Master's thesis

Magnus Hirth

Hardware Acceleration of Asymmetric Elliptic Curve Cryptography

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
July 2019

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

NORDIC
SEMICONDUCTOR

Magnus Hirth

Hardware Acceleration of Asymmetric Elliptic Curve Cryptography

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Per Gunnar Kjeldsberg
July 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Asymmetric cryptography, which is also known as public-key cryptography, provide algorithms for encryption and decryption of data, digital signatures and authentication. Compared with traditional asymmetric techniques, e.g. the RSA algorithm, the elliptic curve cryptography (ECC) achieves an equivalent level of security with smaller key sizes resulting in memory as well as bandwidth savings. Computational intensive operations like scalar multiplication on elliptic curves are required during the processing of ECC protocols. Using dedicated hardware units for these operations improves execution time in an energy efficient manner. Most implementations are based on high-end CPUs and GPUs and their use in mobile devices with limited power resources such as smartcards is untested.
This assignment is a continuation of an autumn project focusing on a theoretical and practical study of ECC, including experiments and profiling using Python and C-based code versions. Based on the results from these profiling experiments, this master thesis work will test the hypothesis that a hardware accelerated ECC implementation where the entire scalar multiplication operation is optimized to minimize memory transfers leads to a more energy efficient yet generic implementation.

NTNU

Abstract

Faculty Name
IE

Master Thesis

Hardware Acceleration of Asymmetric Elliptic Curve Cryptography

by Magnus HIRTH

With the great number of mobile, battery powered devices and IoT devices being developed, there is a need for efficient, energy effective cryptography. Elliptic curve cryptography (ECC) provides high security with small key size, and seems very well suited for use in embedded, low-power systems.

The mathematics of ECC are based on set theory, performing operations on elliptic curves, usually over finite prime fields or binary fields. The security of these mathematical operations are based on the Elliptic Curve Discrete Logarithm Problem.

This thesis has explored how to design a coprocessor for accelerating elliptic curve cryptography, based on the results from a pre-study. The coprocessor designed in the thesis, ECCo, was designed for use with the ARM CM33 processor. The CM33 provides a coprocessor interface for tight integration of coprocessors, which allows instructions to be issued to connected coprocessors from software. This motivated the design of an instruction set for the coprocessor.

For the design in this thesis the operations of modular addition, modular multiplication and integer division was implemented. The design used for testing consisted of a controller, register bank and arithmetic module. A pure software implementation of elliptic curve cryptography, *libecc*, was compared to the ECCo. Results showed that the hardware accelerated designed performed $3.8x$ - $27x$ times better than the pure software implementation.

Area estimates of the design was aquired through synthesis, using Questasim. The ECCo accounted for 45% of the area when synthesizing ECCo+CM33. The estimates showed that the ECCo area consumption was largely dominated by the divisor (73.18% of the total ECCo area), which was implemented using the SystemVerilog division operator, "/", and no optimization in synthesis. However, the atomic operations of ECC, Modular Multiplication and Modular Addition, only occupied 1.97% and 1.92%, respectively.

Preface

This thesis is a continuation of an autumn project which explored how an hardware accelerator of elliptic curve cryptography should be implemented in order to address the shortcomings of elliptic curve cryptography in software. Part of the theory is reused from the project. The project will from now on be referred to as the pre-study.

Contents

List of Abbreviations

CM33	ARM **C**ortex **M33**
CP	**C**o**P**rocessor
EC	**E**lliptic **C**urve
ECC	**E**lliptic **C**urve **C**ryptography
ECCo	**E**lliptic curve **C**ryptography **Co**processor
DMA	**D**irect **M**emory **A**ccess
DSP	**D**igital **S**ignal **P**rocessor
DUT	**D**esign **U**nder **T**est
FPU	**F**loating **P**oint **U**nit
FSM	**F**inite **S**tate **M**achine
ISA	**I**nstruction **S**et **A**rchitecture
LSB	**L**east **S**ignificant **B**it
MA	**M**odular **A**ddition
MM	**M**odular **M**ultiplication
MSB	**M**ost **S**ignificant **B**it
OOP	**O**bject **O**riented **P**rogramming
SIMD	**S**ingel **I**nstruction **M**ultiple **D**ata
SM	**S**calar **M**ultiplication
SV	**S**ystem**V**erilog
SVA	**S**ystem**V**erilog **A**ssertions
TLS	**T**ransport **L**evel **S**ecurity protocol

Chapter 1

Introduction

Today, many mobile and embedded devices are being used daily, and the number of such devices are ever increasing. Embedded devices are used in many applications where security is a concern, be it for a company or personal privacy: In hospitals, smart cards (banking, SIM, access control), mobile phones, wifi routers, etc. Many of these use battery powered devices, which in addition to security issues require low power solutions. This issue motivates the exploration of low-power implementation of cryptographic algorithms. A field of cryptography which seems suited for low-power applications is Elliptic Curve Cryptography (ECC), which was introduced in the 80s by Neil Koblitz [1] and Victor Miller [2]. It has gained popularity for desktop and server use, and many of the algorithms in the Transport Level Security protocol 1.3 (TLS 1.3) are elliptic curve (EC) algorithms.

In this thesis an implementation of a coprocessor for the ARM Cortex-M33 (CM33) designed for accelerating Elliptic Curve Cryptography (ECC) is designed and tested. The work is a continuation of the autumn project on hardware acceleration of ECC, which concluded that the optimal use of a hardware accelerator were to perform the entire operation of scalar multiplication (SM) in hardware. The implementation in this thesis aims at accelerating the entire SM in hardware, and taking advantage of the features the coprocessor interface of the CM33 provides.

In this thesis *cryptosystem* is used in the same way as defined in [3]: "A cryptosystem is a general term referring to a set of cryptographic primitives used to provide information security services. Most often the term is used in conjunction with primitives providing confidentiality, i.e., encryption."

Also, the term *big numbers* are used to refer to numbers of bit length longer than a processors word length.

1.1 Asymmetric Cryptography

Asymmetric cryptography, also known as public key cryptography, are cryptosystems which uses key pairs: A public key and a private key. The private key is only known to the owner, while the public key can be obtained by anyone without compromising the security of the system. The private key may be used to create a digital signature of a message, which allows anyone who got both the public key and the message to verify that the message has not been corrupted, or the private key may be used to decrypt a message which has been encrypted using the public key.

The security of public key cryptography systems relies on the private key being infeasible for an attacker to compute, but not impossible given infinite time and resources. That is, public key cryptosystems are *computationally secure* and it is infeasible for an attacker to compute the private key if it requires $\approx 10^{100}$ instructions [4].

Another very common type of cryptosystems are symmetric cryptography which uses a single shared key. These systems usually require smaller key sizes and have lower power consumption compared to public key systems [5][6]. Because of this symmetric cryptosystems are prefered when encrypting large amounts of data, but since they require the shared key to be shared over a secure channel it is usually not sufficient to rely solely on symmetric key cryptography. As a possible solution to this, a public key cryptosystem was introduced in 1976 by Whitfield Diffie and Martin E. Hellman [4] which enables two parties to securely share a key over an insecure channel, thus allowing secure communication through a combination of asymmetric and symmetric cryptosystems.

This combination of symmetric and asymmetric cryptosystems are now standard and the TLS 1.3 [7] standard describes a set of cryptosystems to use for secure communication over insecure channels. A number of these systems are public key systems and with the increasing demand for high security without reducing the efficiency of low power devices such as IoT [8][9] and mobile devices [10] it seems like a good incentive to explore the possibilities of accelerating public key cryptosystems.

Further more, TLS defines a number of ellptic curve (EC) cryptosystems to use. EC cryptosystems are systems that uses mathematics based on elliptic curves and have traits that makes them suited for use in resource limited environments, such as for IoT devices. ECC algorithms are often considered safer than their non-EC counterparts [1], and this safety is provided with smaller key sizes. The benefit of smaller key sizes is that less storage for the variables of the algorithm is required and less data needs to be transfered between devices. An efficient and good implementation of ECC algorithms could potentially benefit IoT devices by reducing power consumption while still maintaning high security.

1.2 Objective and Approach

The objective of this thesis is to explore how to design a coprocessor for accelerating elliptic curve cryptography, based on the conclusion of the pre-study [11]. This thesis tries to describe how such a coprocessor could be implemented, and implement as much of the proposed design as possible. The implemented design should be benchmarked and compared to the performance of a pure software implementation, to show what benefits a coprocessor could provide.

The design approach is to consider multiple possible designs before choosing one that is appropriate for the setup used in this thesis. All modules should be tested separately during the development process, using test data generated by software scripts, providing reliable test data.

1.3 Main Contributions

The main contributions of this thesis is the design of a flexible coprocessor aimed at accelerating elliptic curve cryptography, with the possibility of extending use to non-EC asynchronous cryptography. Detailing both the design and the design process.

Also, for this thesis a generic modular addition algorithm was designed.

A C library for big numbers was implemented. The library was designed for use with the elliptic curve coprocessor, supporting conversion to and from string representation and loading/storing to/from coprocessor registers.

1.4 Structure

Chapter 2 presents mathematical and other related background information necessary for the rest of the thesis. In Chapter 3 previous work relevant for this thesis is presented. Chapter 4 details the methodology and design choices of the coprocessor. Chapter 5 describes the implementation details of the design, and Chapter 6 presents the results of the thesis. Finally, Chapter 7 discusses thoughts on future work on the coprocessor, and Chapter 8 concludes the report.

Chapter 2

Background

This thesis is mainly concerned with elliptic curve cryptography, which are cryptosystems that uses mathematical operations on elliptic curves over finite fields. In order to give the reader a better understanding of these subjects this chapter gives a brief introduction into the mathematical field of set theory, focusing on the understanding of finite fields, and explaining the fundamentals of elliptic curves and related arithmetic operations on elliptic curves. Further, this chapter describes algorithms for implementation of modular arithmetic and elliptic curve operations in hardware, which are used later in the implementation of the coprocessor. Lastly this chapter also briefly describes the tools used.

2.1 Set theory

A *set* is (informally) a collection of objects (or elements). Sets are classified according to their mathematical properties. In this report the sets of interest are the finite fields, also called Galois fields, denoted by $GF(q)$ or $\mathbb{F}_q$. Finite fields are, without going into details, a set with a finite number, q, of elements where $q = p^k$ (p is prime and $k > 0$), on which the multiplication, addition, subtraction and division operations are defined [12, p.310]. In this thesis we are only interested in finite fields of integers, and, in particular, finite fields $\mathbb{F}_q$ containing all integers from 0 up to, but not including, q. For the rest of the thesis all fields will be assumed to be of this kind. These fields can be constructed with the modulo operator, because: $x = y \mod q$, where y can be any integer, x will always be in the range $0 \leq x < q$. A simple example of such a finite field is $\mathbb{F}_7$, shown in Equation 2.1. It is a field with 7 elements, and can be constructed with modulo 7.

$$\mathbb{F}_7 = \{0, 1, 2, 3, 4, 5, 6\} \tag{2.1}$$

If there exists a positive integer n such that $n \cdot a = 0$ for all $a \in \mathbb{F}$ then the smallest such number is called the *characteristic* of $\mathbb{F}$. If no such number exist then the characteristic of $\mathbb{F}$ is said to be zero [12, p.170]. In our example of $\mathbb{F}_7$ the characteristic is 7, since $7 \cdot a \equiv 0 \pmod 7$ for $a \in \mathbb{F}_7$. The characteristic of any finite field $GF(p^k)$ is p [12, p.311]. The size of a field, q, is also called the order of the field.

Of particular interest when working with elliptic curves are finite fields where $q = p^1$, *prime fields*, and finite fields where $q = 2^k$, *binary fields*.

2.1.1 Finite Field Arithmetic

For this report we are only concerned with finite fields, which implies that all arithmetic operations in field elements are, in fact, moldular arithmetic operations.

The reader is assumed to have basic knowledge of modular arithmetics, but examples of the basic operations on $\mathbb{F}_7$ are illustrated in Equations 2.2-2.5.

$$4+6=3 \tag{2.2}$$
$$1-5=3 \tag{2.3}$$
$$2\cdot 5=3 \tag{2.4}$$
$$5\cdot 4^{-1}=3 \tag{2.5}$$

Equations 2.2, 2.4 and 2.5 is 3 since $10 \equiv 3 \pmod 7$ and Equation 2.3 is 3 since $-4 \equiv 3 \pmod 7$. Equation 2.5 is an example of modular division which is the most complicated operation of the four. In order to perform modular division one needs to find the modular inverse of the divisor, which is why modular division often is written as in Equation 2.5, avoiding the division operator, "/", to avoid confusion with integer division. [13]

To find the modular inverse of a field element the Extended Euclidean Algorithm is used [14]. It is an extension to the Euclidean Algorithm which is an algorithm for finding the greatest common divisor of two numbers, a and b [15]. The extended algorithm can further be used to find two numbers, x and y, such that:

$$ax+by=\gcd(a,b) \tag{2.6}$$

For the level of details needed in this report we can now simply say that a and b has to be co-prime ($\gcd(a,b)=1$) and assign $b=q$, the field size. It can be shown that this leads to Equation 2.7.

$$ax \equiv 1 \pmod q \tag{2.7}$$

This allows us to find the inverse x of element a by solving for x ($x \in \mathbb{F}_q$). In Equation 2.5 $a=4$ and $q=7$, and so, we can find the inverse of 4 by solving for x in Equation 2.7:

$$\begin{gathered} 4x \equiv 1 \pmod 7 \\ \Downarrow \\ x=2 \end{gathered}$$

Equation 2.5 can then be explained by replacing 4^{-1} with the modular inverse of 4:

$$5\cdot 2 \equiv 3 \pmod 7$$

2.2 Elliptic Curves

Only elliptic curves over $\mathbb{F}_p$ and $\mathbb{F}_{2^m}$ are presented as these are the most common in ECC. Details will not be provided, only required conditions and a brief explanation of arithmetic on the curves are provided. A more detailed explanation can be found in [16]. The goal of this section is to get an intuitive understanding of what elliptic curves are, and the difference between continuous and discrete elliptic curves.

2.2.1 EC over $\mathbb{F}_p$

"Let $\mathbb{F}_p$ be a prime finite field so that p is an odd prime number, and let $a, b \in \mathbb{F}_p$ satisfy $4a^3 + 27b^2 \not\equiv 0 \pmod{p}$. Then an elliptic curve $E(\mathbb{F}_p)$ over $\mathbb{F}_p$ defined by the parameters $a, b \in \mathbb{F}_p$ consists of the set of solutions or points $P = (x, y)$ for $x, y \in \mathbb{F}_p$ to the equation:

$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{2.8}$$

together with an extra point $\mathcal{O}$ called the point at infinity." [16]

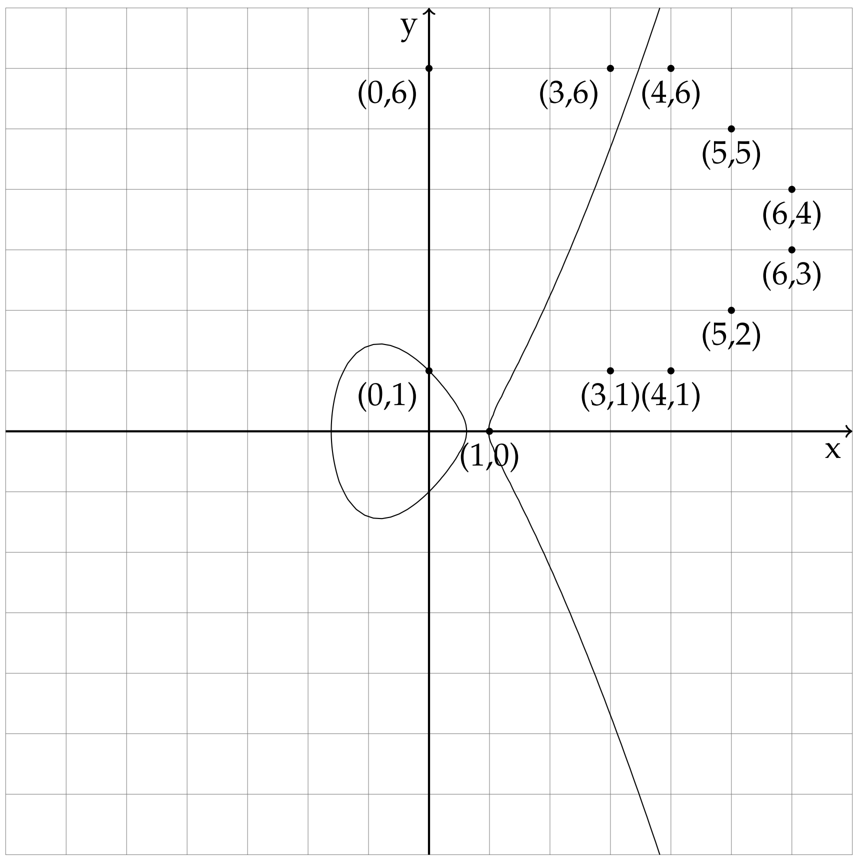

FIGURE 2.1: Illustration of $y^2 = x^3 - 2x + 1$ with the solutions to Equation 2.8 in $\mathbb{F}_7$ plotted.

Figure 2.1 illustrates the elliptic curve $y^2 = x^3 - 2x + 1$, $x \in [-7, 7]$. The continuous curve is the common way to illustrate an elliptic curve, over an infinite field. However, in cryptography finite fields are used, in which case there only exists discrete solutions to the elliptic curve, and for all of the solutions the x and y values must be in $\mathbb{F}_p$.

The discrete solutions to the elliptic curve (Equation 2.8) are plotted in Figure 2.1, and it is apparent that only the solutions $(0,1)$ and $(1,0)$ lie on the curve itself. This is because the x and/or y values resulting in the other solutions produced a LHS or RHS value in Equation 2.8 which were ≥ 7.

