - \frac{b}{2}v_3$	v_7, v_{10}

After replacing 11 $\mathbf{a}$-operations with 5 $\mathbf{m}$-operations, the new cost of the doubling formula is $8\mathbf{M} + 3\mathbf{S} + 7\mathbf{m} + 10\mathbf{a}$.

In the assembly implementation, many operations have been reordered for better out-of-order execution. Furthermore, some operations—other than multiplications and carries—are batched whenever this seemed beneficial. Yet, the assembly code also adds a lot of shuffling operations, which will add some performance overhead. The structure of the resulting assembly code is given in Algorithm 5.1.

Algorithm 5.1 Exception-free point doubling for Barreto's curve, as implemented by this work.

procedure DOUBLE(X, Y, Z) ▷ Compute $(X_3 : Y_3 : Z_3) = 2(X : Y : Z)$

$v_{2X} \leftarrow X + X$
$v_1 \leftarrow X \cdot X$; $v_6 \leftarrow X \cdot Z$; $v_3 \leftarrow Z \cdot Z$; $v_{28} \leftarrow Y \cdot Z$
CARRY v_1, v_6, v_3, v_{28}
$v_{24} \leftarrow 3 \cdot v_1$; $v_{18} \leftarrow 2b \cdot v_6$; $v'_8 \leftarrow -\frac{b}{2} \cdot v_3$; $v_{17} \leftarrow 3 \cdot v_3$
$v_{25} \leftarrow v_{24} - v_{17}$; $v_{19} \leftarrow v_{18} - v_{17}$
$v_{20} \leftarrow v_1 - v_{19}$
$v_{22} \leftarrow -3v_{20}$
$v_9 \leftarrow v'_8 + v_6$
$v_{11} \leftarrow -6 \cdot v_9$; $v_{34} \leftarrow 8 \cdot v_{28}$
CARRY $v_{11}, v_{34}, v_{22}, v_{25}$
$v_{29} \leftarrow v_{28} + v_{28}$
$v_{30} \leftarrow v_{22} \cdot v_{29}$; $v_{26} \leftarrow v_{22} \cdot v_{25}$; $v_2 \leftarrow Y \cdot Y$; $v_5 \leftarrow v_{2X} \cdot Y$
CARRY v_{30}, v_{26}, v_2, v_5
$v_{12} \leftarrow v_2 - v_{11}$
$v_{13} \leftarrow v_2 + v_{11}$
$v_{32} \leftarrow v_2 \cdot v_{34}$; $v_{15} \leftarrow v_5 \cdot v_{12}$; $v_{14} \leftarrow v_{12} \cdot v_{13}$
CARRY v_{32}, v_{15}, v_{14}
$v_{31} \leftarrow v_{15} - v_{30}$
$v_{27} \leftarrow v_{14} + v_{26}$

$X_3 \leftarrow v_{31}$
$Y_3 \leftarrow v_{27}$
$Z_3 \leftarrow v_{34}$
end procedure

5.2 Addition

The Renes-Costello-Batina formulas for doubling a point on Barreto's curve are

$$\begin{aligned}
X_3 &= (X_1Y_2 + X_2Y_1)\,(Y_1Y_2 + 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2)) \\
&\quad - 3(Y_1Z_2 + Y_2Z_1)\,(b(X_1Z_2 + X_2Z_1) - X_1X_2 - 3Z_1Z_2)\,, \\
Y_3 &= 3(3X_1X_2 - 3Z_1Z_2)\,(b(X_1Z_2 + X_2Z_1) - X_1X_2 - 3Z_1Z_2) \\
&\quad + (Y_1Y_2 - 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2))\,(Y_1Y_2 + 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2))\,, \\
Z_3 &= (Y_1Z_2 + Y_2Z_1)\,(Y_1Y_2 - 3(X_1Z_2 + X_2Z_1 - bZ_1Z_2)) \\
&\quad + (X_1Y_2 + X_2Y_1)(3X_1X_2 - 3Z_1Z_2).
\end{aligned}$$

In [57, Algorithm 4], these formulas are implemented in 43 operations. The cost of the original algorithm is $12\mathbf{M} + 2\mathbf{m} + 29\mathbf{a}$. The untouched algorithm is listed in Algorithm C.2 in SSA form.

5.2.1 Batching multiplications

As in Section 5.1, we will compute the multiplications in batches of 4, using our $4\times$ vectorized implementation from Section 4.3.4.

We can group these multiplications using the same method as in Subsection 5.1.1, by looking at the dependency tree. The three groups of multiplications that result in the most structured dependency graph are when

batch 1 computes $\{v_1, v_2, v_2, v_{16}\}$;
batch 2 computes $\{v_6, v_{11}, v_{36}, v_{37}\}$; and
batch 3 computes $\{v_{35}, v_{39}, v_{41}, v_{42}\}$.

5.2.2 Eliminating chained additions

The formulas for addition contain fewer chained additions than the doubling formulas. Nonetheless, they contain four of them, which we can eliminate and replace by a single multiplication operation:

New assignment	Eliminated ops
$v_{22} \leftarrow 3v_{20}$	v_{21}
$v_{27} \leftarrow 3v_3$	v_{26}
$v_{31} \leftarrow 8v_{29}$	v_{30}
$v_{33} \leftarrow 3v_1$	v_{32}

The substitutions replace 8 $\mathbf{a}$-operations by 4 $\mathbf{m}$-operations. This results in a new cost of $12\mathbf{M} + 6\mathbf{m} + 21\mathbf{a}$.

As with the code for doubling, some operations other than multiplications and carries have been batched. In the same way, the assembly code of the addition formula contains many shuffling operations, which will surely impact the performance. The final code is resembled by Algorithm 5.2.

Algorithm 5.2 Exception-free point addition for Barreto's curve, as is has been implemented by this work.

procedure ADD(X_1, Y_1, Z_1, X_2, Y_2, Z_2)
▷ Compute $(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2)$

$v_4 \leftarrow X_1 + Y_1$; $v_9 \leftarrow Y_1 + Z_1$; $v_{14} \leftarrow X_1 + Z_1$
$v_5 \leftarrow X_2 + Y_2$; $v_{10} \leftarrow Y_2 + Z_2$; $v_{15} \leftarrow X_2 + Z_2$
$v_1 \leftarrow X_1 \cdot X_2$; $v_2 \leftarrow Y_1 \cdot Y_2$; $v_3 \leftarrow Z_1 \cdot Z_2$; $v_{16} \leftarrow v_{14} \cdot v_{15}$
CARRY v_1, v_2, v_3, v_{16}
$v_{17} \leftarrow v_1 + v_3$
$v_{18} \leftarrow v_{16} - v_{17}$
$v_{19} \leftarrow b \cdot v_3$
$v_{20} \leftarrow v_{19} - v_{18}$
$v_{25} \leftarrow b \cdot v_{18}$
$v_{27} \leftarrow 3 \cdot v_3$
$v_{28} \leftarrow v_{25} - v_{27}$
$v_{29} \leftarrow v_{28} - v_1$; $v_{v_1 - v_3} \leftarrow v_1 - v_3$
$v_{22} \leftarrow 3 \cdot v_{20}$; $v_{31} \leftarrow 3 \cdot v_{29}$; $v_{34} \leftarrow 3 \cdot v_{v_1 - v_3}$
CARRY v_{22}, v_{31}, v_{34}
$v_{23} \leftarrow v_2 - v_{22}$
$v_{24} \leftarrow v_2 + v_{22}$
$v_6 \leftarrow v_4 \cdot v_5$; $v_{11} \leftarrow v_9 \cdot v_{10}$; $v_{36} \leftarrow v_{31} \cdot v_{34}$; $v_{37} \leftarrow v_{23} \cdot v_{24}$
CARRY $v_6, v_{11}, v_{36}, v_{37}$
$v_8 \leftarrow v_6 - v_7$; $v_{13} \leftarrow v_{11} - v_{12}$
$v_{38} \leftarrow v_{36} + v_{37}$
$v_{35} \leftarrow v_{13} \cdot v_{31}$; $v_{39} \leftarrow v_8 \cdot v_{24}$; $v_{41} \leftarrow v_{13} \cdot v_{23}$; $v_{42} \leftarrow v_8 \cdot v_{33}$
CARRY $v_{35}, v_{39}, v_{41}, v_{42}$
$v_{43} \leftarrow v_{41} + v_{42}$
$v_{40} \leftarrow v_{39} - v_{35}$