2.2.2 EC over $\mathbb{F}_{2^k}$

"Let $\mathbb{F}_{2^m}$ be a characteristic 2 finite field, and let $a, b \in \mathbb{F}_{2^m}$ satisfy $b \neq 0$ in $\mathbb{F}_{2^m}$. Then a elliptic curve $E(\mathbb{F}_{2^m})$ over $\mathbb{F}_{2^m}$ defined by the parameters $a, b \in \mathbb{F}_{2^m}$ consists of the set of solutions or points $P = (x, y)$ for $x, y \in \mathbb{F}_{2^m}$ to the equation:

$$y^2 + xy \equiv x^3 + ax^2 + b \pmod{p} \tag{2.9}$$

together with an extra point $\mathcal{O}$ called the point at infinity." [16]

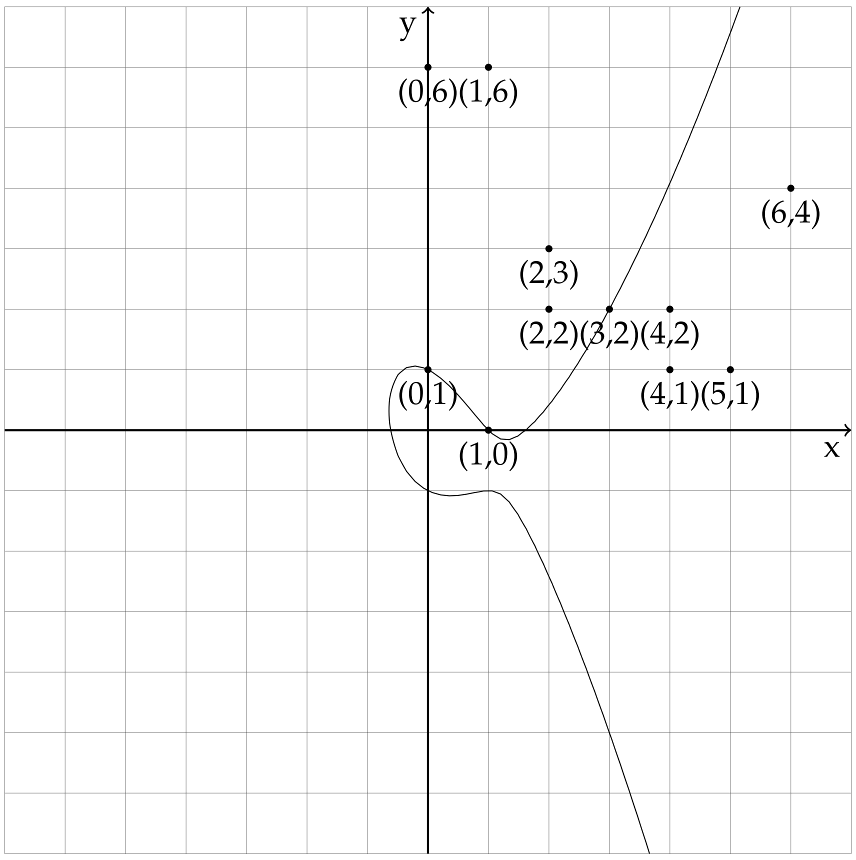

FIGURE 2.2: Illustration of $y^2 + xy = x^3 - 2x^2 + 1$ with the solutions to Equation 2.9 in $\mathbb{F}_7$ plotted.

Figure 2.2 illustrates the elliptic curve $y^2 + xy = x^3 - 2x^2 + 1$, $x \in [-7, 7]$. Also here both the continuous curve over an infinite field is plottet, along with the discrete solutions to the elliptic curve.

2.2.3 Point Arithmetics

In this report the arithmetic operations we are interested in on elliptic curves are point addition and point doubling. An intuitive geometric understanding of these operations where provided by Neal Koblitz [1], as illustrated in Figure 2.3.

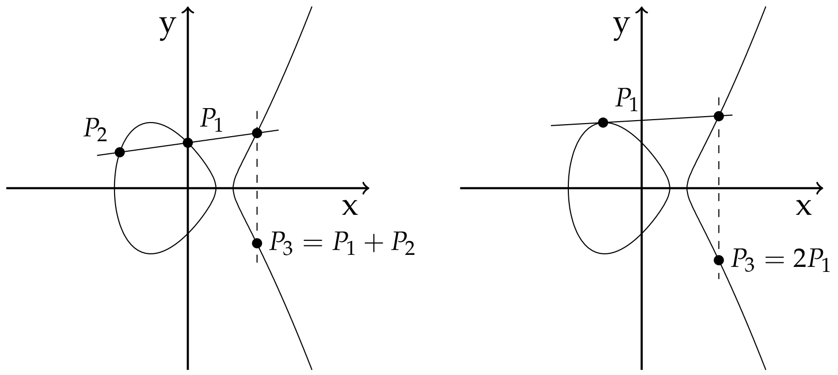

FIGURE 2.3: Illustration of elliptic curve point addition and doubling.

Let $P_1 = (x_1, y_1)$, $P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$ be points on an elliptic curve, where $P_3 = P_1 + P_2$. Draw a line $\overline{P_1P_2}$ through P_1 and P_2, then their sum P_3 will be the negative of the intersection of $\overline{P_1P_2}$ and the curve.

The following equations is a result of the observations from Figure 2.3, but there is not provided enough information to prove it. For a detailed explanation see [1].

$$x_3 \equiv -x_1 - x_2 + \alpha^2 \pmod{p} \tag{2.10}$$

$$y_3 \equiv -y_1 + \alpha(x_1 - x_3) \pmod{p} \tag{2.11}$$

where

$$\alpha = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if} P_1 \neq P_2 \\ \frac{3x_1^2 + a}{2y_1} & \text{if} P_1 = P_2 \end{cases} \tag{2.12}$$

In the case of elliptic curves over $\mathbb{F}_{2^m}$, when $P_1 \neq P_2$:

$$x_3 \equiv \alpha^2 + \alpha + x_1 + x_2 + a \pmod{p} \tag{2.13}$$

$$y_3 \equiv \alpha(x_1 + x_3) + x_3 + y_1 \pmod{p} \tag{2.14}$$

$$\alpha = \frac{y_1 + y_2}{x_1 + x_2} \tag{2.15}$$

and when $P_1 = P_2$:

$$x_3 \equiv \alpha^2 + \alpha + a \pmod{p} \tag{2.16}$$

$$y_3 \equiv x_1^2 + (\alpha + 1)x_3 \pmod{p} \tag{2.17}$$

$$\alpha = x_1 + \frac{y_1}{x_1} \tag{2.18}$$

Note that all of these operations require modular inversion for the division in the calculation of α, which is an expensive operation.

2.3 Scalar Multiplication

The central mathematical operation in all EC cryptosystems are the scalar multiplication, which is to multiply a scalar with a point on an elliptic curve. There are multiple different algorithms for performing a scalar multiplication. Most of these are based on the observation that any multiplication of a point and a scalar can be expressed as a combination of point additions and doublings, e.g. $11P = P + 2(P + 2(2P))$. There are many optimized algorithms for this, and in many applications it is desirable to use algorithms that have a constant execution time, for security reasons. However, in this thesis a basic algorithm, with varying execution time, is presented.

Algorithm 1 displays the pseudocode for this algorithm, called *Double-and-add (left-to-right)*.

Algorithm 1 Double-and-add (left-to-right) [17]

```
INPUT: Base point $P \in E_F$, scalar $k = (k_{t-1}, ..., k_0)_2$
OUTPUT: Point $Q = k \cdot P$
1: $R_0 \leftarrow \infty; R_1 \leftarrow P$
2: for $i$ from $t-1$ downto 0 do
3:     $R_0 \leftarrow 2R_0$
4:     if $k_i = 1$ then
5:         $R_0 \leftarrow R_0 + R_1$
6:     end if
7: end for
8: $Q \leftarrow R_0$
```

In this algorithm P is the base point on the curve, which is being multiplied with the scalar k, and Q is the resulting point on the curve. t is the bit length of k. What Algorithm 1 does is to iterate through all the bits in k, starting to the left (most significant bit). First R_0 is set to the point at infinity, and R_1 to the base point P. For each iteration it performes point doubling of R_0 (doubling of point at infinity returns the point at infinity), and if the current bit i is 1 then the point addition of R_0 and R_1 is stored in R_0 (addition of a point at infinity and a point P returns the point P).

This algorithm will perform t point doublings and, in worst case, t point additions.

2.4 Coordinate Systems

Elliptic Curves are often represented using affine coordinates, (x, y), as we have done so far, but there are several different coordinate systems with different attributes available. The purpose for using different coordinate systems is usually to increase performance. The way computation time is compared between coordinate systems is by calculating how many inversions (I), multiplications (M), and squarings (S) an addition or doubling operation require. From equations 2.10, 2.11 and 2.12 we see that in affine coordinates $(\mathcal{A})$ the computation times are $t(\mathcal{A} + \mathcal{A}) = I + 2M + S$ and $t(2\mathcal{A}) = I + 2M + 2S$. [18]

An alternative coordinate representation often used in practice is projective coordinates $(\mathcal{P})$. Here a point P is represented by a touple (X, Y, Z), where $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$. Using projective coordinates the computation time is $t(\mathcal{P} + \mathcal{P}) = 12M + 2S$ and $t(2\mathcal{P}) = 7M + 5S$. [18] The main motivation for using projective coordinates is reduced computation time since there is no inversion using projective coordinates, which is an expensive operation, as noted in Chapter 2.2.

There are other common alternatives for coordinates, as described in [3, p.86] and [18], but they will not be discussed here.

2.5 ECC Algorithms

Elliptic curve cryptography is commonly used for handshakes and digital signatures, such as in the Transport Layer Security (TLS) protocol 1.3 [7]. To add some perspective as to how the scalar multiplication is used in ECC this section will outline the Elliptic Curve Digital Signature Algorithm (ECDSA) [19].

The two parties involved will be refered to as *Alice* and *Bob* [20], where Alices private and public key are d_A and Q_A, respectively. Same for Bob, d_B and Q_B. For all ECC algorithms Alice and Bob have to agree on a set of parameters, D. In the case of $\mathbb{F}_p$ these parameters are $D = (q, a, b, G, n, h)$, where:

q Are the field order (Number of elements in the field. See Chapter 2.1)

a, b Are the elliptic curve coefficients (See Equation 2.8)

G Is the base point on the curve.

n Is the order of G; The smallest positive number such that $n \cdot G = \mathcal{O}$

h Is a number such that $h = \frac{n}{q}$

For $\mathbb{F}_{2^m}$ the parameters are $D = (m, f(x), a, b, G, n, h)$, where $f(x)$ is an irreducible binary polynomial of degree m specifying the representation of $\mathbb{F}_{2^m}$.

Algorithm 2 ECDSA signature generation [19]

```
INPUT: Domain parameters $D$, private key $d$ and message $m$
OUTPUT: Signature $(r, s)$
1: Select $k \in [1, n-1]$
2: Compute $kG = (x, y)$
3: Compute $r = x \mod n$. If $r = 0$ then go to step 1
4: Compute $e = H(m)$
5: Compute $s = k^{-1}(e + dr) \mod n$. If $s = 0$ the go to step 1
6: Return $(r, s)$
```

If Alice wants to send a message to Bob with a digital signature to verify that the message has not been corrupted during sending, she can use ECDSA, as shown in Algorithm 2. First, a random number k are multiplied with the base point G, and the resulting x value are used to compute r, one of the two parts of the signature. Then, a hash function $H(m)$ are used to produce a hash from the message. A hash function is a one-way function, where the message is very difficult to guess for anyone who knows the hash value. The hash and Alices private key is used to produce the second part of the signature s.

Algorithm 3 ECDSA signature verification [19]

```
INPUT: Domain parameters $D$, public key $Q$, message $m$ and signature $(r, s)$
OUTPUT: Acceptance or rejection of the signature
1: Verify that $r$ and $s$ are integers in the interval $[1, n-1]$ If any verification
   fails then return(“Reject the signature”).
2: Compute $e = H(m)$
3: Compute $w = s^{-1} \mod n$
4: Compute $u_1 = ew \mod n$ and $u_2 = rw \mod n$
5: Compute $X = u_1G + u_2Q$.
6: If $X = \infty$ then reject the signature
7: Convert the x-coordinate $x$ of $X$ to an integer $\overline{x}$ compute $v = \overline{x} \mod n$
8: If $v = r$ then accept the signature
```

When Bob then receives the message and the signature from Alice he can use Algorithm 3 to verify that the message has not been corrupted during sending, and be sure that it is the exact same message as Alice sent. The proof of the verification is out of scope for this thesis, but note that the verification requires two scalar multiplications.

Relating to the TLS 1.3 [7] standard: ECDH [4] [21] is often used to pass a symmetric key between Alice and Bob, along with an ECDSA-signature which verifies that the symmetric key has not been corrupted during transmission.

2.6 Tools

For simulation and synthesis the tool Questasim [22] is used. Questasim is developed by Mentor [23]. It is a high-performance tool supporting simulation, debugging and functional coverage using HDL languages such as VHDL [24], Verilog [25], and SystemVerilog [26], including SystemVerilogs object oriented features and SVA.

2.7 ARM Cortex M33

The Cortex-M33 [27] (CM33) is a processor developed my ARM [28]. It uses the ARMv8-M [29] instruction set architecture and is developed for embedded applications, allowing low power consumption while still providing efficient security and debug capabilities. It contains features such as an FPU and DSP with SIMD instructions.

The CM33 also features a coprocessor interface, which allows for tight integration of coprocessors and accelerators with the CM33. The coprocessors are accessible from software using assembly instructions provided in the ARMv8-M instruction set [29]:

CPD, CPD2 Coprocessor data processing instructions.

MCR, MCR2 32-bit data transfer to the coprocessor.

MRC, MRC2 32-bit data transfer to the CM33.

MCRR, MCRR2 64-bit data transfer to the coprocessor.

MRRC, MRRC2 64-bit data transfer to the CM33.

2.8 Hardware Acceleration

Hardware acceleration is commonly known as a method to speed up calculations by using specialized hardware, designed for a specific task, which often supplements a general purpose CPU [30]. A very common application of hardware acceleration is graphical processing units (GPUs), which are used in virtually every desktop. Other areas where hardware acceleration is common is in the field of AI and neural networks, and relevant to this thesis: cryptography. The security of cryptosystems are based on mathematics which often require heavy computations, which usually can greatly benefit from dedicated hardware.

2.9 libecc

libecc [31] is a library implementing EC mathematics hierarchically, as illustrated in Figure 2.4. The library provides separate modules which provides natural numbers arithmetics, field arithmetics (Chapter 2.1), elliptic curve

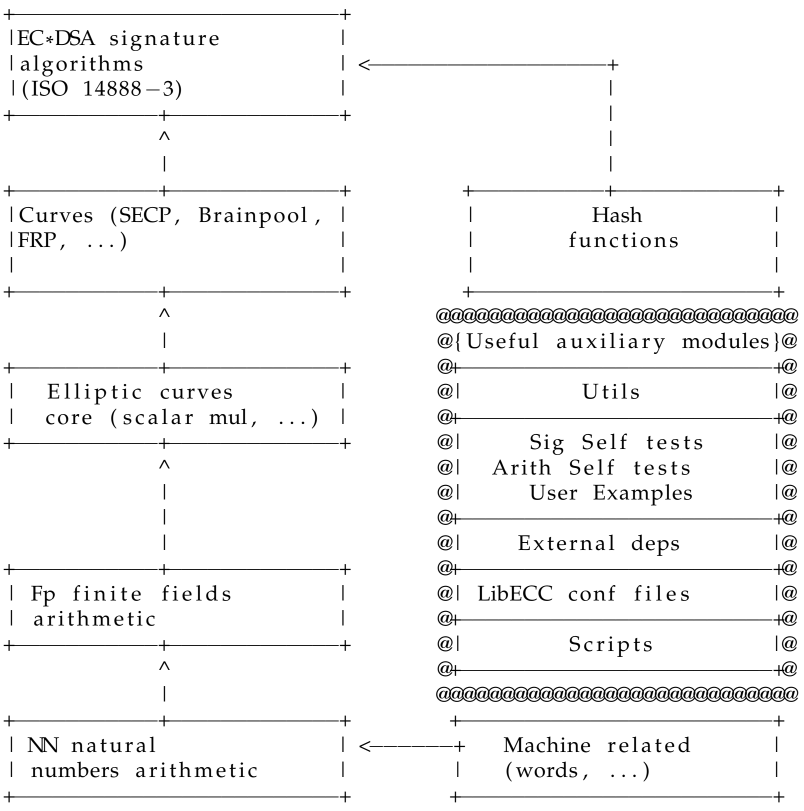
EC*DSA signature algorithms (ISO 14888-3)
Curves (SECP, Brainpool, FRP, ...)
Hash functions
{Useful auxiliary modules}
Utils
Elliptic curves core (scalar mul, ...)
Sig Self tests
Arith Self tests
User Examples
External deps
Fp finite fields arithmetic
LibECC conf files
Scripts
NN natural numbers arithmetic
Machine related (words, ...)

FIGURE 2.4: libecc architecture [31]

operations (Chapter 2.2), hardcoded values for curves, and implementation of the ECDSA algorithm (Chapter 2.5). Also, as seen in Figure 2.4, it provides implementation of some required hash function, self tests and some utilies, which will not be described here (see [31] for details).

Libecc does not actually implement multiple precision arithmetics but implements finite field and point arithmetics on big numbers up to a maximum integer width, which is determined at compile time. It uses projective coordinates, no dynamic memory allocation and is written without any dependencies, including the standard libc library.

2.10 Python

Python [32] is an interpreted, general-purpose programming language with dynamic type checking. Python has several interesting features which makes it flexible and easy to use, e.g. Python integers have an unlimited range [33] which makes handling of big numbers trivial. Internally Python represents big numbers as an array of fixed sized integers, but it is hidden when working with Python. Python also supports object oriented programming.

Chapter 3

Previous Work

In this chapter, existing algorithms for hardware implementations of modular addition and modular multiplication is presented. A thorough explanation and proof of correctness for these algorithms are not provided, see their respective references for more details.

An FPGA implementation of ECC coprocessors are presented, and finally the results from the pre-study is presented.

3.1 Modular Addition Implementation

Modular addition (MA) is the operation of calculating $S = X + Y \pmod{n}$, and is in effect the same operation for both addition and subtraction, if using 2's complement to represent signed numbers.

A straight forward way of implementing MA is to assume that $0 \leq A, B < n$ and do Algorithm 4 [34]. This algorithm may be performed in a single cycle with minimal control logic, depending on the timing constraints and the critical path through the additions on line 1 and 2.

Algorithm 4 Modular Addition Algorithm

```
INPUT: Addends $A$ & $B$, modulo $n$
OUTPUT: Sum $S$
1: Compute $S' = A + B$
2: Compute $S'' = S' - n$
3: if $S'' \geq 0$ then
4:     $S = S''$
5: else
6:     $S = S'$
7: end if
```

The operations on lines 1 and 2 are normal addition and subtraction, and the subtraction will require the 2's complement of n to either be calculated during operation or precomputed and be an input to the HW module. Algorithm 4 is restricted to positive numbers smaller than n.

Another method was proposed in [35]. Let $n < 2^k$ and $m = 2^k - n$, where k may be the word size of the system. It is assumed that $A, B < 2^k$. Modular addition can the be computed as in Algorithm 5.

Algorithm 5 Omura's Method, Modular Addition Algorithm

```
INPUT: Addends $A$ & $B$
OUTPUT: Sum $S$
1: Compute $S' = A + B$
2: if there is a carry then
3:     $S = S' + m$
4: else
5:     $S = S'$
6: end if
```

The value of m will need to either be computed during operation or pre-computed and be an input to the HW module. Here the additions in line 1 and 3 are normal additions. If there is no carry the result is $A + B$, which may be larger than n, in which case it will be reduced later. However, if there is a carry it will be ignored, which implies that $S' = A + B - 2^k$. And the correctness of the algorithm is given by:

$$\begin{aligned} S &= S' + m \\ &= (A + B - 2^k) + (2^k - n) \\ &= A + B - n \end{aligned}$$

Omura's algorithm is still restricted to positive numbers, but accepts addends greater than the modulo.