$X_3 \leftarrow v_{40}$
$Y_3 \leftarrow v_{38}$
$Z_3 \leftarrow v_{43}$
end procedure

Chapter 6

Scalar multiplication

Let us make the last step towards a complete variable-point scalar-multiplication algorithm. First, we will select a suitable scalar-multiplication algorithm and optimize it for our specific instance. In the end of this chapter, we will review all the steps involved in the variable-point scalar multiplication.

For the scalar multiplication, we will use the left-to-right double-and-add algorithm from Subsection 2.2.1. We will apply the fixed-window optimization method from Subsection 2.2.3, with the key-digits recoded into signed form, as described in Subsection 2.2.4.

6.1 Choosing w

To choose the window size w, we will use Equation 2.8 to optimize for the window size that leaves us with the fewest amount of operations. Beforehand, we do not yet know the load of the addition algorithm relative to the doubling algorithm. An educated guess is that—because the doubling formula needs less field additions—**Dbl** $\approx 0.9 \cdot$ **Add**.[1] In Table 6.1 the total amount of "addition costs"—i.e. the cost equivalent to some amount of addition operations—are given for window sizes from 1 up to 9.

We see that in Table 6.1 the amount of operations becomes large whenever w is small, because with a low w the algorithm approximates the double-and-add algorithm with no fixed-window optimizations. When w is too large, the amount of values precomputed in the lookup becomes so large that too few of them will actually be used during the scalar multiplication.

[1] After micro-benchmarking the doubling and add algorithms from Listings 5.1 and 5.2 on the Ivy Bridge microarchitecture, we see that the the doubling algorithm executes in about 1144 cycles, while the addition algorithm needs about 1232 cycles. Apparently, the algorithm for point doubling is indeed about 10% faster than the algorithm for point doubling.

Table 6.1: Amount of operations, equivalent to an addition, for window sizes w from 0 to 9, where it is assumed that either **double** $\approx 1.0 \cdot$ **add**, or **double** $\approx 0.9 \cdot$ **add**. The smallest values are highlighted in **bold**.

w	**double** $\approx 1.0 \cdot$ **add**	**double** $\approx 0.9 \cdot$ **add**
1	511.0	485.4
2	384.0	358.4
3	344.0	317.3
4	326.0	300.1
5	**322.0**	**294.7**
6	329.0	300.9
7	355.0	325.3
8	414.0	382.1
9	539.0	499.7

Furthermore, because the size of the lookup table is $\mathcal{O}(2^w)$, we will spend too many cycles scanning the lookup table.

A sweet spot is seen for the values $w \in \{4, 5, 6\}$, with the smallest value for $w = 5$, which we will therefore choose for our implementation. Consequently, we will be using a lookup table of 16 group elements. The amount of operations needed will be

$$\textbf{\#ops} = \underbrace{8 \cdot \textbf{double} + 7 \cdot \textbf{add}}_{\text{precomputation}} + \underbrace{255 \cdot \textbf{double} + 52 \cdot \textbf{add}}_{\text{loop}},$$

which adds up to $263 \cdot$ **double** $+ 59 \cdot$ **add** operations in total.

Now that we have determined w, we can fill in all the parameters in algorithm 2.4. The result is algorithm 6.1, which resembles the scalar-multiplication algorithm that has been implemented for this thesis.

6.2 Overview of the complete algorithm

At this point we have visited many parts of the implementation of our variable-point scalar-multiplication algorithm. We have seen the core routines, including the finite-field arithmetic, the algorithms for doubling and addition, and the scalar multiplication. However, the scalar-multiplication algorithm from Algorithm 6.1 alone does not constitute a complete and secure variable-point scalar-multiplication algorithm. Some other steps, like input validation and encoding R, still need to be done.

A description of the definitive function for variable-point scalar multiplication of points on Barreto's curve is given by Algorithm 6.2. The steps that are added in this algorithm are described in the rest of this section: lines 3

Algorithm 6.1 Signed double-and-add

1: **function** ScalarMultiplication(k, P) ▷ Compute $[k]P$
2: $\mathbf{T} \leftarrow (\mathcal{O}, P, \ldots, [16]P)$ ▷ Precompute $([2]P, \ldots, [16]P)$
3: $k' \leftarrow$ RecodeSigned(Windows$_5(k)$)
4: $R \leftarrow \mathcal{O}$
5: **for** i **from** 50 **down to** 0 **do**
6: **for** j **from** 0 **to** 4 **do**
7: $R \leftarrow [2]R$ ▷ 5 doublings
8: **end for**
9: **if** $k'_i < 0$ **then**
10: $R \leftarrow R - \mathbf{T}_{-k'_i}$ ▷ Addition
11: **else**
12: $R \leftarrow R + \mathbf{T}_{k'_i}$ ▷ Addition
13: **end if**
14: **end for**
15: **return** R ▷ $R = (X_R : Y_R : Z_R)$
16: **end function**

and 9—about the encoding and decoding of points—are described in Subsection 6.2.1. Point validation on line 4 is described in Subsection 6.2.2. We have already seen the scalar multiplication in line 7 in the previous section. Lastly, in Subsection 6.2.3, we elaborate on the management of the MXCSR register.

Algorithm 6.2 Complete variable-point scalar-multiplication algorithm

1: **function** VariablePointMultiplication(k, P_{in}) ▷ Compute $R = [k]P_{\text{in}}$
2: `old_mxcsr` $\leftarrow$ ReplaceMXCSR($\texttt{1F80}_{16}$)
3: $(X_P, Y_P, Z_P) \leftarrow$ DecodeAffinePoint(P_{in})
4: **if not** ValidPoint(X_P, Y_P, Z_P) **then**
5: **error**
6: **end if**
7: $(X_R, Y_R, Z_R) \leftarrow$ ScalarMultiplication$(k, (X_P, Y_P, Z_P))$
8: $R_{\text{out}} \leftarrow$ EncodeProjectivePoint(X_R, Y_R, Z_R)
9: **if not** CheckMXCSR() **then**
10: **error**
11: **end if**
12: RestoreMXCSR(`old_mxcsr`)
13: **return**
14: **end function**

6.2.1 Point encoding

To operate on the point that is provided by the user, we have to decode it first. The function DecodeAffinePoint(P_{in}) decodes the 64-byte string P_{in} into the point $P = (X_P : Y_P : Z_P)$. Its first 32 bytes represent X_P and the last 32 bytes represent X_P. Each part is interpreted as a 256-bit number in big-endian order and recoded into 12 double floating-point limbs. Z_P is set to 1.

The point at infinity is represented by the full-zero string. That is, whenever $P_{\text{in}} = (0, \ldots, 0)$, instead of decoding the invalid point $(0 : 0 : 1)$, we set $P = (0 : 1 : 0)$.

EncodeProjectivePoint(X_R, Y_R, Z_R) encodes a projective point $R = (X_R : Y_R : Z_R)$ back to a byte string. It first computes the affine representation of R. For the inversion of Z_R we use the radix-2^{51} inversion routine from Sandy2x [21], which is based on the implementation of [9] and internally computes $Z_R^{-1} = Z_R^{2^{255}-21}$.[2]

Then we use Z_R^{-1} to calculate the affine point by computing $R = (x_R, y_R) = (X_R Z_R^{-1}, Y_R Z_R^{-1})$. Finally, we encode x_R and y_R back into a 64-byte string using big-endian ordering.

Note that in the encoding of R, we have implicitly taken the point at infinity into account. That is, because for all $R = \mathcal{O}$, we know that $Z_R = 0$. Because of this, the computed value of $Z_R^{-1} = 0^{2^{255}-21} = 0$. Multiplying the other coordinates of R results in $(x_R, y_R) = (X_R \cdot 0, Y_R \cdot 0) = (0, 0)$. This is encoded into the all-zero string, which encodes $\mathcal{O}$.

6.2.2 Point validation

Before executing the scalar-multiplication algorithm on the input point, we have to check whether the given point actually exists on the curve. In ValidPoint(X_R, Y_R, Z_R), we will check if the coordinates of R are valid according to the curve equation given in Equation 3.1. If the coordinates are invalid, we will refuse to compute the scalar multiplication, and raise an error instead.

6.2.3 The MXCSR control and status register

On Intel processors, the behavior of SIMD floating-point operations can be configured using the "MXCSR Control and Status Register" [36, Section 10.2.3]. The MXCSR is a 32-bit register that contains fields for rounding

[2] Another option for inverting Z_R is by using the method that was recently published in [13]. However, because this method was not available when Sandy2x was implemented, we believe it to be unfair to use it for our work.

control, flag bits for floating-point exceptions and mask bits for floating-point exceptions. Because our field-arithmetic implementation assumes a specific behavior from the CPU, we must set the MXCSR before executing any floating-point instruction.

First of all, we set the rounding control field to "Round to nearest (even)" [36, Table 4-8]. Then, to prevent any side channels due to exceptions being raised, we disable them by setting all the mask bits for floating-point exceptions to 1.[3] Lastly, we set any other bits to their default value. This includes setting all the exception flag bits to 0. The resulting value that we write to the MXCSR is $\texttt{1F80}_{16}$.

At the end of the algorithm, we must clean up after ourselves and restore the value of the MXCSR before returning. As a defense-in-depth measure, we will also check whether any unexpected floating-point exceptions have triggered, e.g. overflow, underflow, or divide-by-zero. If an unexpected exception has occured, we refuse to return the result of our scalar multiplication, and return an error instead.

[3] If we would not disable exceptions, the processor could—for example—be configured by the caller to throw an exception whenever a rounding error occurs. This could be a disastrous vulnerability, because our carry chain is *based* on triggering rounding errors. If the processor throws an exception whenever a rounding error occurs, an attacker could use this as an oracle to gain knowledge about the secret scalar.

Chapter 7

Results and discussion

Our variable-point scalar-multiplication algorithm is completed, and now we can compare the performance of Curve25519's Sandy2x implementation to ours. From our building blocks, we will first estimate how fast we expect our implementation to be. Then we will decompose the results and look at a profile of the individual building blocks. We will see which are the bottlenecks of our algorithm, and consider how our algorithm could be improved.

7.1 Performance expectations

There are two approaches through which we can estimate the performance that we expect to achieve with our implementation. First, we will look at similar implementations of variable-point scalar multiplication, i.e. for short Weierstrass curves with $a = -3$. Secondly, we will estimate the lower bound of the total amount of cycles that our implementation would take, building up from the cycle counts that we have computed earlier in Chapter 4.

7.1.1 From other implementations

This work describes the first effort to write an optimized implementation of the Renes-Costello-Batina addition formulas [57] in software.[1] In their work, they have substituted the complete formulas for OpenSSL's original formulas of the time, for the computation of variable-point scalar multiplication on the NIST curves (which have $a = -3$). While they do not report cycle counts, they do report that, for the NIST P-256 curve, their formulas perform $1.38\times$ slower. OpenSSL's implementation is based on the incomplete

[1] At the time of writing, the only other optimized implementation of these addition formulas has been [46], which implements the formulas in hardware.

formulas from [6]. As such, Renes, Costello and Batina, argue that "the cost of completeness" can be estimated to be around a factor of $1.38\times$.

Because in Barreto's curve, b is small, it would be unfair to compare our implementation with an implementation of a NIST curve. However, [14] has examined the performance of a scalar-multiplication algorithm for a curve similar to Barreto's, i.e. one that uses a pseudo-Mersenne prime, has $a = -3$, and has a small b. For their variable-point scalar-multiplication algorithm (which they refer to as "w-256-mers"), they report a cycle count of 278kcc. We multiply this by the factor 1.38 from [57], to get 384kcc as a first estimate of the expected cycle count for our algorithm.

However, in Appendix C, the authors of [14] explicitly mention the consideration of using complete addition formulas based on [15]. They claim that the complete addition formulas would cost twice as much as the incomplete formulas that they have used. Thereupon, we might even expect a cycle count of $2 \cdot 278\text{kcc} = 556\text{kcc}$.

7.1.2 From building blocks

In Section 4.3 we have reported the theoretical lower bounds for the radix-$2^{21.25}$ field-arithmetic operations. These are listed in Table 7.1. We can use these lower bounds to calculate an expected lower bound for the cycle count of the complete scalar-multiplication algorithm.

Table 7.1: Theoretical lower-bound reciprocal throughputs of finite-field building blocks.

Operation	**Throughput**$^{-1}$
$4\times$ batched carry	60cc
$4\times$ batched multiply	131cc
$4\times$ batched multiply by small constant	12cc
$4\times$ batched addition/subtraction	12cc

In Algorithm 5.1, for doubling a point, we count 5 batched carries, 4 batched multiplications, 3 batched multiplications by a small constant, and 9 batched additions or subtractions. When we pile up these operations, we see that the lower-bound reciprocal throughput of a doubling operation is

$$\textbf{throughput}^{-1}_{\text{double}} = 5 \cdot 60 + 4 \cdot 131 + 3 \cdot 12 + 9 \cdot 12 = 968\text{cc}.$$

In the case of the addition formula from Algorithm 5.2, we count 5 batched carries, 4 batched multiplications, 4 batched multiplications by a small constant, and 13 batched additions. These operations pile up to a lower-bound reciprocal throughput of

$$\textbf{throughput}^{-1}_{\text{add}} = 5 \cdot 60 + 4 \cdot 131 + 4 \cdot 12 + 13 \cdot 12 = 1028\text{cc}.$$

In Chapter 6, we have committed to a scalar-multiplication algorithm that uses 263 double operations and 59 additions. This results in a theoretical lower-bound reciprocal throughput of

$$\textbf{throughput}^{-1}_{\text{scalarmult}} = 263 \cdot 968 + 59 \cdot 1028 = 315236\text{cc}.$$

Of course, this estimation is far from realistic. For one, it leaves out all the overhead that comes from shuffling data, which is 288 instructions (23% overhead) for doubling and 396 instructions (31% overhead) for addition.[2] Second, the lower-bound estimate assumes that no data hazards occur *at all*. Third, all access to memory is left out. Lastly, it *only* concerns the cost of the doubling and addition routines. Some cost-heavy operations in the scalar-multiplication algorithm are left out, like lookup-table scanning and the inversion operation at the end. All in all, we estimate that an added overhead of 20% should be considered in our lower bound in order to encompass the complete variable-point scalar-multiplication algorithm. This leads to a rough estimate of about 378kcc.