3.2 Modular Multiplication Implementation

Modular multiplication (MM) is the operation of calculating $P = A \cdot B \pmod{n}$. There are many algorithms for performing MM, many of which relies on alternative number representations for higher efficiency, such as the Montgomery modular multiplication [34]

An intuitive way of calculating MM is the multiply-and-divide method [34], illustrated in Algorithm 6.

Algorithm 6 Multiply and Divide Algorithm

```
INPUT: Multiplicand $A$, multiplier $B$, modulo $n$
OUTPUT: Product $P$
1: $P' = A \cdot B$
2: $P = P' \% n$
3: return $P$
```

This is, however, not an efficient implementation. The word size of P' will have to be twice that of A and B in order to avoid overflow, and the need to optimize the modulo reduction % will introduce unnecessary complexity to the design. Unless the product P' is needed an interleaving algorithm is usually to be preferred.

A basic interleaving algorithm is presented in Algorithm 7, where A and B are k-bit numbers between $0 \leq A, B < n$ of which A_i and B_i represents the ith bit.

Algorithm 7 Modular Multiplication Interleaving Algorithm

INPUT: Multiplicand A, multiplier B, modulo n
OUTPUT: Product P

```
1: $P = 0$
2: for $i = 0$ to $k - 1$ do
3:     $P = 2 \cdot P + A \cdot B_{k-1-i}$
4:     $P = P \% n$
5: end for
6: return $P$
```

Since $A, B, P < n$ it follows that

$$2P + A \cdot B_j \ \leq \ 2(n-1) + (n-1) = 3n - 3$$

Thus, maximum two subtractions are needed to reduce P to $0 \leq P < n$, which means the modulo operation in line 4 may be implemented as conditional subtractions.

Another efficient modular multiplication algorithm was proposed by Peter Montgomery in [36]. The result from the Montgomery algorithm is

$$P = A \cdot B \cdot r^{-1} \pmod n$$

where $A, B < n$ and $\gcd(n, r) = 1$. This adds overhead by requiring conversion of the result. The number of bits in A or B is less than k, and we take $r = 2^k$ [34]. The multiplication is shown in Algorithm 8.

Algorithm 8 Montgomery Modular Multiplication Algorithm

INPUT: Multiplicand A, multiplier B, modulo n
OUTPUT: Product $P = A \cdot B \cdot r^{-1} \pmod n$

```
1: $P = 0$
2: for $i = 0$ to $k - 1$ do
3:     $P = P + A_i \cdot B$
4:     if $P$ is odd then
5:         $P = P + n$
6:     end if
7:     $P = P/2$
8: end for
9: return $P$
```

Here, the division on line 7 is just a right shift, and the operations on line 3 and 5 can be combined: the LSB of P can be calculated before computing the sum on line 3.

Coprocessor	Modular Multiplication	Modular Addition	Modular Subtraction	Point Doubling	Point Addition	Scalar Multiplication
CP 1	100	-	-	-	-	-
CP 2	100	99	99	-	-	-
CP 3	147	146	146	899	801	-
CP 4	147	146	146	899	801	240000

TABLE 3.2: Execution times of coprocessors, in clock cycles.

3.3 FPGA Elliptic Curve Coprocessor

In [17] four different EC coprocessors were implemented and tested on an FPGA, each one implementing different arithmetic operations: CP 1 implemented modular multiplication (Chapter 2.1.1); CP 2 implemented modular multiplication, addition and subtraction (Chapter 2.1.1); CP 3 also implemented point doubling and addition (Chapter 2.2.3); and CP 4 implemented SM in addition to the arithmetic operations (Chapter 2.3).

The execution time of the implemented operations in each CP is listed in Table 3.2. The execution time is displayed in clock cycles.

The tests were performed using 256-bit values. The connected microcontroller used 8-bit word width, and the coprocessors were connected to and read the operands from RAM. Execution times includes reading operands and writing results.

3.4 Pre-Study

In the pre-study [11] possible partitioning between hardware and software for an ECC accelerator was explored. Using a pure software implementation of ECC profiling results were analyzed, trying to determine which parts of the software implementation could benefit the most from hardware acceleration.

The results showed that roughly 18.8% of execution time during testing was spent on managing the software implementation of big numbers: initialization, checking correct behavior, and handling number meta data. The conclusion was that as much as possible of an EC cryptosystem, in particular the scalar multiplication, should be performed by a coprocessor to reduce the overhead of dealing with big numbers in software.

Chapter 4

Methodology and Architecture Design

The main goal for this thesis is to implement an Elliptic Curve Cryptography Coprocessor (ECCo) which primary purpose is to accelerate the scalar multiplication in EC cryptosystems, as was the conclusion of the pre-study [11]. To perform the scalar multiplication the fundamental mathematical operations needed are modular multiplication and modular addition (Chapter 2.1.1), and integer division, when using affine coordinates (Chapter 2.4). These operations are enough to perform point doubling and point addition (Chapter 2.2.3), which allows implementation of an entire scalar multiplication (SM). The primary goal when designing the ECCo is therefore to implement the modular arithmetic operations.

The design of a coprocessor are potentially a complex and lengthy process. In the design process of the ECCo, to try to simplify this process, reusable design patterns was actively used: communication between submodules in the ECCo was generalized with clearly defined protocols; test data for all arithmetic operations was generated with a single Python script, utilizing Pythons OOP features; and a common testbench setup was used for all modules. These design patterns are further explained in their respective methodology and implementation chapters.

This chapter discusses which choices where made during the design and testing of the ECCo, and why these choices were made. Further, it highlights important aspects of the design process, specifically where and why reusable design patterns where used.

4.1 ECCo Design

The goal of the ECCo is to be able to perform scalar multiplication. Without any restrictions from any specific systems this allows for a number of different implementations.

1. It may be designed as a SM module which only performs the SM, similar to familiar division and multiplication modules. This module could be integrated in a processor, or connected to a buss, possibly using DMA to fetch operands.

2. It may be designed as a collection of modules, each implementing an atomic operation (i.e. modular addition or modular multiplication, see Chapters 3.1 - 3.2), similar to an FPU. This would be particularly suited for tight integration with a processor, and provide a flexible design which could be used for non-EC cryptosystems which also rely on finite field arithmetic, like RSA.

3. It may be designed as a combination of the previous solutions: Providing both the atomic operations and the SM operation. This could provide both a flexible design and an optimized SM, and would also be very well suited for tight integration with a processor.

The ECCo design in this thesis will interface with the ARM Cortex M33 (Chapter 2.7) for use from software. The CM33 provides a coprocessor interface which allows for tight integration of coprocessors and issuing opcodes to the coprocessor from software. Because of this, Solutions 2. and 3. are good choices. Ideally, Solution 3. would be chosen, but due to time limitations Solution 2. is the choice for this thesis. Allowing for estimates of SM speedup with and without the coprocessor by comparing speed of atomic operations in hardware and software. This minimal implementation will also be able to give an indication on how the size of the coprocessor will compare to that of the CM33 core itself.

Since the ECCo will be controlled from software through the coprocessor interface an instruction set has to be defined for the ECCo. The instruction set proposed in this thesis is presented in Chapter 5.1. The proposed instruction set includes more than the atomic operations and data transfer; It also includes logical, comparison, and shift operations. The pre-study concluded that an entire SM should be performed in the coprocessor in order to maximize the benefit of the coprocessor. By including these flow-control and common operations the ECCo will be able to perform an entire SM without datatransfer between the ECCo and CM33 during execution, even though it is being controlled from SW.

4.2 Choice of Alorithms

The two essential atomic operations are modular addition and modular multiplication, both of which can be implemented with multiple different algorithms (as described in Chapters 3.1 - 3.2). When choosing which algorithms to implement, this thesis chose the simplest algorithms in order to reduce time spent on implementation. Optimizations of the algorithms will be left for furute work.

The modular multiplication algorithm implemented is the *modular multiplication interleaving algorithm* (Algorithm 7), which is described in Chapter 3.2. This algorithm requires no overhead or added complexity from number conversion, but is not the most efficient algorithm and is not designed for security.

For the modular addition Algorithm 4 is the simplest presented algorithm, but it does not support negative numbers (i.e. no subtraction) nor intermediate sums greater than $2n$. To address these limitations an improved, generic version of the algorithm was designed. The new algorithm is described in Algorithm 9.

Algorithm 9 Generic Modular Addition Algorithm

```
INPUT: Addends $A$ & $B$, modulo $n$
OUTPUT: Sum $S$
1: Compute $S' = A + B$
2: while $S' \geq n$ do
3:     $S' = S' - n$
4: end while
5: while $S' < 0$ do
6:     $S' = S' + n$
7: end while
8: $S = S'$
```

This algorithm can handle both positive and negative numbers, and intermediate sums larger than $2n$. Notice that the **while** loops are mutually exclusive; After the intermediate sum, $S' = A + B$, has been calculated, S' will either be reduced or increased. Clearly, the **while** loops are not synthesizable. Details on the interpretation of this algorithm are presented in Chapter 5.

4.3 Interpretation of Algorithms

The mathematical foundation of ECC requires several abstract concepts and algorithms to be "translated" into hardware, i.e. the modulo operator; multiplication over a finite field (see Chapters 2.1.1 and 3); EC point addition (Chapter 2.2.3 and 3). There are often many ways of doing this, depending on the algorithm being implemented and system requirements. A significant decision when designing the implementation is the choice between sequential or combinatorial. Combinatorial designs are much more restricted by the clock frequency of the system, and can make it harder to meet timing requirements. For this thesis the sequential approach is preferred, and state machines has been designed to implement the chosen algorithms. The reason being that a sequential implementation is more similar to a state machine representation of the system, which makes it easier to reason about the behavior of the system.

4.4 Test Data

In order to verify the results from the implementations of arithmetic operations a set of known test data is required. In the pre-study [11] test data for the scalar multiplication and point arithmetic from reliable sources was used. This test data will be reused in this thesis. Test data for simpler operations (i.e. modular addition, division, etc.) is easy to generate using a Python script. Using a Python script will also allow generating more test data for SM and point arithmetic, since a Python implementation of these operations was written for the pre-study. The details of this script are described in Chapter 5, and full source code is listed in Appendix A.

Generation of test data contains a repeating pattern, regardless of what data is being generated: reading data from file, and writing properly formatted data to file. This can be handled by Pythons OOP features (see Chapter 5.5.5 and 2.10).

4.5 Verification

In order to both verify correct behavior and to speed up the development process, the entire ECCo and each sub-module are separately tested with a testbench verifying correct behavior. In the case of the arithmetic operations this includes checking results with test data, previously mentioned in Chapter 4.4.

Design of testbenches are a repeating process, which can be simplified by following a design pattern. During the development of ECCo the chosen pattern was:

- Each testbench consisted of a module, for instantiating and connecting the design under test (DUT); An interface connected to the DUT; A package with module specific parameters; A test program.
- All signals in the DUTs interface are connected to, and controlled by, the testbench. Allowing independent testing of all sub-modules.
- The testbench uses drivers and dummy implementation of modules to control the DUT. These dummies and driver can be reused between testbenches, and can utilize system verilogs OOP features.

4.6 Internal Interfaces

During design of the ECCo a repeating design question is how to communicate between sub-modules. The sub modules of the system are primarily modules implementing the operations defined by the instruction set, all of which may share a common communication protocol. Because of this all communication between sub-modules have been cleary defined using two interfaces: one for all communication with the register bank, another for all communication with the ECCo controller module. See Chapter 5.3 for further details.

4.7 Area Measurement

To aquire the results for area measurement the design was synthezised. The results presented are relative values, compared between synthesis of the CM33+ECCo and the CM33 only.

The speed results were measured during simulation, counting clock cycles used to execute benchmarking code of modular addition and modular multiplication, for both software and hardware implementations of those operations. Further details in Chapter 5.8.3.

Chapter 5

Implementation

This chapter describes implementation details about the work done for this thesis: proposed instruction set for the ECCo; the implementation of the ECCo and its integration with the CM33; testbench architecture and verification of the ECCo and its sub-modules; test data generation using a Python script; C implementation of the big numbers library, and the ECCo software wrapper; benchmarking of modular arithmetic operations, using the ECCo and a pure software implementation.

The logical, shift and comparison operations mentioned are not implemented in the ECCo for this thesis. The proposed instruction set includes these instructions, and discusses why they should be included in a future implementatin of an elliptic curve coprocessor.

5.1 ECCo Instruction Set

The ECCo instruction set was aimed at allowing software controlled implementations of SM, while reducing data transfer between between CM33 and ECCo. The instruction set designed in this thesis is listed in Table 5.2.

The connection between these instructions and the coprocessor instructions of the ARMv8-M instruction set (Chapter 2.7) is: the **MCRR** and **MRRC** are used to for the *Load* and *Store* instructions; the **CPD** and **CPD2** instructions are used for all other instructions, where the *opc1* and *opc2* arguments are opcodes for the issued operation (see [29] for description of assembly instructions).

In the instruction set the conditional operations are not explicitly listed, the reason being that all operations has a conditional conterpart, using the **CPD2** instruction.

While further evaluation about the necessity of all instructions are required, the instruction set proposed in this thesis are based on the following reasoning:

- The arithmetic instructions are fundamental for the SM (as discussed in Chapter 4).

- The logical instructions allows functionality like masking and setting registers to zero.

- Shift instructions allows efficient divide/multiply by 2, as required in algorithms like Montgomery (Algorithm 8)

Operation	**Parameter 1 (register)**	**Parameter 2 (register)**	**Parameter 3 (register)**
Modular Multiplication	*Multiplicand*	*Multiplier*	*Product*
Modular Addition	*Addend*	*Addend*	*Sum*
Integer division	*Dividend*	*Divisor*	*Quotient*
Negate 2's complement	*Operand*		*Result*
or	*Operand 1*	*Operand 2*	*Result*
and	*Operand 1*	*Operand 2*	*Result*
xor	*Operand 1*	*Operand 2*	*Result*
not	*Operand*		*Result*
Left shift	*Operand*	*Shift size*	*Result*
Logic right shift	*Operand*	*Shift size*	*Result*
Arithmetic right shift	*Operand*	*Shift size*	*Result*
Is zero	*Operand*		
Is equal	*Operand 1*	*Operand 2*	
Less than	*Operand 1*	*Operand 2*	
Greater than	*Operand 1*	*Operand 2*	
Load	*Offset*		*Index*
Store	*Offset*		*Index*
Increment	*Operand*		*Result*
Decrement	*Operand*		*Result*
Invert comparison			
Set signed bit	*Index*		
Unset signed bit	*Index*		

TABLE 5.2: Instruction set for elliptic curve coprocessor.

- Comparison and conditional instructions allow control flow.
- Increment and decrement are common operations. Since immediate values are not available for the coprocessor instructions this avoids the need of using a register for increment/decrement value.
- Inverting comparison allows for comparisons like *greater or equal to*, by inverting *Less than*.
- Set/Unset are required because the signed bit is not accessible through the data transfer instructions (see Chapter 5.4 for details).

An implementation of this instruction set will therefore allow an entire scalar multiplication to be performed in the ECCo, without data transfer during execution, while still being controlled by the CM33.

5.2 ECCo Architecture

The architecture of the ECCo were based on Solution 2 in Chapter 4.1. The architecture is illustrated in Figure 5.1.

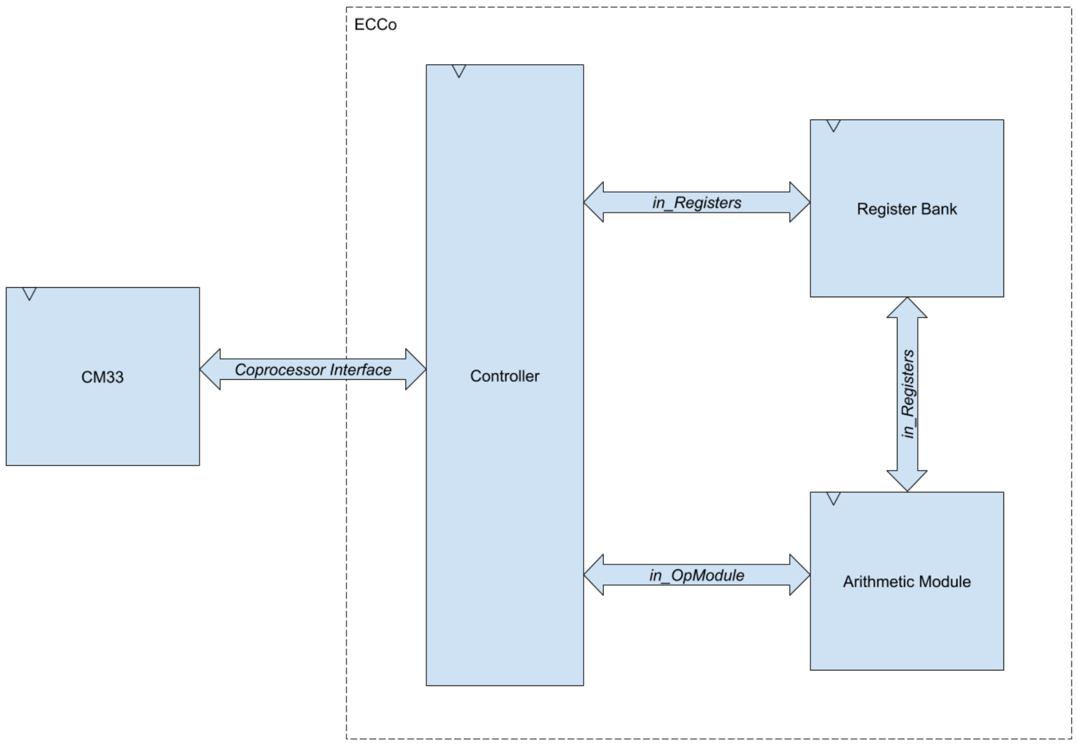

FIGURE 5.1: Architecture of ECCo, connected to the CM33 processor through the coprocessor interface.

The ECCo is connected to the CM33 through the coprocessor interface. Internally the sub-modules are connected through two interfaces, as discussed in Chapter 4.6. These interfaces are described in Chapter 5.3.

5.3 Internal Interfaces

There were used two internal interfaces in the design: *in_OpModule* which defines the protocol for issuing an operation to one of the operation-modules (a sub-module implementing one or more of the operations in the instruction set), and *in_Registers* which defines the protocol for reading from and writing to the register bank of the ECCo.