We see that the calculated estimate is consistent with the estimations derived from the literature in Subsection 7.1.1. Although, the new estimate suggests that the claim from [14]—that the cost of completeness is a factor 2—is somewhat pessimistic.

7.2 Results

The complete algorithm was tested and benchmarked on Sandy Bridge[3], Ivy Bridge[4], and Haswell[5] machines. All measurements were done with Turbo Boost disabled, all Hyper-Threading cores shut down, and with the CPU clocked at the maximum frequency. A more detailed description of the benchmarking setup can be found in Appendix D.

Table 7.2: Measured cycle counts of the variable-base-point scalar-multiplication routines from Sandy2x and this work.

Implementation	Sandy Bridge	Ivy Bridge	Haswell
Sandy2x [21]	159kcc	157kcc	–
this work	390kcc	383kcc	340kcc

On Sandy Bridge, we have measured a cycle count of 390kcc. Hence we have found the answer to our question what the cost of a Weierstrass curve with complete formulas is, over Curve25519: It is around a factor 2.5.

[2] These instruction counts include blend and broadcast operations.
[3] Model: Intel Core-i7-2600
[4] Model: Intel Core-i5 3210M
[5] Model: Intel Core-i7 4770K

We see that the measured cycle count is close to our estimates. We profiled the complete implementation to see which parts of the algorithm were costly. The results are displayed in Table 7.3.[6]

Table 7.3: Cycle counts of the different parts of our scalar-multiplication algorithm (measured on the Ivy Bridge machine).

Part	Cycles	Percentage
point doubling	275732	72.0%
point addition	61940	16.2%
precomputation	19412	5.1%
inversion	14144	3.7%
lookup-table scanning	10088	2.6%
other	1522	0.4%

The doublings and additions, including the computation of the lookup table, account for 357.1kcc (93.3%). In other words, we see that the actual throughput of the doubling and addition routines is 13.3% slower than the theoretical lower bound. This difference can be attributed to the shuffling operations that have been added in the implementations of the addition formulas.

7.3 Discussion

7.3.1 Performance

After implementing the variable-base-point scalar-multiplication algorithm for Barreto's curve, we see that the complete addition formulas perform considerably slower than Curve25519. However, this result is not surprising, given the fact that the Renes-Costello-Batina formulas need considerably more operations.

While a factor of 2.5 slowdown may seem dramatic, in the context of software—depending on the application—it may be barely noticeable. Moreover, the Renes-Costello-Batina formulas are applicable to *any* elliptic curve, including Curve25519. Thus, the Renes-Costello-Batina formulas can always be used if the implementor aims to unify different cryptographic implementations, for example to reduce the code size. Such an implementation would need to add functionality for transforming point Curve25519 and isomorphic Weierstrass curve.

[6] In our code, the parts of the algorithm cannot be easily profiled using profiling tools, because most functions are manually inlined. Therefore we have chosen to construct the profile manually. The cycle counts are obtained by running the implementation, while *skipping* some part of it. We then subtract the measured cycle count from the baseline measurement, i.e. the cycle count of the complete algorithm (382838cc).

7.3.2 The cost of completeness

Another interesting observation is the measurement of the "cost of completeness" for Weierstrass curves. As we mentioned in Subsection 7.1.1, [57] reported the overhead of the Renes-Costello-Batina formulas to be $1.38\times$. On the other hand, [14] reported the overhead to be around a factor 2. However, if we use their implementation—that uses incomplete formulas—as a basis, we can make a new estimation of the overhead of the complete formulas. When we divide our cycle count 390kcc by their cycle count 278kcc, we get a factor of 1.40. We observe that this factor lies close to the factor reported by [57]. Therefore, we conclude that the estimate from [14], which reports a factor 2, is too pessimistic.

In the end, we recommend every implementor to use the complete addition formulas, when they need to implement arithmetic over Weierstrass curves. The overhead of the Renes-Costello-Batina formulas is small, and they prevent any exceptions from occurring in scalar-multiplication algorithms.

7.3.3 Design choices

When reflecting on our results, we make two observations. First of all, we see that the batched carry chain is relatively expensive. This results from the fact that—contrary to the other building blocks in our algorithm—the carry chain is mainly dominated by instruction latency, instead of throughput. Sandy2x's carry chain is fast, because most instructions on integers execute in a single cycle. However, our implementation uses only the floating-point instructions `vaddpd` and `vsubpd`, which need 3 cycles to finish. Another factor is that the Sandy2x carry chain uses the CPU ports 0, 1 and 5 simultaneously, while ours can only use port 1.

Another downside to using floating-point representation for large integers, instead of integers is the overhead that is added due to shuffling. On Sandy Bridge, packing and unpacking all values from a 4x vectorized `ymmX`-register costs 3 µops on port 5. For `xmmX`-registers, these packing operations cost only one µop.

With this new knowledge, we are unsure whether the use of radix $2^{21.25}$ using floating-point numbers has been the correct choice for the Sandy Bridge microarchitecture. Regardless, Intel has introduced $4\times$ vectorized 64-bit integer arithmetic and more powerful shuffling instructions in AVX2, which is included in the Haswell microarchitecture. Using AVX2, we can implement the finite-field arithmetic in radix $2^{25.5}$ using integer arithmetic, while still using the same code for doubling and addition. This will greatly improve the performance of the carry chain, and reduce the overhead from shuffling.

7.3.4 Future work

From this results of this work, we are only able to directly gather knowledge on the performance of traditional Weierstrass curves with $a = -3$. While we can generalize the results from this work onto other curves, these generalizations will undoubtedly be inaccurate. Therefore, we suggest implementing the Renes-Costello-Batina formulas for more prime order curves than just Barreto's—for example BN-curves [4]—to see how the complete addition formulas perform in these cases.

Apart from the implementation of other curves, the implementation on other platforms should be considered. For instance, after Sandy Bridge, Intel has released numerous newer microarchitectures with more powerful instruction sets. For the sake of more data points, we can adapt our implementation to the Haswell microarchitecture and compare the performance to [27], which is currently the fastest implementation of Curve25519 for that microarchitecture.

Chapter 8

Conclusion

In this thesis we have implemented an optimized algorithm for variable-base-point scalar multiplication for Barreto's curve on the Sandy Bridge microarchitecture. On Sandy Bridge, an execution of our algorithm needs about 390 thousand cycles. This is a factor 2.5 slower than Sandy2x, the fastest implementation of scalar-multiplication algorithm for Curve25519.

We conclude that scalar multiplication using complete addition formulas is indeed considerably slower for Weierstrass curves than it is for Curve25519. Furthermore, we see that the general overhead of the Renes-Costello-Batina formulas for arithmetic on Weierstrass curves is about a factor of 1.4.