The *in_OpModule* interface uses a valid-ready protocol: when the sub-module is ready to accept a new operation a *ready* signal is asserted. An operation is issued by raising the *valid* signal, and it is accepted on the first clock cycle where *valid* and *ready* are both asserted. As long as *valid* is asserted all parameter values of the interface must be valid and stable. The interface also defines an *error* signal, which is asserted whenever an operation fails. The parameters of *in_OpModule* are:

op1Reg	Register index of operand 1
op2Reg	Register index of operand 2
resReg	Register index of result
opcode	Opcode for the requested operation

Figure 5.2 illustrates the protocol of the *in_Opmodule* interface. At *t*3 an operation is accepted. The controller issues another operation at *t*6, and has to wait, while keeping the parameters valid, until the previous operation has completed. At *t*9 the operation completed successfully, and the second operation is accepted. The second operation fails, as indicated by the *error* signal at *t*11. When the following, third, operation is accepted at *t*13, both the *ready* and *error* signals are deasserted. The SV interface implementation of *in_OpModule*s is listed in Appendix B.

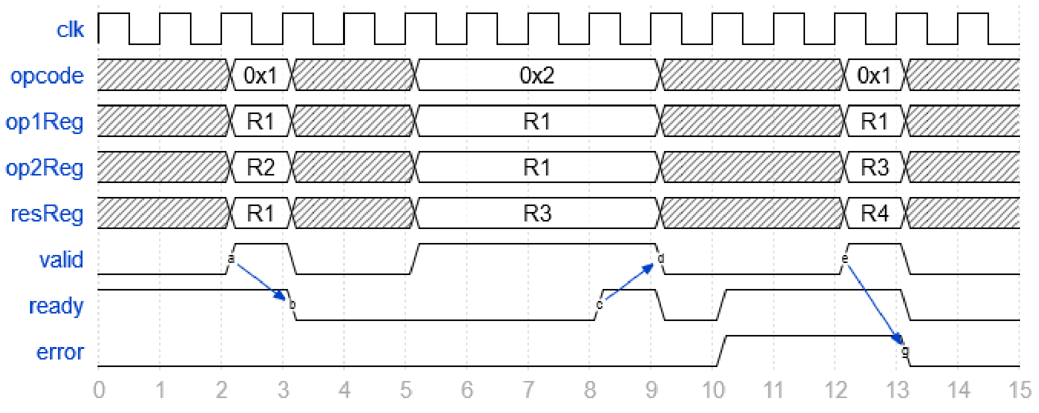

FIGURE 5.2: Illustration of *in_OpModule* communication protocol.

Because of this generalization of communication with all operation sub-modules, a common state machine is implemented as the controller in all of them, which is illustrated in Figure 5.3.

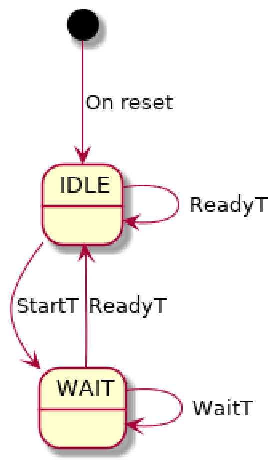

FIGURE 5.3: Illustration of FSM implementing the *in_OpModule* communication protocol.

In the state machine in Figure 5.3 **StartT**, **ReadyT**, and **WaitT** are names of possible transitions. This is because the output of the state machine are determined by both state and input. In *IDLE* the *ready* signal is asserted, and the value of *error* may be either 0 or 1. In *WAIT* both *ready* and *error* is always 0.

The *in_Registers* interface exposes all the registers directly, for reading. To write, the signals *enable*, *register*, and *data* are used, indicating when to enable writing, which register to write to, and the write data, respectively. The SV interface implementation of *in_Registers* are listed in Appendix B.

5.4 Register Bank

The register bank is a module containing 16 registers, which may be read from and written to. The choice of 16 registers was done based on a limitation from the CM33 which required the indexing of register using no more than 4 bits. However, it may not be necessary with these many registers to perform the SM. An evaluation of necessary number of registers are left for future work, considering both the area usage of the register bank and required number of registers for the SM implementation. All 16 registers are exposed for reading through the *in_Registers* interface. Writing is implemented following the *in_Registers* protocol.

The registers are of width $WORD\_WIDTH + 1$, e.g. if the ECCo is instantiated with a word width of 256-bit the word width of the registers will be 257-bit. The reason for this is that parameter values from standards such as [37] and [38] require $WORD\_WIDTH$-bits to represent positive values. Because of this the signed bit of registers are manipulated through dedicated instructions, to avoid using a 64-bit data transfer to access the signed bit.

Register Name	Register Index	Writable	Readable
CR0	0	X	X
CR1	1	X	X
...	...	...	...
CR13	13	X	X
Modulo Register	14	X	X
Status Register	15		X

TABLE 5.4: List of ECCo registers.

Table 5.4 lists all registers in the register bank. There is only two non-general registers: the modulo register and the status register. The modulo register is used for storing the modulo during modular arithmetic operations. The status register is read-only (all writing to it is done inside the register bank) and contains information about the current status of the ECCo:

Bit 0 Comparison result bit.

Bit 1-15 Active bits. These are reserved for future use in an asynchronous design, for indicating which operation modules are currently working and which are idle.

Bit 16-30 Signed bits. The signed bits of register 0-14, respectively.

Bit 31- Unused.

5.5 Arithmetic Module

The arithmetic operations sub-module is implemented as a controller implementing the *in_OpModule* protocol and wrapping the modules implementing each individual arithmetic operation: negation, integer division, modular addition, and modular multiplication. In Figure 5.4 the block diagram of the arithmetic module are shown. The arithmetic controller implements the *in_OpModule* FSM, as illustrated in Figure 5.3.

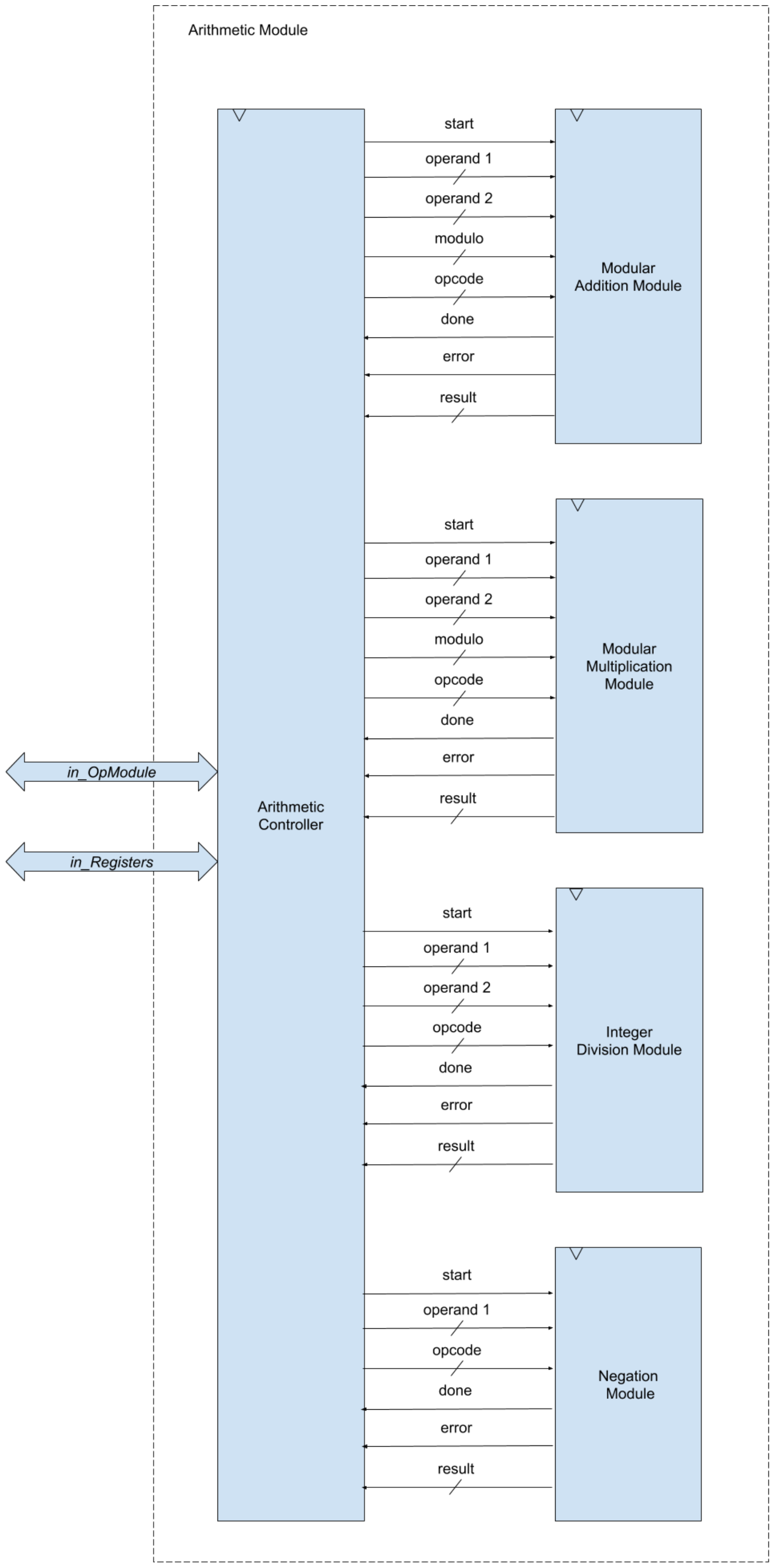

FIGURE 5.4: Block diagram of arithmetic module.

5.5.1 Negation

The negation operation is a single cycle operation which is straight forward to implement, and performs a 2's complement negation of the operand. It is continually calculated:

```
      assign res = ~(operand) + 1;
```

5.5.2 Integer Division

The integer division is a necessary operation when using Affine coordinates, but its implementation is not very interresting in regards to the ECCo. Therefore, it was initially implemented using an opensource design from Open-Cores [39]. However, this design did not function properly and instead integer division was implemented using the SystemVerilog division operator, "/".

It is also a single cycle operation, but requires divide-by-zero detection and handling of negative numbers: If the divisor and/or dividend is negative its positive 2's complement is used in the division and the sign of the result is calculated using basic algebra rules, as shown in Listing 5.1.

```
// MSB of dividend (op1) and divisor (op2)
logic msbOp1, msbOp2;
// Internal signals
logic [WORD_WIDTH:0] intOp1;
logic [WORD_WIDTH:0] intOp2;
logic [WORD_WIDTH:0] intRes;

// The division is continuously calculated.
assign divideByZero = (op2 == 0);
assign intRes = intOp1 / intOp2;
assign msbOp1 = op1[WORD_WIDTH];
assign msbOp2 = op2[WORD_WIDTH];

always_comb begin
  intOp1 = op1;
  intOp2 = op2;
  if ( msbOp1 && msbOp2 ) begin
    intOp1 = (~op1) + 1;
    intOp2 = (~op2) + 1;
  end
  else if ( msbOp1 )
    intOp1 = (~op1) + 1;
  else if ( msbOp2 )
    intOp2 = (~op2) + 1;
end

always_ff @(posedge ck)
  res <= (msbOp1 ^ msbOp2) ? (~intRes) + 1 : intRes;
```

LISTING 5.1: Division SV implementation.

5.5.3 Modular Addition

The modular addition is implemented using Algorithm 9, designed for this thesis, as discussed in Chapter 4.2. This algorithm is interpreted as illustrated by the FSM in Figure 5.5, and the datapath in Figure 5.6. The transitions in the illustration are referred to by name.

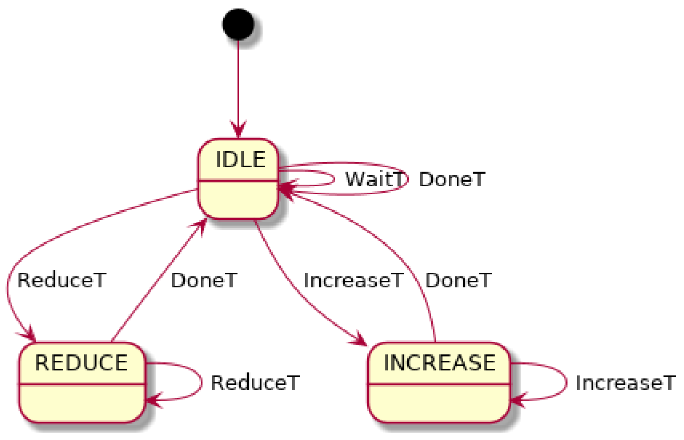

FIGURE 5.5: FSM interpretation of Generic Modular Addition Algorithm.

DoneT Transition to *IDLE* when an addition has finished. Asserting *done* for one cycle.

WaitT Transition in *IDLE* when not performing an operation.

ReduceT Transition to *REDUCE* when the intermediate sum is greater than the modulo, and need to be reduced to $0 \leq Sum < Modulo$.

IncreaseT Transition to *INCREASE* when the intermediate sum is less than 0, and need to be increased to $0 \leq Sum < Modulo$.

If initially: $op1 + op2 < mod$ then the calculation only takes one cycle to complete, or else *op1 mux* selects the intermediate result as operand 1 and *op2 mux* selects either *mod* or $-mod$ as operand 2, depending on if the state is *INCREASE* or *REDUCE*, respectively. In worst case the addition could take $2^{WORD\_WIDTH} - 1$ cycles to perform, calculating $((2^{WORD\_WIDTH} - 1) + 0)\ \%\ 1$.

5.5.4 Modular Multiplication

The modular multiplication is implemented using the Algorithm 7, as discussed in Chapter 4.2. This algorithm is interpreted as illustrated by the FSM in Figure 5.7, and the datapath in Figure 5.8. The transitions in the illustration are referred to by name.

DoneT Transition to *IDLE* when an multiplication has finished. Asserting *done* for one cycle.

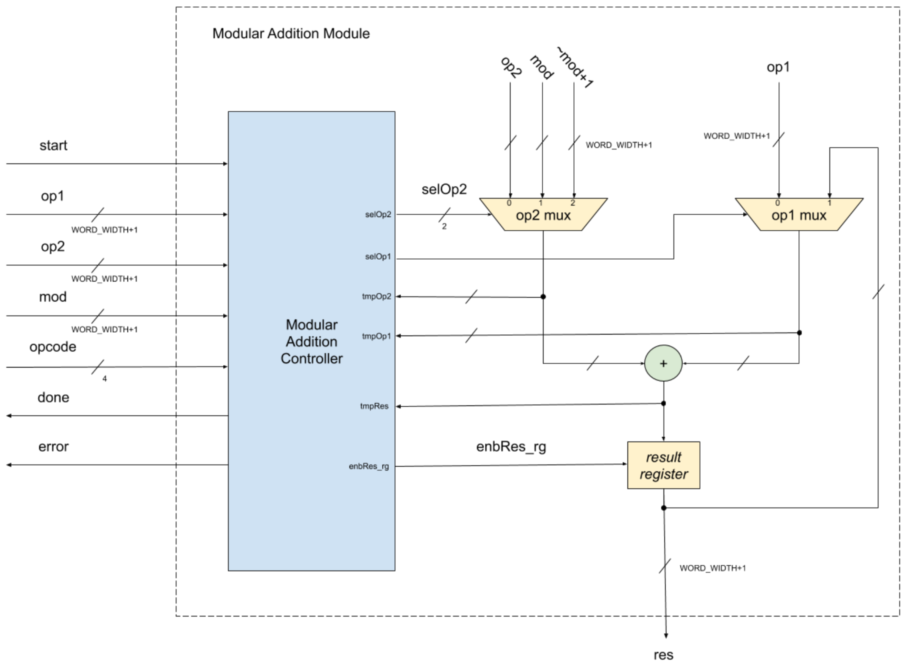

FIGURE 5.6: Block diagram of modular addition module.

WaitT Transition in *IDLE* when not performing an operation.

AddT Transition to *ADD* when calculating the sum of $2 \cdot P + A \cdot B_{k-1-i}$ (as described in Chapter 3.2).

ReduceT Transition to *REDUCE* when the intermediate sum is greater than the modulo, and need to be reduced to $0 \leq Sum < Modulo$.

ReduceDoneT Transition to *REDUCE_DONE* when the intermediate sum is greater than the modulo, and need to be reduced to $0 \leq Sum < Modulo$, before finishing to operation.

The modular multiplication always has an execution time of at least *WORD_WIDTH* cycles since it has to iterate through all bits of *op2*, except the signed bit. None of *op1*, *op2*, or *mod* are allowed to be negative. The emphpartial product mux selects the current value of $A \cdot B_{k-1-i}$. *op1 mux* and *op2 mux* selects whether to calculate $2 \cdot P + A \cdot B_{k-1-i}$ or to reduce the intermediate result.

5.5.5 Test Data

Test data was generated using a python script, which was written with an architecture as illustrated in Figure 5.9. The test data solutions are created by python operators, as shown in Listing 5.2.

```python
def modular_addition(op1: int, op2: int, mod: int) -> int:
    return (op1 + op2) % mod

def modular_multiplication(op1: int, op2: int, mod: int) ->
   int:
```

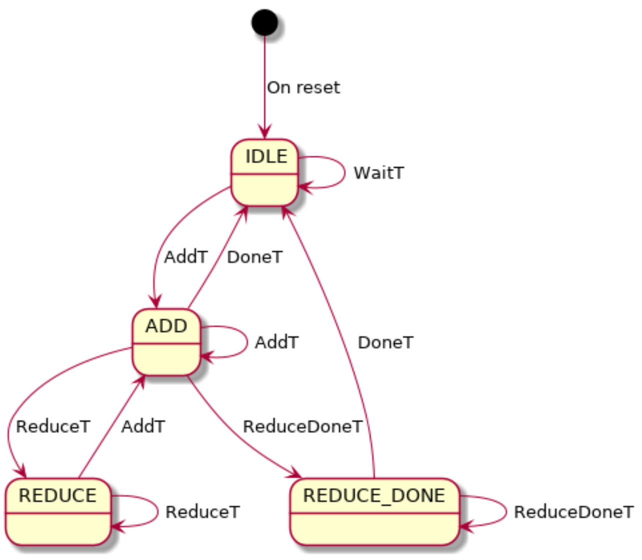

FIGURE 5.7: FSM interpretation of Multiply and Divide Algorithm.

```
    return (op1 * op2) % mod

def integer_division(op1: int, op2: int) -> int:
    if op1 < 0 and op2 < 0:
        res = abs(op1) // abs(op2)
    elif op1 < 0:
        res = -(abs(op1) // op2)
    elif op2 < 0:
        res = -(op1 // abs(op2))
    else:
        res = op1 // op2
    return res
```

LISTING 5.2: Test data solution calculations.

Notice the integer division // does not handle division of negative numbers correctly. Instead any negative numbers are negated, and basic algebra rules are used to determine the sign of the result, just as it is implemented in hardware.

The script source code is listed in Appendix A. Test data values used for verification are listed in Appendix C.

5.5.6 Verification - Arithmetic Module

The arithmetic module was tested using a TB design as illustrated in Figure 5.10. The test program communicates with the arithmetic module through an *in_OpModule* driver, and controls and verifies the register content during testing through a dummy register bank, connected to the arithmetic module.

During testing the values listed in Appendix C were used to verify correct results from arithmetic operations.

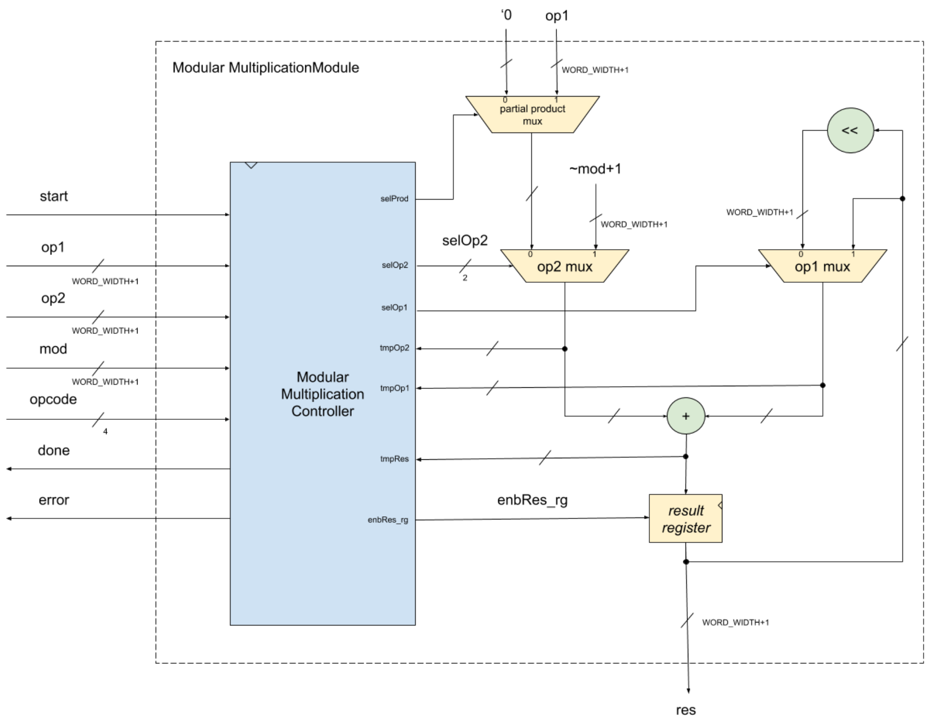

FIGURE 5.8: Block diagram of modular multiplication module.

5.6 Controller Module

The controllers primary purpose is to handle communication with the CM33 using the coprocessor interface, the FSM in Figure 5.11 illustrates the implemented state machine which does this. This is a synchronous design: the controller will wait for any multicycle operation to finish before signaling to the CM33 that it is ready to accept further instructions.

The outputs of the FSM is the coprocessor interface signals *valid* and *error*, and an internal *valid*, which are used in the *in_OpModule* interface. The transitions in the illustration are referred to by name. The output signals of the FSM are determined by both state and input, easiest described as the set of all possible transitions:

RyT - ready transition Transition to *READY*, with *ready* asserted and *error* deasserted, waiting for an instruction to be issued.

ET - error transition Transition to *READY*, with both *ready* and *error* asserted. May be from an write error, read error, data processing error or an invalid instruction being issued.

WaT - wait transition Transition to *WAIT* when *valid* is asserted and a data processing operation is issued.

WaWT - wait wait transition Transition to *WAIT*, from *WAIT*, while current data processing operation is not yet finished.

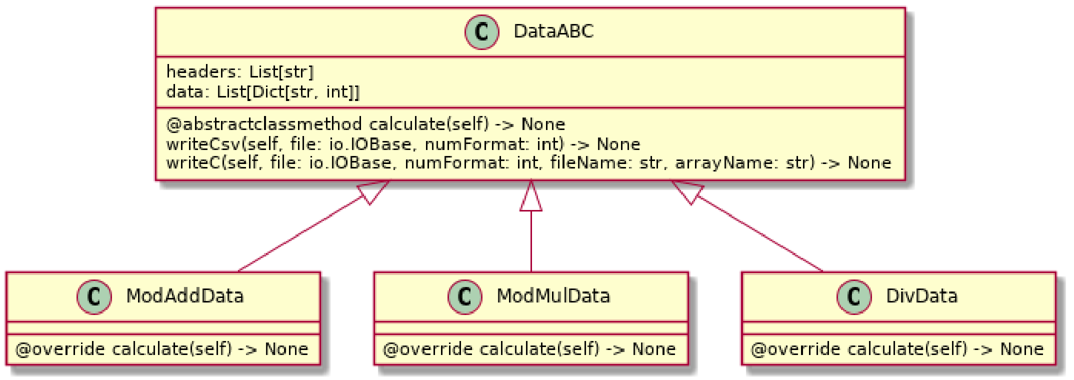
DataABC
headers: List[str]
data: List[Dict[str, int]]
@abstractclassmethod calculate(self) -> None
writeCsv(self, file: io.IOBase, numFormat: int) -> None
writeC(self, file: io.IOBase, numFormat: int, fileName: str, arrayName: str) -> None
ModAddData
@override calculate(self) -> None
ModMulData
@override calculate(self) -> None
DivData
@override calculate(self) -> None

FIGURE 5.9: Class diagram of python script generating test data.

WaRT - wait ready transition Transition to *WAIT*, from *WAIT*, when a data processing operation finished successfully and *valid* is asserted, requesting a new data processing operation immediately.

WaET - wait error transition Transition to *WAIT*, from *WAIT*, when a data processing operation finished with error and *valid* is asserted, requesting a new data processing operation immediately.

ReT - read transition Transition to *READ*, when the processor wants to read from a coprocessor register.

ReRT - read ready transition Transition to *READ*, from *WAIT*, when a data processing operation finished successfully and *valid* is asserted, requesting a data transfer operation (read) immediately.

ReET - read error transition Transition to *READ*, from *WAIT*, when a data processing operation finished with error and *valid* is asserted, requesting a data transfer operation (read) immediately.

WrT - write transition Transition to *WRITE*, when the processor wants to write to a coprocessor register.

WrRT - write ready transition Transition to *WRITE*, from *WAIT*, when a data processing operation finished successfully and *valid* is asserted, requesting a data transfer operation (write) immediately.

WrET - write error transition Transition to *WRITE*, from *WAIT*, when a data processing operation finished with error and *valid* is asserted, requesting a data transfer operation (write) immediately.

5.6.1 Verification - Controller Module

The testbench setup for the verification of the controller module is illustrated in Figure 5.12.

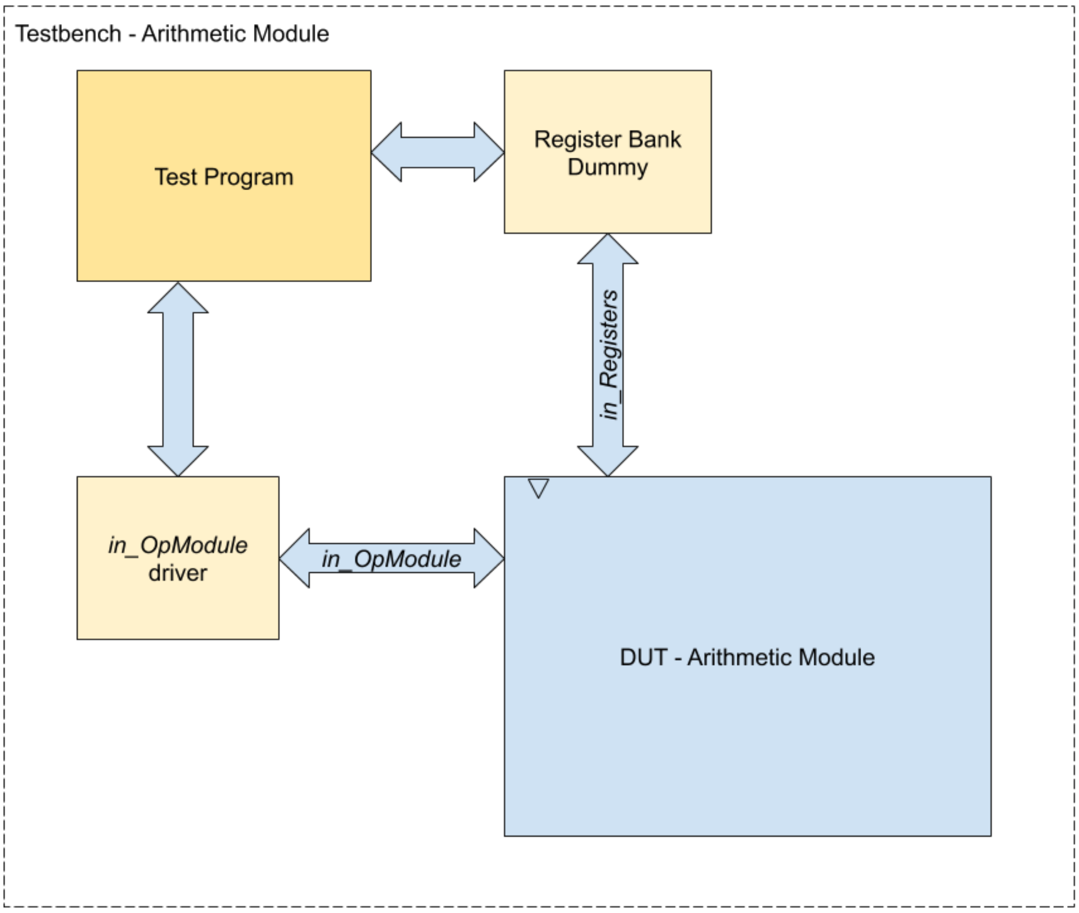

FIGURE 5.10: Block diagram of Arithmetic Module TB.

Operation module dummies for the arithmetic, logical, comparison and shift modules are connected to the controller, and controlled by the test program. A dummy register bank is connected to the controller, and the controller is tested using a coprocessor interface driver for communication.

5.7 Verification - ECCo

The testbench setup for verification of the entire ECCo is illustrated in Figure 5.13.

A coprocessor interface driver is used to communicate with the ECCo, and the test values from Appendix C are used to check for correct behavior of the implemented operations.

5.8 Software

For this thesis three software components were implemented: a wrapper for the coprocessor interface instructions; a big number library for use with the ECCo; and a benchmarking program.

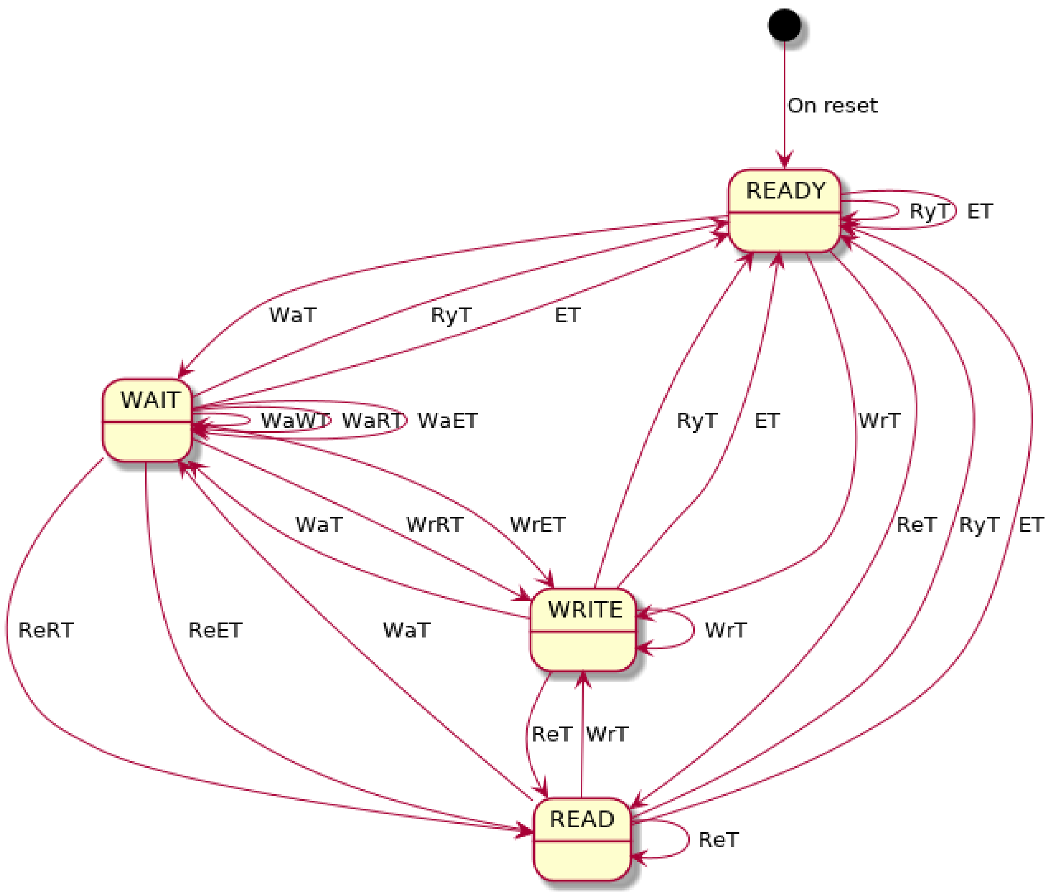

FIGURE 5.11: FSM of ECCo controller module.

The big number library and ECCo wrapper were used to verify that communication with the ECCo using the coprocessor interface was working as expected. To verify correct behavior of the ECCo controller and the implemented operations the test data form Appendix C were used. The source code of the test programs used for verification are listed in Appendix F.

5.8.1 ECCo Wrapper

The ECCo wrapper was implemented to simplify calling the ECCo from C using the coprocessor interface. The coprocessor instructions of the ARMv8-M instruction set have to be called from assembly, using string literals to refer to coprocessor registers and opcodes. Therefore a series of macros were created for all the instructions in the proposed instruction set (Table 5.2). The code for the wrapper is listed in Appendix D.

5.8.2 Big Number library

When using the ECCo some minor handling of big numbers in software are still required. For this a big number library was implemented for use with the ECCo. The functionality it provided was:

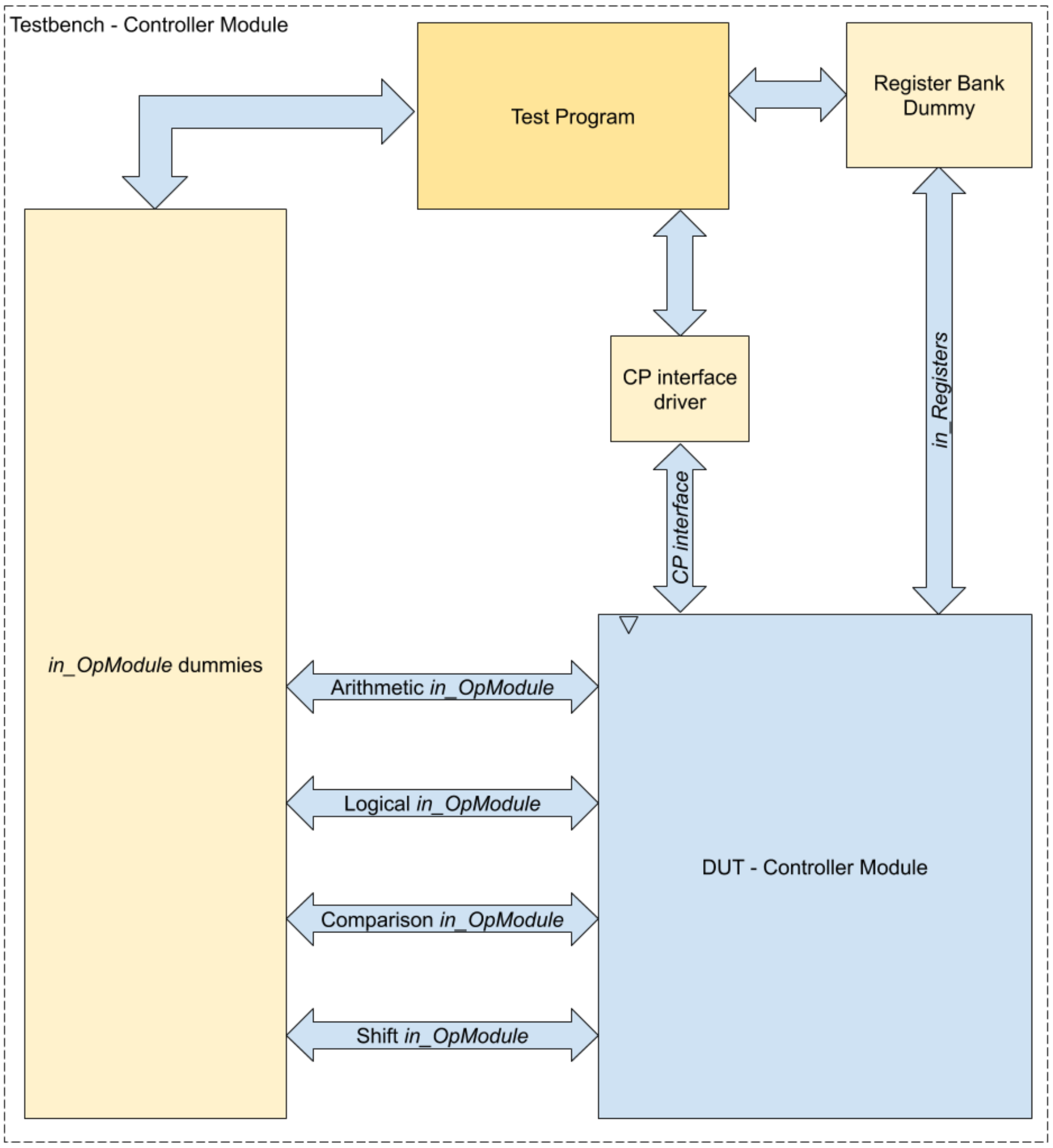

FIGURE 5.12: Testbench setup for verification of the controller module.

- Converting to and from number strings on hexadecimal format.
- Comparing two numbers.
- Loading a number to an ECCo register.
- Storing a number from an ECCo register.
- Some other convenient functionality.

The source code for the big number library is listed in Appendix E.