The complete addition formulas can be used to implement efficient scalar multiplication on other elliptic curves. Their overhead is small, and they allow for safe arithmetic on Weierstrass curves. However, when a new cryptographic protocol demands for a prime-order curve, it is preferred to use Curve25519 in combination with Ristretto instead.

Bibliography

[1] Adleman, L.: A subexponential algorithm for the discrete logarithm problem with applications to cryptography. In: SFCS 1979. pp. 55–60 (Oct 1979), https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4568001 9

[2] Barker, E.: Recommendation for key management (Jan 2016), https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf 9

[3] Barreto, P.S.L.M.: on Twitter (May 2017), https://twitter.com/pbarreto/status/869103226276134912 25

[4] Barreto, P.S.L.M., Naehrig, M.: Pairing-friendly elliptic curves of prime order. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 319–331. Springer (Aug 2006), https://eprint.iacr.org/2005/133.pdf 58

[5] Benger, N., van de Pol, J., Smart, N.P., Yarom, Y.: "Ooh Aah... Just a Little Bit": A small amount of side channel can go a long way. In: Batina, L., Robshaw, M. (eds.) CHES 2014. LNCS, vol. 8731, pp. 75–92. Springer (Sep 2014), https://eprint.iacr.org/2014/161.pdf 23

[6] Bernstein, D.J.: A software implementation of NIST P-224. In: ECC 2001 (Oct 2001), http://cr.yp.to/nistp224/timings.html 54

[7] Bernstein, D.J.: Floating-point arithmetic and message authentication (Sep 2004), http://cr.yp.to/papers.html#hash127 34, 35

[8] Bernstein, D.J.: Curve25519: New Diffie-Hellman speed records. In: Yung, M., Dodis, Y., Kiayias, A., Malkin, T. (eds.) PKC 2006. LNCS, vol. 3958, pp. 207–228. Springer (Apr 2006), https://cr.yp.to/ecdh/curve25519-20060209.pdf 4, 13, 25, 34

[9] Bernstein, D.J., Duif, N., Lange, T., Schwabe, P., Yang, B.Y.: High-speed high-security signatures. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 124–142. Springer (Sep / Oct 2011), https://ed25519.cr.yp.to/ed25519-20110926.pdf 5, 27, 51

[10] Bernstein, D.J., Hamburg, M., Krasnova, A., Lange, T.: Elligator: elliptic-curve points indistinguishable from uniform random strings. In: Sadeghi, A.R., Gligor, V.D., Yung, M. (eds.) ACM CCS 2013. pp. 967–980. ACM Press (Nov 2013), https://elligator.cr.yp.to/elligator-20130828.pdf 27

[11] Bernstein, D.J., Lange, T.: SafeCurves: choosing safe curves for elliptic-curve cryptography; Completeness, https://safecurves.cr.yp.to/complete.html, accessed 7 Mar. 2019 6

[12] Bernstein, D.J., Schwabe, P.: NEON crypto. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 320–339. Springer (Sep 2012), https://cr.yp.to/highspeed/neoncrypto-20120320.pdf 25, 28, 32

[13] Bernstein, D.J., Yang, B.Y.: Fast constant-time gcd computation and modular inversion. Cryptology ePrint Archive, Report 2019/266 (Mar 2019), https://eprint.iacr.org/2019/266 51

[14] Bos, J.W., Costello, C., Longa, P., Naehrig, M.: Selecting elliptic curves for cryptography: an efficiency and security analysis. Journal of Cryptographic Engineering 6(4), 259–286 (Nov 2016), https://doi.org/10.1007/s13389-015-0097-y 54, 55, 57

[15] Bosma, W., Lenstra, H.: Complete systems of two addition laws for elliptic curves. Journal of Number Theory 53(2), 229–240 (1995), http://www.sciencedirect.com/science/article/pii/S0022314X85710888 4, 54

[16] Brier, E., Joye, M.: Weierstraß elliptic curves and side-channel attacks. In: Naccache, D., Paillier, P. (eds.) PKC 2002. LNCS, vol. 2274, pp. 335–345. Springer (Feb 2002), http://joye.site88.net/papers/BJ02espa.pdf 15, 26

[17] Brier, E., Joye, M.: Fast point multiplication on elliptic curves through isogenies. In: Fossorier, M., Høholdt, T., Poli, A. (eds.) AAECC 2003. LNCS, vol. 2643, pp. 43–50. Springer (2003), http://joye.site88.net/papers/BJ03isog.pdf 26

[18] Bundesamt für Sicherheit in der Informationstechnik: Elliptic Curve Cryptography. BSI TR-03111 version 2.10 (Jun 2012), https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TR03111/BSI-TR-03111_pdf.pdf?__blob=publicationFile, advises implementors to follow [48] 26

[19] Certicom Research: SEC 1: Elliptic Curve Cryptography (version 1.0) (Sep 2000), https://web.archive.org/web/20051101000000*/http://www.secg.org/download/aid-385/sec1_final.pdf 5

[20] Certicom Research: SEC 2: Recommended Elliptic Curve Domain Parameters (version 2.0) (Jan 2010), http://www.secg.org/sec2-v2.pdf 26

[21] Chou, T.: Sandy2x: New Curve25519 speed records. In: Dunkelman, O., Keliher, L. (eds.) SAC 2015. LNCS, vol. 9566, pp. 145–160. Springer (Aug 2016), https://www.win.tue.nl/~tchou/papers/sandy2x.pdf 6, 25, 28, 51, 55

[22] Costigan, N., Schwabe, P.: Fast elliptic-curve cryptography on the Cell Broadband Engine. In: Preneel, B. (ed.) AFRICACRYPT 09. LNCS, vol. 5580, pp. 368–385. Springer (Jun 2009), https://cryptojedi.org/papers/celldh-20090331.pdf 32

[23] Darrel Hankerson, Alfred J. Menezes, S.V.: Guide to Elliptic Curve Cryptography. Springer (2005), https://cdn.preterhuman.net/texts/cryptography/Hankerson,%20Menezes,%20Vanstone.%20Guide%20to%20elliptic%20curve%20cryptography%20(Springer,%202004)(ISBN%20038795273X)(332s)_CsCr_.pdf 26

[24] Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Transactions on Information Theory 22(6), 644–654 (1976), https://ieeexplore.ieee.org/abstract/document/1055638 5, 9

[25] Düll, M., Haase, B., Hinterwälder, G., Hutter, M., Paar, C., Sánchez, A.H., Schwabe, P.: High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers. Designs, Codes and Cryptography 77(2), 493–514 (Dec 2015), https://eprint.iacr.org/2015/343.pdf 25

[26] European Telecommunications Standards Institute (ETSI): TS 102 176-1 V2.0.0 (Nov 2007), https://www.etsi.org/deliver/etsi_ts/102100_102199/10217601/02.00.00_60/ts_10217601v020000p.pdf, advises implementors to follow FIPS 186-2 (predecessor of [52]) or [48] 26

[27] Faz-Hernández, A., López, J.: Fast implementation of Curve25519 using AVX2. In: Lauter, K.E., Rodríguez-Henríquez, F. (eds.) LATINCRYPT 2015. LNCS, vol. 9230, pp. 329–345. Springer (Aug 2015), https://link.springer.com/chapter/10.1007/978-3-319-22174-8_18 25, 58

[28] Ge, Q., Yarom, Y., Cock, D., Heiser, G.: A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. Journal of Cryptographic Engineering 8(1), 1–27 (Apr 2018), https://eprint.iacr.org/2016/613.pdf 4, 23

[29] Genkin, D., Valenta, L., Yarom, Y.: May the Fourth Be With You: A microarchitectural side channel attack on several real-world applications of Curve25519. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017. pp. 845–858. ACM Press (Oct / Nov 2017), https://eprint.iacr.org/2017/806.pdf 5

[30] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Comput. Surv. 23(1), 5–48 (Mar 1991), http://doi.acm.org/10.1145/103162.103163 34

[31] Hamburg, M.: Decaf: Eliminating cofactors through point compression. In: Gennaro, R., Robshaw, M.J.B. (eds.) CRYPTO 2015, Part I. LNCS, vol. 9215, pp. 705–723. Springer (Aug 2015), https://www.shiftleft.org/papers/decaf/decaf.pdf 5

[32] Hamburg, M.: Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625 (2015), http://eprint.iacr.org/2015/625 5

[33] Hasse, H.: Zur Theorie der abstrakten elliptischen Funktionenkörper III. Die Struktur des Meromorphismenrings. Die Riemannsche Vermutung. Journal für die reine und angewandte Mathematik (Crelle's Journal) 1936(175), 193–208 (1936) 13

[34] Hutter, M., Schwabe, P.: Multiprecision multiplication on AVR revisited. Journal of Cryptographic Engineering 5(3), 201–214 (Sep 2015), https://cryptojedi.org/papers/avrmul-20150101.pdf 38

[35] IEEE Computer Society Standards Committee. Working group of the Microprocessor Standards Subcommittee: IEEE standard for binary floating-point arithmetic. Institute of Electrical and Electronic Engineers (1985) 34

[36] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, vol. 1 (Sep 2016), https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf 51, 52

[37] Intel Corporation: Intel® 64 and IA-32 Architectures Optimization Reference Manual (Apr 2018), https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf 29, 38

[38] Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security 1(1), 36–63 (Aug 2001), https://link.springer.com/article/10.1007/s102070100002 27

[39] Karatsuba, A., Ofman, Y.: Multiplication of many-digital numbers by automatic computers. Dokl. Akad. Nauk SSSR 145(2), 293–294 (1962), http://www.mathnet.ru/php/getFT.phtml?jrnid=dan&paperid=26729&what=fullt&option_lang=eng 21

[40] Kaufmann, T., Pelletier, H., Vaudenay, S., Villegas, K.: When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In: Foresti, S., Persiano, G. (eds.) CANS 16. LNCS, vol. 10052, pp. 573–582. Springer (Nov 2016), https://infoscience.epfl.ch/record/223794/files/32_1.pdf 5

[41] Koblitz, N.: Elliptic curve cryptosystems. Math. Comp. 48, 209–209 (1987), https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866109-5/S0025-5718-1987-0866109-5.pdf 4, 9

[42] Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) CRYPTO'96. LNCS, vol. 1109, pp. 104–113. Springer (Aug 1996), https://www.paulkocher.com/TimingAttacks.pdf 22

[43] Langley, A., Hamburg, M., Turner, S.: Elliptic curves for security. RFC 7748 (Jan 2016), https://rfc-editor.org/rfc/rfc7748.txt 5

[44] Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), 515–534 (1982), https://www.math.leidenuniv.nl/~hwl/PUBLICATIONS/1982f/art.pdf 23

[45] Liu, F., Yarom, Y., Ge, Q., Heiser, G., Lee, R.B.: Last-level cache side-channel attacks are practical. In: 2015 IEEE Symposium on Security and Privacy. pp. 605–622. IEEE Computer Society Press (May 2015), http://palms.ee.princeton.edu/system/files/SP_vfinal.pdf 23

[46] Massolino, P.M.C., Renes, J., Batina, L.: Implementing complete formulas on Weierstrass curves in hardware. In: Carlet, C., Hasan, M.A., Saraswat, V. (eds.) SPACE 2016. LNCS, vol. 10076, pp. 89–108. Springer (2016), https://eprint.iacr.org/2016/1133.pdf 6, 53

[47] Maurer, U.M., Wolf, S.: The Diffie-Hellman protocol. Designs, Codes and Cryptography 19(2), 147–171 (Mar 2000), https://doi.org/10.1023/A:1008302122286 8

[48] Merkle, J., Lochter, M.: Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation. RFC 5639 (Mar 2010), https://rfc-editor.org/rfc/rfc5639.txt 26, 61, 62

[49] Miller, V.S.: Use of elliptic curves in cryptography. In: Williams, H.C. (ed.) CRYPTO'85. LNCS, vol. 218, pp. 417–426. Springer (Aug 1986), https://www.researchgate.net/profile/Victor_

Miller/publication/227128293_Use_of_Elliptic_Curves_in_Cryptography/links/0c96052e065c94b47c000000/Use-of-Elliptic-Curves-in-Cryptography.pdf 4, 9

[50] Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Math. Comp. 48, 243–264 (1987), https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025--5718-1987-0866113-7.pdf 4, 15

[51] National Institute for Standards and Technology (NIST): 186-2. Digital Signature Standard (DSS) (Jan 2000), https://csrc.nist.gov/publications/detail/fips/186/2/archive/2000-01-27 4, 5

[52] National Institute for Standards and Technology (NIST): 186-4. Digital Signature Standard (DSS) (Jul 2013), http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf 26, 62

[53] Perrin, T.: Subject: [curves] CryptoNote and equivalent points (May 2017), https://moderncrypto.org/mail-archive/curves/2017/000898.html 5

[54] Pohlig, S., Hellman, M.: An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance. IEEE Transactions on Information Theory 24(1), 106–110 (1978), https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1055817 13

[55] van de Pol, J., Smart, N.P., Yarom, Y.: Just a little bit more. In: Nyberg, K. (ed.) CT-RSA 2015. LNCS, vol. 9048, pp. 3–21. Springer (Apr 2015) 23

[56] Pollard, J.: Monte Carlo methods for index computation (mod p). Math. Comp. 32, 918–924 (1978), https://www.ams.org/journals/mcom/1978-32-143/S0025-5718-1978-0491431-9/S0025-5718-1978-0491431-9.pdf 14

[57] Renes, J., Costello, C., Batina, L.: Complete addition formulas for prime order elliptic curves. In: Fischlin, M., Coron, J.S. (eds.) EUROCRYPT 2016, Part I. LNCS, vol. 9665, pp. 403–428. Springer (May 2016), http://eprint.iacr.org/2015/1060 6, 26, 28, 41, 42, 45, 53, 54, 57, 72, 73

[58] Rivest, R.L., Shamir, A., Tauman, Y.: How to leak a secret. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 552–565. Springer (Dec 2001), https://people.csail.mit.edu/rivest/pubs/RST01.pdf 27

[59] Schnorr, C.P.: Efficient signature generation by smart cards. Journal of Cryptology 4(3), 161–174 (Jan 1991), https://www.researchgate.net/profile/Claus_Schnorr/publication/227088517_Efficient_signature_generation_by_smart_cards/

links/0046353849579ce09c000000/Efficient-signature-generation-by-smart-cards.pdf 5, 27

[60] Schnorr, C.P., Euchner, M.: Lattice basis reduction: Improved practical algorithms and solving subset sum problems. Mathematical programming 66(1-3), 181–199 (1994), https://www.researchgate.net/profile/Claus_Schnorr/publication/226499611_Lattice_Basis_Reduction_Improved_Practical_Algorithms_and_Solving_Subset_Sum_Problems/links/54231e520cf26120b7a6bb45/Lattice-Basis-Reduction-Improved-Practical-Algorithms-and-Solving-Subset-Sum-Problems.pdf 23