5.8.3 Benchmark Software

For benchmarking the pure software implementation of ECC, ANSSI libecc (Chapter 2.9), were compared to the ECCo. The benchmarked operations

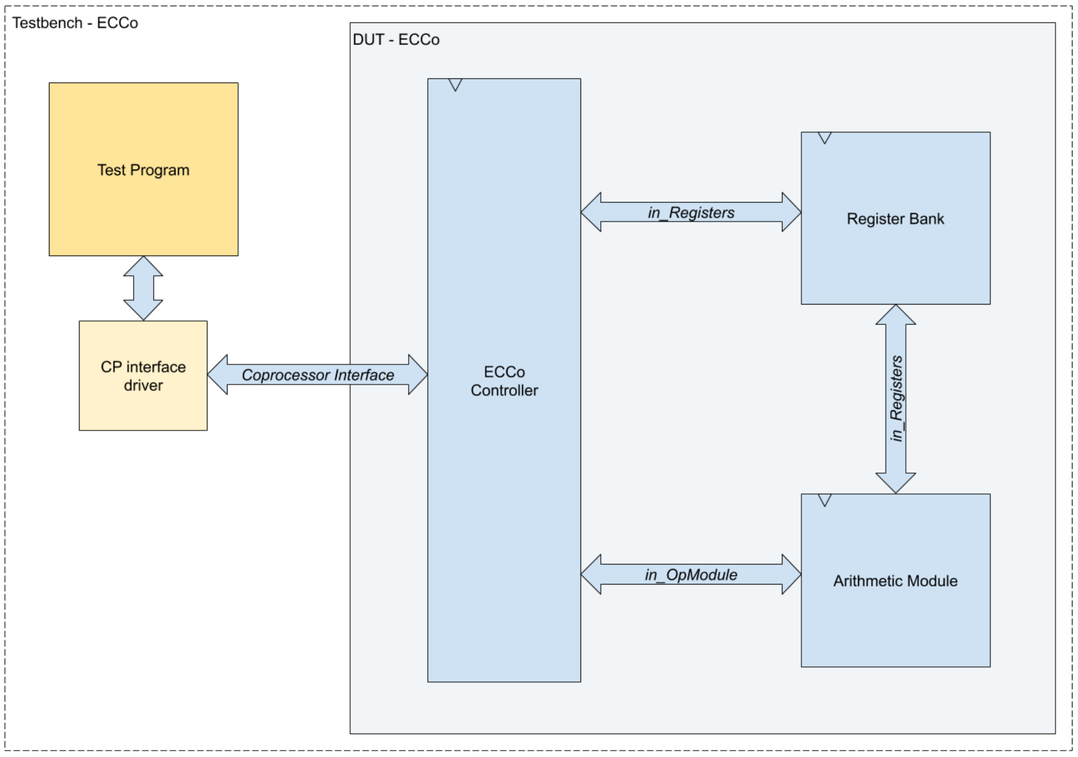

FIGURE 5.13: Testbench setup for verification of ECCo.

were the modular multiplication and modular addition. As these are the fundamental operations of SM the execution time of these will give an indication of the possible speedup. The benchmarking was performed by doing the setup of parameters once, instantiating operand 1 ($OP1$), operand 2 ($OP2$), and modulo (MOD) to large 256-bit values. The same values were used for the libecc and ECCo benchmarks. Then the operation $OP1 = OP1 + OP2 \ \% \ MOD$ were performed for the modular addition benchmark, and $OP1 = OP1 * OP2 \ \% \ MOD$ for the modular multiplication benchmark.

The benchmarks were performed doing runs of 10 and 100 iterations, i.e. performing the operation 10 or 100 times, updating the $OP1$ value each time. The test values were large 256-bit values, making them similar to values used during 256-bit SM. These benchmarks does, however, not include tests of edge cases, such as when $MOD << OP1 + OP2$ in which case the ECCo will have a very long execution time, nor does it guarantee coverage of the case when $MOD > OP1 + OP2$ or $MOD > OP1 * OP2$.

The source code for the benchmarking programs are listed in Appendix F.

Chapter 6

Results

The simulation tests described in Chapter 5, verifying correct behavior of all sub-modules and correct results from implemented arithmetic operations, all succeeded.

This chapter presents the results from the benchmark, comparing the execution time between the modular arithmetic software implementation by libecc and the ECCo implementation. Lastly, the area estimates from synthesis are presented.

6.1 Speed

The execution time of modular addition and modular multiplication is compared between benchmark code running the operations on ECCo and using the software implementation from libecc. Table 6.2 summarizes the benchmarking results. The execution time is measured in clock cycles. As a reference, a simulation run without any operation was performed in order to measure the setup time of the system. This empty run had an execution time of 36,790 cycles (this is included in the results presented in Table 6.2).

The results show that the ECCo performed 3.8 times faster for modular addition at 10 iterations, and 8 times faster at 100 iterations. As for the modular multiplication the ECCo performed 7.8 times faster at 10 and 27 times faster at 100 iterations.

While the ECCo is significantly faster than the compared software implementation another notable result is how the ECCo and software implementation scales differently: From 10 to 100 iterations the ECCo had an increase

Operation	**Exec. Time - 10 Iterations**	**Exec. Time - 100 Iterations**
Modular Addition - ECCo	42,818	43,294
Modular Addition - libecc	164,906	347,966
Modular Multiplication - ECCo	46,840	87,864
Modular Multiplication - libecc	367,664	2,375,744

TABLE 6.2: Execution time of atomic operations. Measured in clock cycles.

Measurement	Increase
Combinational Area	3.12x
Noncombinational Area	1.36x
Total Area	1.83x

TABLE 6.4: Area increase for design when adding ECCo.

Module	Sub-Module	ECCo Acc. Area	Comb. Area	Noncomb. Area
Arithmetic		84.63%	5.80%	13.61%
	Multiplication*	1.97%	1.56%	4.81%
	Addition*	1.92%	1.53%	4.61%
	Negation	0.78%	0.25%	4.49%
	Division	73.18%	83.02%	4.50%
Controller		4.65%	5.25%	0.42%
Register Bank		10.72%	2.59%	67.31%

TABLE 6.6: Area distribution of ECCo modules. *(*modular)*

in execution time of 1.01x (addition) and 1.8x (multiplication), while the software implementation had an increase of 2.1x (addition) and 6.5x (multiplication). This gives an indication on the benefit of having a coprocessor which allows an extensive amount of operations to be performed without the need for data transfer between processor and coprocessor.

6.2 Area

The design of the CM33 with the ECCo was synthesizable, and did not have any negative slack. It was synthesized without any optimization, at a frequency of 128MHz. The area results are presented as a comparison between synthesis estimates of the design with and without the ECCo included (Table 6.4), and a area distribution between the sub-modules of the ECCo (Table 6.6).

The values shown in Table 6.4 are percentage increase in area when synthesizing the CM33 and CM33+ECCo. Clearly, the ECCo contains a great deal of combinatorial logic, increasing area of combinatorial cell area by 312%. In total the ECCo's area equals 83% of existing design.

The values shown in Table 6.6 are the area distribution of the ECCo submodules.

ECCo Accumulative Area The area percentage of the ECCo occupied by this module, included its sub-modules. The percentages of *Arithmetic*, *Controller*, and *Register Bank* modules add up to 100%, being all the sub-modules of the ECCo. The percentages of *Multiplication*, *Addition*, *Negation*, and *Division* are included in the *Arithmetic* percentage, but they do not sum up to 84.63% since the *Arithmetic* module contains some logic of its own.

Combinatorial Area The area percentage of combinatorial cells for only this module, not including any of its sub-modules. E.g. the *Arithmetic* module uses 5.8% of the total area of combinatorial cells in the ECCo, excluded its sub-modules, and the *Division* module uses 83.02% of the total combinatorial area of the ECCo.

Noncombinatorial Area Same as for combinatorial.

Not surprisingly, a majority of the noncombinational area are occupied by the register bank. However, most of the area of the ECCo are occupied by the divider, which were synthesized using the SV division operator "/" without any optimization from the synthesizer.

The implementation of the most essential modules, *Modular Multpilcation* and *Modular Addition*, only occupied 1.97% and 1.92%, respectively. Combined with the benchmark results, this gives an indication of the advantages of using the ECCo: Significant speedup, with only a small area increase, assuming the divisor can be more efficiently implemented. Assuming a more efficient divisor implementation: the register bank may be the module occupying the largest area, currently being $5x$ the size of the *Modular Multiplication* and *Modular Addition* modules, and $2x$ the size of the controller.

Chapter 7

Future Work

The ECCo implementation in this thesis has only included a small subset of necessary operations and features for the suggested design of a complete elliptic curve coprocessor. This chapter discusses possible changes and considerations for future work on the coprocessor proposed in this thesis.

7.1 Instruction Set Architecture

The instruction set proposed in Table 5.2 is intended for a design aimed for solution 2 in Chapter 4.1. The desired solution, however, is solution 3, which requires some additional, higher level operations to be included in the instruction set. More specifically point arithmetic (Chapter 2.2.3) and/or scalar multiplication (Chapter 2.3).

Also, another desirable functionality would be to have a way of generating random numbers of the coprocessors word size. This is because random numbers used in many cryptography algorithms, like ECDSA (Chapter 2.5).

The currently implemented arithmetic operations of modular addition and modular multiplication are also the fundamental operations of common, non-EC crypto systems, like RSA [20] and Diffie-Hellman [4]. Adding instructions for these common algorithms could be usefull, but would require the possibility of working with numbers of bit sizes up to 4096-bit to provide acceptable security.

7.2 Security

An issue which has not been addressed in this thesis, but which must be considered for future work, is security of the implementation against attacks such as side-channel attacks. A way of trying to defend against side-channel attacks is by using constant time algorithms for calculations, which should be considered both for the finite-field arithmetic, point operations and the scalar multiplication algorithm.

7.3 Algorithms

While the implemented algorithms for modular addition and modular multiplication are simple, with more complex and efficient methods available

(Chapters 3.1 and 3.2), the current implementation already provides significant speedup over pure software implementation. A future change in choice of algorithms is necessary for further development, a decision in which a compromise between security and efficiency surely is needed.

The integer division will, however, need a more area efficient implementation. Reducing the area consumption of the divisor module could, potentially, significantly reduce the total area of the ECCo.

Chapter 8

Conclusion

This thesis has explored how to design a coprocessor for accelerating elliptic curve cryptography, based on the results from the prestudy [11]. The coprocessor designed in the thesis, ECCo, was designed for use with the ARM CM33 processor. The CM33 provides a coprocessor interface for tight integration of coprocessors, which allows the instructions to be issued to connected coprocessors from software.

This lead to the ECCo being designed with an instruction set providing the atomic mathematical operations for ECC, with the possibility of adding implementations of scalar multiplication to the instruct set in a future work.

As time did not allow for the entire proposed instruction set to be implemented only the atomic arithmetic operations were implemented, and an ECCo design with a controller, register bank and arithmetic module were used to compare execution time with an ECC software implementation, and to estimate area usage by synthesis. The ECCo accounted for 45% of the area when synthesizing ECCo+CM33. The estimates showed that the ECCo area consumption was largely dominated by the divisor (73.18% of the total ECCo area), which was implemented using the SystemVerilog division operator, "/", and no optimization in synthesis. However, the atomic operations of ECC, Modular Multiplication and Modular Addition, only occupied 1.97% and 1.92%, respectively. These modules also performed 3.8x - 27x faster than a pure software implementation of ECC.

While the implemented algorithms for modular addition and modular multiplication are simple, with more complex and efficient methods available (Chapters 3.1 and 3.2), the current implementation already provides significant speedup over pure software implementation. Providing a complete system which allows efficiency to be achieved through several methods: reducing data transfers, optimizing implementation of mathematical operations and flexibility and ease-of-use.

Appendix A

Test Data Python script

```python
import argparse
import csv
import io
import os
import re
import shutil
import sys
from abc import ABC, abstractclassmethod
from typing import *


# Exception class used to differentiote between known and unknown errors.
class DataError(Exception):
    pass


###############################################################################
#                                                                             #
#                                  Baseclass                                  #
#                                                                             #
###############################################################################

class DataABC(ABC):
    """DataABC is the baseclass for all calculations. It handles reading from
       and writing to csv data files, writing to C files, and number formatting
       (decimal, hex & binary).
    """
    headers = []
    data    = []

    def __init__(self, headers, file: io.IOBase, numBase: int) -> None:
        self.headers = headers
        rd = csv.reader(file)
        # First line of the file must be the headers
        fileHeaders = rd.__next__()
        if self.headers != fileHeaders:
            raise DataError(f'[!!]␣DataABC,␣__init__:␣Invalid␣headers!␣Want␣{self.headers}␣-␣got␣
↪ {fileHeaders}')

        # Read all data
        for j, cols in enumerate(rd):
            # Report and skip empty lines
            if not cols:
                print(f'[␣]␣DataABC,␣__init__:␣Reading␣{file}:␣Found␣empty␣line␣({j+2}).␣
↪ Ignoring...')
                continue
            # Represent the data as a dict, indexed by header names
            tmp = dict()
            for i, h in enumerate(self.headers):
                # Sanitychecks to avoid decimal interpreted as hex etc.
                if not re.match(r'^-?\d+$', cols[i]) and numBase == 10:
                    raise DataError(f'DataABC,␣__init__:␣Reading␣{file}:␣Tried␣interpreting␣
↪ non-decimal␣number␣as␣decimal:␣"{cols[i]}"')
                elif not re.match(r'^-?0x[0-9a-fA-F]+$', cols[i]) and numBase == 16:
                    raise DataError(f'DataABC,␣__init__:␣Reading␣{file}:␣Tried␣interpreting␣non-hex␣
↪ number␣as␣hexadecimal:␣"{cols[i]}"')
                elif not re.match(r'^-?0b[01]+$', cols[i]) and numBase == 2:
                    raise DataError(f'DataABC,␣__init__:␣Reading␣{file}:␣Tried␣interpreting␣
↪ non-binary␣number␣as␣binary:␣"{cols[i]}"')
                tmp[h] = int(cols[i], numBase)
            self.data.append(tmp)

    @abstractclassmethod
    def calculate(self) -> None:
        pass

    @staticmethod
    def _formatNumber(num: int, numFormat: int) -> str:
        # Determine number format string
        if numFormat == 16:
            return f'0x{num:x}' if num >= 0 else f'-0x{abs(num):x}'
        elif numFormat == 2:
            return f'0b{num:b}' if num >= 0 else f'-0b{abs(num):b}'
        else:
            return f'{num}'


    def _formatDataCsv(self, numFormat: int) -> Generator[Dict[str, str], None, None]:
        # Iterate through data values, yield dictionaries with strings of formatted numbers
        for d in self.data:
            tmp = dict()
            for k, v in d.items():
                tmp[k] = self._formatNumber(v, numFormat)
            yield tmp

    def writeCsv(self, file: io.IOBase, numFormat: int) -> None:
        wr = csv.DictWriter(file, fieldnames=self.headers)
        # First writeCsv the header line
        wr.writeheader()
        # Write all data to the file
        for d in self._formatDataCsv(numFormat):
            wr.writerow(d)

    def _formatDataC(self, numFormat: int) -> Generator[List[str], None, None]:
        for d in self.data:
            tmp = list()
            for v in d.values():
                tmp.append(self._formatNumber(v, numFormat))
            yield tmp

    def writeC(self, file: io.IOBase, numFormat: int, fileName: str, arrayName: str) -> None:
        # Need to know size of all the arrays dimensions
        numEntries = len(self.data) + 1  # Zero terminated
        numHeaders = len(self.headers)
        numChars = 0
        # Iterate through all values and find the longest string
        for d in self.data:
            for v in d.values():
                l = len(self._formatNumber(v, numFormat))
                if l > numChars:
                    numChars = l
        numChars += 1  # One extra, for terminating zero

        # Print some general information comments
        print(f'// Created by {sys.argv[0]} with data from {fileName}\n// Number base: {numFormat}',
  ↪ file=file, end='\n\n')
        # Print some macros with meta data
        print(f'#define {arrayName.upper()}_NUM_ENTRIES {numEntries-1}', file=file)
        print(f'#define {arrayName.upper()}_NUM_HEADERS {numHeaders}', file=file)
        print(f'#define {arrayName.upper()}_NUM_CHARS {numChars-1}', file=file, end='\n\n')
        # Print a comment with the headers
        print(f'// [{", ".join(self.headers)}]', file=file)
        # Write the actual data
        print(f'char {arrayName}[{numEntries}][{numHeaders}][{numChars}] = {{', file=file)
        for data in self._formatDataC(numFormat):
            print(f"""        {{"{'", "'.join(data)}"}},""", file=file)
        # End with zero termination
        print('        {0}\n};', file=file)



###############################################################################
#                                                                             #
#                                  Addition                                   #
#                                                                             #
###############################################################################

class ModAddData(DataABC):
    def __init__(self, file: io.IOBase, numBase: int):
        super().__init__(['modulo', 'operand1', 'operand2', 'result'], file, numBase)

    def calculate(self):
        # For each entry calculate op1+op2 % mod
        for i, d in enumerate(self.data):
            self.data[i]['result'] = (d['operand1'] + d['operand2']) % d['modulo']


###############################################################################
#                                                                             #
#                               Multiplication                                #
#                                                                             #
###############################################################################

class ModMulData(DataABC):
    def __init__(self, file: io.IOBase, numBase: int):
        super().__init__(['modulo', 'operand1', 'operand2', 'result'], file, numBase)

    def calculate(self):
        # For each entry calculate op1*op2 % mod
        for i, d in enumerate(self.data):
            self.data[i]['result'] = (d['operand1'] * d['operand2']) % d['modulo']


###############################################################################
#                                                                             #
#                                  Division                                   #
#                                                                             #
###############################################################################

class DivData(DataABC):
    def __init__(self, file: io.IOBase, numBase: int):
        super().__init__(['operand1', 'operand2', 'result'], file, numBase)

    def calculate(self):
        # For each entry calculate op1/op2, integer division
        for i, d in enumerate(self.data):
            op1 = d['operand1']
            op2 = d['operand2']
            # Integer division doesn't behave as expected when dealing with
            # negative numbers (e.g. it thinks 3//-4 = -1) so just give it
            # positive numbers instead and use basic arithmetic rules for
            # determining result sign.
            if op1 < 0 and op2 < 0:
                self.data[i]['result'] = abs(op1) // abs(op2)
            elif op1 < 0:
                self.data[i]['result'] = -(abs(op1) // op2)
            elif op2 < 0:
                self.data[i]['result'] = -(op1 // abs(op2))
            else:
                self.data[i]['result'] = op1 // op2


###############################################################################
#                                                                             #
#                                  Main code                                  #
#                                                                             #
###############################################################################

if __name__ == "__main__":
    # Setup argparse
    par = argparse.ArgumentParser()
    par.add_argument('FILE', type=str, help='data file on either hexa, binary or decimal format.')
    par.add_argument('-o', metavar="FILE", type=str, help='optional output file')
    par.add_argument('-c', action='store_true', help='output the data as C-array instead of CSV')
    par.add_argument('-b', action='store_true', help='create a backup file')
    # Use a mutually exclusive group for selecting number format
    formatGroup = par.add_mutually_exclusive_group(required=True)
    formatGroup.add_argument('--dec', action='store_true', help='input data is on decimal format.')
    formatGroup.add_argument('--hex', action='store_true', help='input data is on hexadecimal format.')
    formatGroup.add_argument('--bin', action='store_true', help='input data is on binary format.')
    # Use a mutually exclusive group for selecting output number format
    formatGroup = par.add_mutually_exclusive_group(required=False)
    formatGroup.add_argument('--outDec', action='store_true', help='output data is on decimal format.')
    formatGroup.add_argument('--outHex', action='store_true', help='output data is on hexadecimal format.')
    formatGroup.add_argument('--outBin', action='store_true', help='output data is on binary format.')
    # Use a mutually exclusive group for selecting operation
    operationGroup = par.add_mutually_exclusive_group(required=True)
    operationGroup.add_argument('--add', action='store_true', help='calculate data for modular addition.')
    operationGroup.add_argument('--mul', action='store_true', help='calculate data for modular multiplication.')
    operationGroup.add_argument('--div', action='store_true', help='calculate data for integer division.')

    args     = vars(par.parse_args())
    dataFile = args['FILE']
    bkupFile = f'{dataFile}.backup'
    outFile  = args['o'] if args['o'] else dataFile
    csvOut   = not args['c']

    # Select data operation
    if args['add']:
        dataClass = ModAddData
        cArrayName = 'dataAdd'
    elif args['mul']:
        dataClass = ModMulData
        cArrayName = 'dataMul'
    elif args['div']:
        dataClass = DivData
        cArrayName = 'dataDiv'

    # Select input number base
    if args['dec']:
        inBase = 10
    elif args['hex']:
        inBase = 16
    elif args['bin']:
        inBase = 2
    # Select output number base
    if args['outDec']:
        outBase = 10
    elif args['outHex']:
        outBase = 16
    elif args['outBin']:
        outBase = 2
    else:
        outBase = inBase
    cArrayName = f'{cArrayName}{outBase}'

    # Perform calculation
    try:
        with open(dataFile, 'r', newline='') as fin:
            data = dataClass(fin, inBase)
        data.calculate()
        if args['b']:
            shutil.copy(dataFile, bkupFile)
        with open(outFile, 'w', newline='') as fout:
            if csvOut:
                data.writeCsv(fout, outBase)
            else:
                data.writeC(fout, outBase, outFile, cArrayName)
    except DataError as e:
        print(e, file=sys.stderr)
```

LISTING A.1: Python script for generating test data

Appendix B

Internal Interfaces SV Code

```
interface in_Registers;
  logic [NUM_REGS-1:0][WORD_WIDTH:0] registers;
  logic [WORD_WIDTH:0]               wData;
  logic [3:0]                        wReg;
  logic                              wEnable;

  modport slave (
    output registers,
    input  wData,
    input  wReg,
    input  wEnable
  );
  modport master (
    input  registers,
    output wData,
    output wReg,
    output wEnable
  );
endinterface

interface in_OpModule;
  logic       ready;
  logic       error;
  logic       valid;
  logic [3:0] opcode;
  logic [3:0] op1Reg;
  logic [3:0] op2Reg;
  logic [3:0] resReg;

  modport slave (
    output ready,
    output error,
    input  valid,
    input  opcode,
    input  op1Reg,
    input  op2Reg,
    input  resReg
  );
  modport master (
    input  ready,
    input  error,
    output valid,
    output opcode,
    output op1Reg,
    output op2Reg,
    output resReg
  );
endinterface
```

LISTING B.1: SystemVerilog code for the internal interfaces of ECCo.