[61] Susella, R., Montrasio, S.: A compact and exception-free ladder for all short Weierstrass elliptic curves. In: Lemke-Rust, K., Tunstall, M. (eds.) Smart Card Research and Advanced Applications. LNCS, vol. 10146, pp. 156–173. Springer (2017), https://doi.org/10.1007/978-3-319-54669-8_10 26

[62] The Sage Developers: SageMath, the Sage Mathematics Software System (Version 7.5.1) (2017), https://www.sagemath.org 69

[63] Tibouchi, M.: Elligator squared: Uniform points on elliptic curves of prime order as uniform random strings. In: Christin, N., Safavi-Naini, R. (eds.) FC 2014. LNCS, vol. 8437, pp. 139–156. Springer (Mar 2014), https://eprint.iacr.org/2014/043.pdf 27

[64] Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on AES, and countermeasures. Journal of Cryptology 23(1), 62–74 (Jan 2010), https://www.cs.tau.ac.il/~tromer/papers/cache.pdf 23

[65] Yarom, Y., Falkner, K.: FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In: USENIX Security Symposium. vol. 1, pp. 22–25 (2014), https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-yarom.pdf 23

Appendix A

Instruction glossary

The tables on the next page contain general information about the instructions that are mentioned in this thesis. An instruction's operands can be:

- `xmm` for a 128-bit register,
- `ymm` for a 256-bit register, or
- `mX` for an X-bit memory location.

The µops column separates different µops with a comma. If a µop can be executed on different ports A and B, it is listed as `pAB`. Whenever a + is listed beside a µop, this means that this µop takes 2 cycles to execute.

The latencies in this table do not take cache latency into account, i.e. they only count the execution of the µops. Furthermore, when an instruction micro-fuses a separate arithmetic and load instruction, the latency is omitted. One can regard these micro-fused instructions as a separate arithmetic and load instruction.

Lastly, note that the throughput is listed in a *reciprocal* unit, i.e. cycles per instruction.

Table A.1: Integer arithmetic

mnemonic	operands	μops	latency	throughput^{-1}
vmovdqa	xmm, m128	p23	3	0.5
vmovdqa	m128, xmm	p23, p4	3	1
vp{and,or}	xmm, xmm	p015	1	0.5
vp{and,or}	xmm, m128	p015, p23		0.5
vps{l,r}lq	xmm, i	p0	1	1
vp{add,sub}q	xmm, xmm	p15	1	0.5
vp{add,sub}q	xmm, m128	p15, p23		0.5
vpmuludq	xmm, xmm	p0	5	1
vpmuludq	xmm, m128	p0, 23		1

Table A.2: Floating point arithmetic

mnemonic	operands	μops	latency	throughput^{-1}
vmovapd	ymm, m256	p23+	4	1
vmovapd	m256, ymm	p23, p4+	4	2
v{and,or}pd	ymm, ymm	p015, p5, p4+	1	1
v{and,or}pd	ymm, m256	p015, p5, p23+		1
v{add,sub}pd	ymm, ymm	p1	3	1
v{add,sub}pd	ymm, m256	p1, p23+		1
vmulpd	ymm, ymm	p0	5	1
vmulpd	ymm, m256	p0, p23+		1
vandpd	ymm, ymm	p5	1	1
vandpd	ymm, m256	p5, p23+		1
{add,sub}sd	xmm, xmm	p1	3	1
{add,sub}sd	xmm, m264	p1, p23		1
vmulsd	xmm, xmm	p0	5	1
vmulsd	xmm, m64	p0, p23		1

Appendix B

Curve verification with Sage

The Sage Mathematics Software System [62] was used to verify Barreto's curve. The script that was used is listed in Listing 6.

Listing 6: Curve13318.sage

```
#!/usr/bin/env sage
# *-* encoding: utf-8 *-*

"""
Find a value `b` for the curve `E : y^2 = x^3 - 3x + b` s.t.
the curve has a prime order and its twist also has a prime order.
"""

from itertools import chain, count

F = GF(2^255 - 19)

def plusmin(it):
    for x in it:
        yield x
        yield -x

for i,b in enumerate(plusmin(count())):
    percents = int(100.0 * i / float(2*13318))
    print('[% 3d%%] b = % 6d' % (percents, b))

    try:
        E = EllipticCurve(F, [-3, b])
    except ArithmeticError:
        continue # Curve has a singularity
    n = E.order()

    # Only accept curves of prime order
```

```
if not is_prime(n):
    continue

# Only accept if the twist is also of prime order
twist = E.quadratic_twist()
if not is_prime(twist.order()):
    continue

bits = int(log(float(E.order()))/log(2.0))
print('')
print('Found a good curve E : y^2 = x^3 - 3x + {}'.format(b))

# Start selecting random points for the rest of the search to find G
for x in plusmin(range(0, 256)):
    for y in range(256):
        try:
            G = E(x, y)
        except TypeError: # (x, y) is not a valid point
            continue
        print('  - G = {}'.format(G))
break
```

Appendix C

Renes-Costello-Batina addition formulas

Algorithm C.1 Exception-free point doubling formula for Barreto's curve, as described by [57, Algorithm 6], transcribed into SSA form. Each value v_i is computed on the corresponding line i in Algorithm 6 from [57].

procedure DOUBLE(X, Y, Z) ▷ Compute $(X_3 : Y_3 : Z_3) = 2(X : Y : Z)$

$v_1 \leftarrow X \cdot X$
$v_2 \leftarrow Y \cdot Y$
$v_3 \leftarrow Z \cdot Z$
$v_4 \leftarrow X \cdot Y$
$v_5 \leftarrow v_4 + v_4$
$v_6 \leftarrow X \cdot Z$
$v_7 \leftarrow v_6 + v_6$
$v_8 \leftarrow b \cdot v_3$
$v_9 \leftarrow v_8 - v_7$
$v_{10} \leftarrow v_9 + v_9$
$v_{11} \leftarrow v_{10} + v_9$
$v_{12} \leftarrow v_2 - v_{11}$
$v_{13} \leftarrow v_2 + v_{11}$
$v_{14} \leftarrow v_{12} \cdot v_{13}$
$v_{15} \leftarrow v_{12} \cdot v_5$
$v_{16} \leftarrow v_3 + v_3$
$v_{17} \leftarrow v_3 + v_{16}$
$v_{18} \leftarrow b \cdot v_7$
$v_{19} \leftarrow v_{18} - v_{17}$
$v_{20} \leftarrow v_{19} - v_1$
$v_{21} \leftarrow v_{20} + v_{20}$
$v_{22} \leftarrow v_{20} + v_{21}$
$v_{23} \leftarrow v_1 + v_1$
$v_{24} \leftarrow v_{23} + v_1$
$v_{25} \leftarrow v_{24} - v_{17}$
$v_{26} \leftarrow v_{25} \cdot v_{22}$
$v_{27} \leftarrow v_{14} + v_{26}$
$v_{28} \leftarrow Y \cdot Z$
$v_{29} \leftarrow v_{28} + v_{28}$
$v_{30} \leftarrow v_{29} \cdot v_{22}$
$v_{31} \leftarrow v_{15} - v_{30}$
$v_{32} \leftarrow v_{29} \cdot v_2$
$v_{33} \leftarrow v_{32} + v_{32}$
$v_{34} \leftarrow v_{33} + v_{33}$

$X_3 \leftarrow v_{31}$
$Y_3 \leftarrow v_{27}$
$Z_3 \leftarrow v_{34}$

end procedure

Algorithm C.2 Exception-free point addition formula for Barreto's curve, as described by [57, Algorithm 4], transcribed into SSA form. Each value v_i is computed on the corresponding line i in Algorithm 4 from [57].

procedure ADD(X_1, Y_1, Z_1, X_2, Y_2, Z_2)
▷ Compute $(X_3 : Y_3 : Z_3) = (X_1 : Y_1 : Z_1) + (X_2 : Y_2 : Z_2)$

$v_1 \leftarrow X_1 \cdot X_2$
$v_2 \leftarrow Y_1 \cdot Y_2$
$v_3 \leftarrow Z_1 \cdot Z_2$
$v_4 \leftarrow X_1 + Y_1$
$v_5 \leftarrow X_2 + Y_2$
$v_6 \leftarrow v_4 \cdot v_5$
$v_7 \leftarrow v_1 + v_2$
$v_8 \leftarrow v_6 - v_7$
$v_9 \leftarrow Y_1 + Z_1$
$v_{10} \leftarrow Y_2 + Z_2$
$v_{11} \leftarrow v_9 \cdot v_{10}$
$v_{12} \leftarrow v_2 + v_3$
$v_{13} \leftarrow v_{11} - v_{12}$
$v_{14} \leftarrow X_1 + Z_1$
$v_{15} \leftarrow X_2 + Z_2$
$v_{16} \leftarrow v_{14} \cdot v_{15}$
$v_{17} \leftarrow v_1 + v_3$
$v_{18} \leftarrow v_{16} - v_{17}$
$v_{19} \leftarrow b \cdot v_3$
$v_{20} \leftarrow v_{18} - v_{19}$
$v_{21} \leftarrow v_{20} + v_{20}$
$v_{22} \leftarrow v_{20} + v_{21}$
$v_{23} \leftarrow v_2 - v_{22}$
$v_{24} \leftarrow v_2 + v_{22}$
$v_{25} \leftarrow b \cdot v_{18}$
$v_{26} \leftarrow v_3 + v_3$
$v_{27} \leftarrow v_{26} + v_3$
$v_{28} \leftarrow v_{25} - v_{27}$
$v_{29} \leftarrow v_{28} - v_1$
$v_{30} \leftarrow v_{29} + v_{29}$
$v_{31} \leftarrow v_{30} + v_{29}$
$v_{32} \leftarrow v_1 + v_1$
$v_{33} \leftarrow v_{32} + v_1$
$v_{34} \leftarrow v_{33} - v_{27}$
$v_{35} \leftarrow v_{13} \cdot v_{31}$
$v_{36} \leftarrow v_{34} \cdot v_{31}$
$v_{37} \leftarrow v_{24} \cdot v_{23}$
$v_{38} \leftarrow v_{37} + v_{36}$
$v_{39} \leftarrow v_{24} \cdot v_8$
$v_{40} \leftarrow v_{39} - v_{35}$
$v_{41} \leftarrow v_{23} \cdot v_{13}$
$v_{42} \leftarrow v_8 \cdot v_{34}$
$v_{43} \leftarrow v_{41} + v_{42}$

$X_3 \leftarrow v_{40}$
$Y_3 \leftarrow v_{38}$
$Z_3 \leftarrow v_{43}$
end procedure

Appendix D

Benchmarking setup

This thesis has used two different approaches for benchmarking pieces of code. Section D.1 concerns the benchmarking of long, complete algorithms. This includes the cycle counts measured for Table 7.2. The other method—described by Section D.2—applies to smaller snippets of code.

All measurements were done with Turbo Boost disabled, all Hyper-Threading cores shut down, and with the CPU clocked at the maximum frequency.

The code for this work is published at `https://github.com/dsprenkels/curve13318`. A brief manual with information on how to reproduce the results is included.

D.1 Macro-benchmarks

To get accurate measurements of a complete run of the scalar-multiplication algorithm, we used the Intel `rdtsc` instruction. We measured the median of the calling overhead to be 58 cycles. This "blank" measurement is subtracted from every trail. For every measurement, we did 1000 trials and reported the *median* value.

Listing 7: Benchmarking large chunks of code.

```
#include <inttypes.h>
#include <stdio.h>
#include <assert.h>

extern int
crypto_scalarmult_curve13318_scalarmult(uint8_t*, const uint8_t*, const uint8_t*);

static __inline__ unsigned long long
rdtsc(void)
{
    unsigned hi, lo;
    __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
    return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
}

int
main(int argc, char *argv[])
{
    unsigned long long start, diff, blank = 58;

    uint8_t out[64] = {0};
    const uint8_t key[32] = {1};
    const uint8_t in[64] = {
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        179, 43, 106, 247, 206, 176, 201, 77, 137, 224, 122, 176, 76, 93, 29, 69,
        190, 137, 17, 103, 105, 172, 236, 172, 225, 72, 243, 7, 94, 128, 240, 17
    };

    for (unsigned int i = 0; i < 1000; i++) {
        start = rdtsc();
        int ret = crypto_scalarmult_curve13318_scalarmult(out, key, in);
        diff = rdtsc() - start - blank;
        printf("%llu\n", diff);
    }

    return 0;
}
```

D.2 Micro-benchmarks

A separate benchmarking method was used for smaller snippets of code, when the overhead of a function call would impact the accuracy of a measurement. This method has been used to measure small building blocks of the scalar-multiplication algorithm, like field multiplication, point doubling, etc.

The micro-benchmarks were constructed using the macros from Listing 8. Every benchmark is surrounded by `bench_prologue` and `bench_epilogue`. These snippets execute `rdtsc` to read the CPU time-stamp counter value before and after the call to the benchmark. Moveover, the `mfence` instruction in `bench_prologue` makes sure no out-of-order execution of benchmark code is started during the execution of the prologue; the `mfence` instruction in `bench_epilogue` makes sure that all instructions from the benchmark are retired before saving the new RDTSC value.

Intel also recommends to add `lfence` instructions before the execution of the `rdtsc` instruction, in order to serialize the instruction stream. We found, however, that this tweak was not needed to get consistent measurements in our case.

Lastly, the label `_bench_blank` holds the code for the *empty benchmark*. The empty benchmark measures the overhead of the benchmark framework itself. For each benchmark, the overhead is subtracted from the measured cycle count to get the cycle count of the actual benchmark body.

Listing 8: Macros for benchmarking small chunks of code.

```
; Macros for constructing benchmarks

%ifndef BENCH_ASM_
%define BENCH_ASM_

section .text

global _bench_blank, _bench_fns, _bench_names, _bench_fns_n

%macro bench_prologue 0
    push rbx
    push r12
    push r13
    push r14
    push r15
    vzeroupper
    rdtsc
    push rax
    push rbp
    mov rbp, rsp
    and rsp, -32
    mfence
%endmacro

%macro bench_epilogue 0
    mfence
    rdtsc
    mov rsp, rbp
    pop rbp
    pop rdx
    sub rax, rdx
    vzeroupper
    pop r15
    pop r14
    pop r13
    pop r12
    pop rbx
%endmacro

_bench_blank:
    bench_prologue
    bench_epilogue
    ret

%endif
```