Appendix C

Test Data

```
modulo, operand1, operand2, result
7,15,1,2
11,3,2,5
11,3,−4,10
233,75,77,152
233,567,895,64
233,567,−895,138
28657,16578,19504,7425
514229,546500,357980,390251
99194853094755497,98275954794755497,12457956214,98275967252711711
99194853094755497,98275954794755497,−12457956214,98275942336799283
92567853094755497,98275954794755497,92657924597654697,5798173202899200

92567853094755497,−98275954794755497,−92657924597654697,86769679891856297

75356465794755497,65245765798756497,70253759756423697,60143059760424697

74225698149877013133163669918490695756676765155849109751738796007550114900164,5522897

55228977394393414412853003502097247104908965897402951232160234933662925082798,4522897

74225698149877013133163669918490695756676765155849109751738796007550114900164,5522897

74225698149877013133163669918490695756676765155849109751738796007550114900164,6522897

74225698149877013133163669918490695756676765155849109751738796007550114900164,3528977

74225698149877013133163669918490695756676765155849109751738796007550114900164,9528977

74225698149877013133163669918490695756676765155849109751738796007550114900164,8528977

74225698149877013133163669918490695756676765155849109751738796007550114900164,4522897
```

LISTING C.1: Modular addition test data.

```
modulo, operand1, operand2, result
7,15,1,1
11,3,2,6
233,75,77,183
233,567,895,224
```

```
28657,16578,19504,381
514229,546500,357980,218095
99194853094755497,98275954794755497,12457956214,31017271154744113
92567853094755497,98275954794755497,92657924597654697,48036520782282743

75356465794755497,65245765798756497,70253759756423697,65782237743603078

74225698149877013133163669918490695756676765155849109751738796007550114900164,5522

55228977394393414412853003502097247104908965897402951232160234933662925082798,4522

74225698149877013133163669918490695756676765155849109751738796007550114900164,5522

74225698149877013133163669918490695756676765155849109751738796007550114900164,6522

74225698149877013133163669918490695756676765155849109751738796007550114900164,3528

74225698149877013133163669918490695756676765155849109751738796007550114900164,9528

74225698149877013133163669918490695756676765155849109751738796007550114900164,8528

74225698149877013133163669918490695756676765155849109751738796007550114900164,4522
```

LISTING C.2: Modular multiplication test data.

```
operand1,operand2,result
5,1,5
3,2,1
3,-4,0
75,77,0
567,895,0
567,-895,0
16578,19504,0
546500,357980,1
98275954794755497,12457956214,7888609
98275954794755497,-12457956214,-7888609
98275954794755497,92657924597654697,1
98275954794755497,97,1013154173141809
98275954794755497,-97,-1013154173141809
65245765798756497,70256423697,928680
55228977394654679572853003502097247104908965897402951232160234933662925082798,4128

65228977394654679572853003502097247104908965897402951232160234933662925082798,4128

35289773946546795728530035020972471049089658974029512321602349336629250827981,41285

95289773946546795728530035020972471049089658974029512321602349336629250827981,91285

85289773946546795728530035020972471049089658974029512321602349336629250827981,91285

45228977394393414412853003502097247104908965897402951232160234933662925082798,1329
```

LISTING C.3: Integer division test data.

Appendix D

ECCo C Wrapper

```
#ifndef ECC_H
#define ECC_H

/***************************************************************
 *                                                             *
 *                    Internal ecc.h macros                    *
 *                                                             *
 ***************************************************************/

// Coprocessor number of the ECCo
#define __ECC_COPROC "p0"

/************
 *  Opcodes  *
 ************/

// Arithmetic
#define __ECC_OPC1_MUL "0x0"
#define __ECC_OPC1_ADD "0x1"
#define __ECC_OPC1_DIV "0x2"
#define __ECC_OPC1_NEG "0x3"
// Logical
#define __ECC_OPC1_LOG "0xd"
#define __ECC_OPC2_OR  "0x0"
#define __ECC_OPC2_AND "0x1"
#define __ECC_OPC2_XOR "0x2"
#define __ECC_OPC2_NOT "0x3"
// Shift
#define __ECC_OPC1_SFT "0xe"
#define __ECC_OPC2_LSL "0x0"
#define __ECC_OPC2_LSR "0x1"
#define __ECC_OPC2_ASR "0x2"
// Comparison
#define __ECC_OPC1_CMP "0xf"
#define __ECC_OPC2_ZR  "0x0"
#define __ECC_OPC2_NZR "0x1"
#define __ECC_OPC2_EQ  "0x2"
#define __ECC_OPC2_NEQ "0x3"
#define __ECC_OPC2_LT  "0x4"
#define __ECC_OPC2_GT  "0x5"
// Miscellaneous
#define __ECC_OPC1_INC "0xa"
#define __ECC_OPC1_DEC "0xb"
#define __ECC_OPC1_SSB "0xc"
#define __ECC_OPC2_SSB "0x0"
#define __ECC_OPC1_USB "0xc"
#define __ECC_OPC2_USB "0x1"


/***************************************************************
 *                                                             *
 *                    Exported ecc.h macros                    *
 *                                                             *
 ***************************************************************/
```

```

#ifndef NULL
 #define NULL ((void*)0)
#endif

/********************************
 *  Coprocessor interface meta  *
 ********************************/

#define ECC_OP1_WIDTH           4
#define ECC_OP1_MAX             15
#define ECC_OP2_WIDTH           3
#define ECC_OP2_MAX             7
#define ECC_REG_IDX_WIDTH       4
#define ECC_REG_IDX_MAX         15
#define ECC_WORD_WIDTH          256
#define ECC_WORD_WIDTH_BYTE     (ECC_WORD_WIDTH/8)
#define ECC_MODULO_REG          "14"
#define ECC_STATUS_REG          "15"

/***************************
 *  Arithmetic operations  *
 ***************************/

// All arguments are coprocessor register indexes, which must be integers
    in double quotes.
#define ECC_MUL(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_MUL",␣cr"op2Reg",␣cr"op1Reg",␣cr"resReg",␣#0")
#define ECC_ADD(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_ADD",␣cr"op2Reg",␣cr"op1Reg",␣cr"resReg",␣#0")
#define ECC_DIV(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_DIV",␣cr"op2Reg",␣cr"op1Reg",␣cr"resReg",␣#0")
#define ECC_NEG(opReg,  resReg)         asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_NEG",␣cr0,␣␣␣␣␣␣␣␣cr"opReg",␣cr"resReg",␣#0")


/************************
 *  Logical operations  *
 ************************/

// All arguments are coprocessor register indexes, which must be integers
    in double quotes.
#define ECC_OR( op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_LOG",␣cr" op2Reg",␣cr"op1Reg",␣cr"resReg",␣#"__ECC_OPC2_OR
    )
#define ECC_AND(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_LOG",␣cr" op2Reg",␣cr"op1Reg",␣cr"resReg",␣#"
    __ECC_OPC2_AND)
#define ECC_XOR(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_LOG",␣cr" op2Reg",␣cr"op1Reg",␣cr"resReg",␣#"
    __ECC_OPC2_XOR)
#define ECC_NOT(opReg,  resReg)         asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_LOG",␣cr0,␣␣␣␣␣␣␣␣␣cr"op1Reg",␣cr"resReg",␣#"
    __ECC_OPC2_NOT)


/**********************
 *  Shift operations  *
 **********************/

// All arguments are coprocessor register indexes, which must be integers
    in double quotes.
#define ECC_LSL(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_SFT",␣cr"op2Reg",␣cr"op1Reg",␣cr"resReg",␣#"__ECC_OPC2_LSL
    )
#define ECC_LSR(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_SFT",␣cr"op2Reg",␣cr"op1Reg",␣cr"resReg",␣#"__ECC_OPC2_LSR
    )
#define ECC_ASR(op1Reg, op2Reg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣
    #"__ECC_OPC1_SFT",␣cr"op2Reg",␣cr"op1Reg",␣cr"resReg",␣#"__ECC_OPC2_ASR
    )

```

```

/*************************
 *  Comparison operations  *
 *************************/

// All arguments are coprocessor register indexes, which must be integers in double quotes.
#define ECC_ZR( reg)            asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_CMP ",␣cr0,␣␣␣␣␣␣␣␣cr"reg",␣␣␣␣cr0,␣#"__ECC_OPC2_ZR)
#define ECC_NZR(reg)            asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_CMP ",␣cr0,␣␣␣␣␣␣␣␣cr"reg",␣␣␣␣cr0,␣#"__ECC_OPC2_NZR)
#define ECC_EQ( op1Reg, op2Reg) asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_CMP ",␣cr"op2Reg",␣cr"op1Reg",␣cr0,␣#"__ECC_OPC2_EQ)
#define ECC_NEQ(op1Reg, op2Reg) asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_CMP ",␣cr"op2Reg",␣cr"op1Reg",␣cr0,␣#"__ECC_OPC2_NEQ)
#define ECC_LT( op1Reg, op2Reg) asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_CMP ",␣cr"op2Reg",␣cr"op1Reg",␣cr0,␣#"__ECC_OPC2_LT)
#define ECC_GT( op1Reg, op2Reg) asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_CMP ",␣cr"op2Reg",␣cr"op1Reg",␣cr0,␣#"__ECC_OPC2_GT)


/*****************************
 *  Miscellaneous operations  *
 *****************************/

// All arguments are coprocessor register indexes, which must be integers in double quotes.
#define ECC_INC(opReg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_INC ",␣cr0,␣cr"opReg",␣cr"resReg",␣#0")
#define ECC_DEC(opReg, resReg) asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_DEC ",␣cr0,␣cr"opReg",␣cr"resReg",␣#0")
#define ECC_SSB(reg)           asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_SSB ",␣cr0,␣cr"reg",␣␣␣cr0,␣␣␣␣␣␣␣␣#"__ECC_OPC2_SSB)
#define ECC_USB(reg)           asm volatile ("cdp␣"__ECC_COPROC",␣#"__ECC_OPC1_USB ",␣cr0,␣cr"reg",␣␣␣cr0,␣␣␣␣␣␣␣␣#"__ECC_OPC2_USB)


/************************
 *  Data transfer macros  *
 ************************/

/* Load coprocessor register macros. Offset is in hexa. 'reg' is a coprocessor
   register index and must be a decimal integer in double quotes. 'Rt' and 'Rt2' are
   32-bit input variables. */
#define ECC_LOAD_0(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x0,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#if ECC_WORD_WIDTH > 64
 #define ECC_LOAD_1(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x1,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_1(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 128
 #define ECC_LOAD_2(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x2,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_2(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 192
 #define ECC_LOAD_3(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x3,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_3(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 256
 #define ECC_LOAD_4(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x4,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_4(Rt, Rt2, reg)
#endif
```

```
#if ECC_WORD_WIDTH > 320
 #define ECC_LOAD_5(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x5,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_5(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 384
 #define ECC_LOAD_6(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x6,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_6(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 448
 #define ECC_LOAD_7(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x7,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_7(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 512
 #define ECC_LOAD_8(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x8,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_8(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 576
 #define ECC_LOAD_9(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0x9,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_9(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 640
 #define ECC_LOAD_10(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0xa,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_10(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 704
 #define ECC_LOAD_11(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0xb,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_11(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 768
 #define ECC_LOAD_12(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0xc,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_12(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 832
 #define ECC_LOAD_13(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0xd,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_13(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 896
 #define ECC_LOAD_14(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0xe,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_14(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 960
 #define ECC_LOAD_15(Rt, Rt2, reg) asm volatile ("mcrr␣"__ECC_COPROC",␣#0xf,␣%0,␣%1,␣cr"reg :: "rm" (Rt), "rm" (Rt2))
#else
 #define ECC_LOAD_15(Rt, Rt2, reg)
#endif

/* Store coprocessor register macros. Offset is in hexa. 'reg' is a coprocessor
   register index and must be a decimal integer in double quotes. 'Rt' and 'Rt2' are
   32-bit output variables. */
```

```
#define ECC_STORE_0(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x0,
    ␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#if ECC_WORD_WIDTH > 64
 #define ECC_STORE_1(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x1
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_1(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 128
 #define ECC_STORE_2(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x2
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_2(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 192
 #define ECC_STORE_3(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x3
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_3(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 256
 #define ECC_STORE_4(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x4
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_4(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 320
 #define ECC_STORE_5(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x5
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_5(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 384
 #define ECC_STORE_6(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x6
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_6(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 448
 #define ECC_STORE_7(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x7
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_7(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 512
 #define ECC_STORE_8(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x8
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_8(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 576
 #define ECC_STORE_9(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0x9
    ,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_9(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 640
 #define ECC_STORE_10(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0
    xa,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_10(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 704
 #define ECC_STORE_11(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0
    xb,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_11(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 768
 #define ECC_STORE_12(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0
    xc,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
```

```
 #define ECC_STORE_12(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 832
 #define ECC_STORE_13(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0
     xd,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_13(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 896
 #define ECC_STORE_14(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0
     xe,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_14(Rt, Rt2, reg)
#endif
#if ECC_WORD_WIDTH > 960
 #define ECC_STORE_15(Rt, Rt2, reg) asm volatile ("mrrc␣"__ECC_COPROC",␣#0
     xf,␣%0,␣%1,␣cr"reg : "=rm" (Rt), "=rm" (Rt2))
#else
 #define ECC_STORE_15(Rt, Rt2, reg)
#endif

#endif  // ECC_H
```

LISTING D.1: ECCo C wrapper source.

Appendix E

ECCo Big Number library

```c
#ifndef ECC_WORD_H
#define ECC_WORD_H

#include <stdbool.h>

#include "ecc.h"

/* Length of array in word struct. Define here instead of ecc.h since it
    depends
   on array type. */
#define EW_LENGTH (ECC_WORD_WIDTH_BYTE/sizeof(int))

/* +4 to fit terminating '\0', leading '0b' and optional '-' sign. */
#define EW_STR_LENGTH ECC_WORD_WIDTH+4

/* ecc_word is the datatype to work with big numbers width the same width
    as
   the ECC coprocessors word size. */
typedef struct {
    int  word [EW_LENGTH];
    bool is_zero;
    bool is_negative;
} ecc_word_t;

/* String-type big enough to represent any number on either
   binary, decimal or hexadecimal format. */
typedef char ew_str_t[EW_STR_LENGTH];

/* Initializes a ecc_word. Returns a pointer to the given word. */
ecc_word_t* ew_init(ecc_word_t*);

/* Creates a new copy of an ecc_word. Returns a pointer to dst. */
ecc_word_t* ew_copy(const ecc_word_t* restrict src, ecc_word_t* restrict
    dst);


/*************************************************************
 *                                                           *
 *                   Content handlers                        *
 *                                                           *
 *************************************************************/

/* Sets the content of a ecc_word to 0. Returns a pointer to the given word
    . */
ecc_word_t* ew_zero(ecc_word_t*);

/* Set the value to an integer value. */
ecc_word_t* ew_set_int(ecc_word_t*, int);

/* Set the value of a word to a number represented by a string in
    hexadecimal
   (0x prefix) format. Return a pointer to the word, or NULL on failure. */
ecc_word_t* ew_set_str(ecc_word_t*, const char[]);

```

```
/* Set parts of the content of a word, based on the given offset. */
ecc_word_t* ew_set_offs(ecc_word_t* w, int offs, int r1, int r2);

/* Return a pointer to the hexadecimal formatted string of the number. */
char* ew_to_str(const ecc_word_t*, char[], int);


/**************************************************************
 *                                                            *
 *                         Comparison                         *
 *                                                            *
 **************************************************************/

/* Check if two words are equal. */
bool ew_eq(const ecc_word_t*, const ecc_word_t*);


/**************************************************************
 *                                                            *
 *                  Coprocessor interraction                  *
 *                                                            *
 **************************************************************/

/* Load the given word into a coprocessor register. */
void ew_load_cr0(const ecc_word_t*);
void ew_load_cr1(const ecc_word_t*);
void ew_load_cr2(const ecc_word_t*);
void ew_load_cr3(const ecc_word_t*);
void ew_load_cr4(const ecc_word_t*);
void ew_load_cr5(const ecc_word_t*);
void ew_load_cr6(const ecc_word_t*);
void ew_load_cr7(const ecc_word_t*);
void ew_load_cr8(const ecc_word_t*);
void ew_load_cr9(const ecc_word_t*);
void ew_load_cr10(const ecc_word_t*);
void ew_load_cr11(const ecc_word_t*);
void ew_load_cr12(const ecc_word_t*);
void ew_load_cr13(const ecc_word_t*);
void ew_load_cr14(const ecc_word_t*);
/* CP register 15 is status register and unwriteable */

/* Store the value of a coprocessors register in the given word. Takes
   coprocessor register index as second argument. */
void ew_store_cr0(ecc_word_t*);
void ew_store_cr1(ecc_word_t*);
void ew_store_cr2(ecc_word_t*);
void ew_store_cr3(ecc_word_t*);
void ew_store_cr4(ecc_word_t*);
void ew_store_cr5(ecc_word_t*);
void ew_store_cr6(ecc_word_t*);
void ew_store_cr7(ecc_word_t*);
void ew_store_cr8(ecc_word_t*);
void ew_store_cr9(ecc_word_t*);
void ew_store_cr10(ecc_word_t*);
void ew_store_cr11(ecc_word_t*);
void ew_store_cr12(ecc_word_t*);
void ew_store_cr13(ecc_word_t*);
void ew_store_cr14(ecc_word_t*);
void ew_store_cr15(ecc_word_t*);

/* Convenience macros */
#define EW_LOAD_MOD(WORD) ew_load_cr14(WORD)
#define EW_STORE_MOD(WORD) ew_store_cr14(WORD)
#define EW_STORE_STATUS(WORD) ew_store_cr15(WORD)


/*************************
 *  Offset select macros  *
 *************************/

#define EW_GET_0(Rt, Rt2, W) Rt = W->word[0]; Rt2 = W->word[1]
#if ECC_WORD_WIDTH > 64
```

```
 #define EW_GET_1(Rt, Rt2, W) Rt = W->word[2]; Rt2 = W->word[3]
#else
 #define EW_GET_1(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 128
 #define EW_GET_2(Rt, Rt2, W) Rt = W->word[4]; Rt2 = W->word[5]
#else
 #define EW_GET_2(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 192
 #define EW_GET_3(Rt, Rt2, W) Rt = W->word[6]; Rt2 = W->word[7]
#else
 #define EW_GET_3(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 256
 #define EW_GET_4(Rt, Rt2, W) Rt = W->word[8]; Rt2 = W->word[9]
#else
 #define EW_GET_4(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 320
 #define EW_GET_5(Rt, Rt2, W) Rt = W->word[10]; Rt2 = W->word[11]
#else
 #define EW_GET_5(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 384
 #define EW_GET_6(Rt, Rt2, W) Rt = W->word[12]; Rt2 = W->word[13]
#else
 #define EW_GET_6(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 448
 #define EW_GET_7(Rt, Rt2, W) Rt = W->word[14]; Rt2 = W->word[15]
#else
 #define EW_GET_7(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 512
 #define EW_GET_8(Rt, Rt2, W) Rt = W->word[16]; Rt2 = W->word[17]
#else
 #define EW_GET_8(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 576
 #define EW_GET_9(Rt, Rt2, W) Rt = W->word[18]; Rt2 = W->word[19]
#else
 #define EW_GET_9(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 640
 #define EW_GET_10(Rt, Rt2, W) Rt = W->word[20]; Rt2 = W->word[21]
#else
 #define EW_GET_10(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 704
 #define EW_GET_11(Rt, Rt2, W) Rt = W->word[22]; Rt2 = W->word[23]
#else
 #define EW_GET_11(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 768
 #define EW_GET_12(Rt, Rt2, W) Rt = W->word[24]; Rt2 = W->word[25]
#else
 #define EW_GET_12(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 832
 #define EW_GET_13(Rt, Rt2, W) Rt = W->word[26]; Rt2 = W->word[27]
#else
 #define EW_GET_13(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 896
 #define EW_GET_14(Rt, Rt2, W) Rt = W->word[28]; Rt2 = W->word[29]
#else
 #define EW_GET_14(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 960
 #define EW_GET_15(Rt, Rt2, W) Rt = W->word[30]; Rt2 = W->word[31]
#else
```

```
 #define EW_GET_15(Rt, Rt2, W)
#endif

#define EW_SET_0(Rt, Rt2, W) ew_set_offs(W, 0, Rt, Rt2)
#if ECC_WORD_WIDTH > 64
 #define EW_SET_1(Rt, Rt2, W) ew_set_offs(W, 1, Rt, Rt2)
#else
 #define EW_SET_1(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 128
 #define EW_SET_2(Rt, Rt2, W) ew_set_offs(W, 2, Rt, Rt2)
#else
 #define EW_SET_2(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 192
 #define EW_SET_3(Rt, Rt2, W) ew_set_offs(W, 3, Rt, Rt2)
#else
 #define EW_SET_3(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 256
 #define EW_SET_4(Rt, Rt2, W) ew_set_offs(W, 4, Rt, Rt2)
#else
 #define EW_SET_4(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 320
 #define EW_SET_5(Rt, Rt2, W) ew_set_offs(W, 5, Rt, Rt2)
#else
 #define EW_SET_5(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 384
 #define EW_SET_6(Rt, Rt2, W) ew_set_offs(W, 6, Rt, Rt2)
#else
 #define EW_SET_6(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 448
 #define EW_SET_7(Rt, Rt2, W) ew_set_offs(W, 7, Rt, Rt2)
#else
 #define EW_SET_7(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 512
 #define EW_SET_8(Rt, Rt2, W) ew_set_offs(W, 8, Rt, Rt2)
#else
 #define EW_SET_8(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 576
 #define EW_SET_9(Rt, Rt2, W) ew_set_offs(W, 9, Rt, Rt2)
#else
 #define EW_SET_9(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 640
 #define EW_SET_10(Rt, Rt2, W) ew_set_offs(W, 10, Rt, Rt2)
#else
 #define EW_SET_10(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 704
 #define EW_SET_11(Rt, Rt2, W) ew_set_offs(W, 11, Rt, Rt2)
#else
 #define EW_SET_11(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 768
 #define EW_SET_12(Rt, Rt2, W) ew_set_offs(W, 12, Rt, Rt2)
#else
 #define EW_SET_12(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 832
 #define EW_SET_13(Rt, Rt2, W) ew_set_offs(W, 13, Rt, Rt2)
#else
 #define EW_SET_13(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 896
 #define EW_SET_14(Rt, Rt2, W) ew_set_offs(W, 14, Rt, Rt2)
#else
```

```
 #define EW_SET_14(Rt, Rt2, W)
#endif
#if ECC_WORD_WIDTH > 960
 #define EW_SET_15(Rt, Rt2, W) ew_set_offs(W, 15, Rt, Rt2)
#else
 #define EW_SET_15(Rt, Rt2, W)
#endif

#endif // ECC_WORD_H
```

LISTING E.1: Header file for big number implementation of an ECCo word.

```
#include "ecc_word.h"

#include <ee_printf.h>
#include <stdbool.h>

#include "ecc.h"

ecc_word_t*
ew_init(ecc_word_t* w)
{
    for ( int i = 0; i < EW_LENGTH; i++ )
        w->word[i] = 0;
    w->is_zero     = true;
    w->is_negative = false;
    return w;
}

ecc_word_t*
ew_copy(const ecc_word_t* restrict src, ecc_word_t* restrict dst)
{
    if ( !src->is_zero )
        for ( int i = 0; i < EW_LENGTH; i++ )
            dst->word[i] = src->word[i];
    else
        for ( int i = 0; i < EW_LENGTH; i++ )
            dst->word[i] = 0;

    dst->is_zero     = src->is_zero;
    dst->is_negative = src->is_negative;
    return dst;
}


/***************************************************************
 *                                                             *
 *                      Content handlers                       *
 *                                                             *
 ***************************************************************/

ecc_word_t*
ew_zero(ecc_word_t* w)
{
    if ( !w->is_zero ) {
        for ( int i = 0; i < EW_LENGTH; i++ )
            w->word[i] = 0;
        w->is_zero = 1;
    }
    return w;
}

ecc_word_t*
ew_set_int(ecc_word_t* w, int val)
{
    ew_zero(w);
    w->word[0] = val;
    w->is_zero = false;
    return w;
}

ecc_word_t*
ew_set_str(ecc_word_t* w, const char str[])
{
    int    shift, tmp;
    int*   num = w->word;
    const char* c;

    for ( c = str ; *c != '\0'; c++ )
        ;

    /* Check sign */
    if ( *str == '-' ) {
        w->is_negative = true;
        str++;
    }
    else
        w->is_negative = false;

    /* Sanity checks */
    if ( *str++ != '0' ) {
        MSG(("ew_set_str:␣badly␣formatted␣string,␣must␣start␣with␣'0x'␣or␣
    '-0x'\n"));
        return NULL;
    }
    if ( *str != 'x' ) {
        MSG(("ew_set_str:␣badly␣formatted␣string,␣must␣start␣with␣'0x'␣or␣
    '-0x'\n"));
        return NULL;
    }

    /* Set word to zero if non-zero */
    if ( !w->is_zero ) {
        do
            *num = 0;
        while ( ++num != w->word+EW_LENGTH );
        w->is_zero = true;
        num        = w->word;
    }

    do {
        tmp = 0;
        for ( shift = 0; shift < 32 && --c != str; shift += 4 ) {
            switch ( *c ) {
            case 'f': case 'F':
                tmp ^= 0xf << shift;
                break;
            case 'e': case 'E':
                tmp ^= 0xe << shift;
                break;
            case 'd': case 'D':
                tmp ^= 0xd << shift;
                break;
            case 'c': case 'C':
                tmp ^= 0xc << shift;
                break;
            case 'b': case 'B':
                tmp ^= 0xb << shift;
                break;
            case 'a': case 'A':
                tmp ^= 0xa << shift;
                break;
            default:
                if ( *c < '0' && *c > '9' ) {
                    MSG(("ew_set_str:␣invalid␣character␣in␣string:␣%c", *c)
    );
                    return NULL;
                }
                tmp ^= (*c - '0') << shift;
                }
            }
            if ( tmp && w->is_zero )
                w->is_zero = false;
            *num = tmp;
        } while ( c != str && ++num != w->word+EW_LENGTH );

        return w;
}

ecc_word_t*
ew_set_offs(ecc_word_t* w, int offs, int r1, int r2)
{
        if ( w->is_zero )
            if ( r1 || r2 )
                w->is_zero = false;
        offs *= 2;
        w->word[offs]   = r1;
        w->word[offs+1] = r2;
        return w;
}

char*
ew_to_str(const ecc_word_t* w, char s[], int sz)
{
        int            i = 0, shift;
        const int*     num = w->word+EW_LENGTH;
        unsigned char tmp;

        if ( sz < 4 ) {
            MSG(("ew_to_str:␣too␣small␣string:␣sz␣=␣%d\n", sz));
            return NULL;
        }
        if ( w->is_negative )
            s[i++] = '-';
        s[i++] = '0';
        s[i++] = 'x';

        while ( i < sz && num-- != w->word )
            for ( shift = 28; shift >= 0 && i < sz; shift -= 4, i++ )
                switch ( (tmp = (*num >> shift) & 0xf) ) {
                case 0xf:
                    s[i] = 'f';
                    break;
                case 0xe:
                    s[i] = 'e';
                    break;
                case 0xd:
                    s[i] = 'd';
                    break;
                case 0xc:
                    s[i] = 'c';
                    break;
                case 0xb:
                    s[i] = 'b';
                    break;
                case 0xa:
                    s[i] = 'a';
                    break;
                default:
                    s[i] = (tmp > 9) ? 'X' : tmp + '0';
            }

        if ( i < sz )
            s[i] = '\0';
        else {
            MSG(("ew_to_str:␣too␣small␣string:␣sz␣=␣%d\n", sz));
            return NULL;
        }
        return s;
}

```

```

/*****************************************************************
 *                                                               *
 *                          Comparison                           *
 *                                                               *
 *****************************************************************/

bool
ew_eq(const ecc_word_t* lhs, const ecc_word_t* rhs)
{
    const int* lw = lhs->word+EW_LENGTH;
    const int* rw = rhs->word+EW_LENGTH;

    if ( lhs->is_zero && rhs->is_zero )
        return true;
    while ( *--lw == *--rw )
        if ( lw == lhs->word )
            return true;
    return false;
}

/*****************************************************************
 *                                                               *
 *                       Coprocessor load                        *
 *                                                               *
 *****************************************************************/

#define _EW_LOAD_CR(N) void ew_load_cr##N(const ecc_word_t* w) { \
    volatile register int r1, r2; \
    /* Offset 0 */ \
    EW_GET_0(r1, r2, w); \
    ECC_LOAD_0(r1, r2, #N); \
    /* Offset 1 */ \
    EW_GET_1(r1, r2, w); \
    ECC_LOAD_1(r1, r2, #N); \
    /* Offset 2 */ \
    EW_GET_2(r1, r2, w); \
    ECC_LOAD_2(r1, r2, #N); \
    /* Offset 3 */ \
    EW_GET_3(r1, r2, w); \
    ECC_LOAD_3(r1, r2, #N); \
    /* Offset 4 */ \
    EW_GET_4(r1, r2, w); \
    ECC_LOAD_4(r1, r2, #N); \
    /* Offset 5 */ \
    EW_GET_5(r1, r2, w); \
    ECC_LOAD_5(r1, r2, #N); \
    /* Offset 6 */ \
    EW_GET_6(r1, r2, w); \
    ECC_LOAD_6(r1, r2, #N); \
    /* Offset 7 */ \
    EW_GET_7(r1, r2, w); \
    ECC_LOAD_7(r1, r2, #N); \
    /* Offset 8 */ \
    EW_GET_8(r1, r2, w); \
    ECC_LOAD_8(r1, r2, #N); \
    /* Offset 9 */ \
    EW_GET_9(r1, r2, w); \
    ECC_LOAD_9(r1, r2, #N); \
    /* Offset a */ \
    EW_GET_10(r1, r2, w); \
    ECC_LOAD_10(r1, r2, #N); \
    /* Offset b */ \
    EW_GET_11(r1, r2, w); \
    ECC_LOAD_11(r1, r2, #N); \
    /* Offset c */ \
    EW_GET_12(r1, r2, w); \
    ECC_LOAD_12(r1, r2, #N); \
    /* Offset d */ \
    EW_GET_13(r1, r2, w); \
    ECC_LOAD_13(r1, r2, #N); \
    /* Offset e */ \
    EW_GET_14(r1 , r2 , w) ; \
    ECC_LOAD_14(r1 , r2 , #N) ; \
    /* Offset f */ \
    EW_GET_15(r1 , r2 , w) ; \
    ECC_LOAD_15(r1 , r2 , #N) ; \
 \
    if ( w->is_negative ) /* Set signed bit if negative */ \
        ECC_NEG(#N, #N) ; \
    else /* Else make sure it's unset */ \
        ECC_USB(#N) ; \
}

_EW_LOAD_CR(0)
_EW_LOAD_CR(1)
_EW_LOAD_CR(2)
_EW_LOAD_CR(3)
_EW_LOAD_CR(4)
_EW_LOAD_CR(5)
_EW_LOAD_CR(6)
_EW_LOAD_CR(7)
_EW_LOAD_CR(8)
_EW_LOAD_CR(9)
_EW_LOAD_CR(10)
_EW_LOAD_CR(11)
_EW_LOAD_CR(12)
_EW_LOAD_CR(13)
_EW_LOAD_CR(14)


/***************************************************************
 *                                                             *
 *                      Coprocessor store                      *
 *                                                             *
 ***************************************************************/

#define _EW_STORE_CR(N) void ew_store_cr##N(ecc_word_t* w) { \
    register int r1 , r2 ; \
    unsigned     mask ; \
 \
    /* Check sign */ \
    ECC_STORE_0(r1 , r2 , ECC_STATUS_REG) ; \
    mask = 1 << (0x10 + N) ; \
    if ( r1 & mask ) { \
        w->is_negative = true ; \
        ECC_NEG(#N, #N) ; \
    } \
    else \
        w->is_negative = false ; \
 \
    w->is_zero = true ; \
    /* Offset 0 */ \
    ECC_STORE_0(r1 , r2 , #N) ; \
    EW_SET_0(r1 , r2 , w) ; \
    /* Offset 1 */ \
    ECC_STORE_1(r1 , r2 , #N) ; \
    EW_SET_1(r1 , r2 , w) ; \
    /* Offset 2 */ \
    ECC_STORE_2(r1 , r2 , #N) ; \
    EW_SET_2(r1 , r2 , w) ; \
    /* Offset 3 */ \
    ECC_STORE_3(r1 , r2 , #N) ; \
    EW_SET_3(r1 , r2 , w) ; \
    /* Offset 4 */ \
    ECC_STORE_4(r1 , r2 , #N) ; \
    EW_SET_4(r1 , r2 , w) ; \
    /* Offset 5 */ \
    ECC_STORE_5(r1 , r2 , #N) ; \
    EW_SET_5(r1 , r2 , w) ; \
    /* Offset 6 */ \
    ECC_STORE_6(r1 , r2 , #N) ; \
    EW_SET_6(r1 , r2 , w) ; \
    /* Offset 7 */ \
    ECC_STORE_7(r1, r2, #N); \
    EW_SET_7(r1, r2, w); \
    /* Offset 8 */ \
    ECC_STORE_8(r1, r2, #N); \
    EW_SET_8(r1, r2, w); \
    /* Offset 9 */ \
    ECC_STORE_9(r1, r2, #N); \
    EW_SET_9(r1, r2, w); \
    /* Offset 10 */ \
    ECC_STORE_10(r1, r2, #N); \
    EW_SET_10(r1, r2, w); \
    /* Offset 11 */ \
    ECC_STORE_11(r1, r2, #N); \
    EW_SET_11(r1, r2, w); \
    /* Offset 12 */ \
    ECC_STORE_12(r1, r2, #N); \
    EW_SET_12(r1, r2, w); \
    /* Offset 13 */ \
    ECC_STORE_13(r1, r2, #N); \
    EW_SET_13(r1, r2, w); \
    /* Offset 14 */ \
    ECC_STORE_14(r1, r2, #N); \
    EW_SET_14(r1, r2, w); \
    /* Offset 15 */ \
    ECC_STORE_15(r1, r2, #N); \
    EW_SET_15(r1, r2, w); \
 \
    if ( w->is_negative ) \
        ECC_NEG(#N, #N); \
}

_EW_STORE_CR(0)
_EW_STORE_CR(1)
_EW_STORE_CR(2)
_EW_STORE_CR(3)
_EW_STORE_CR(4)
_EW_STORE_CR(5)
_EW_STORE_CR(6)
_EW_STORE_CR(7)
_EW_STORE_CR(8)
_EW_STORE_CR(9)
_EW_STORE_CR(10)
_EW_STORE_CR(11)
_EW_STORE_CR(12)
_EW_STORE_CR(13)
_EW_STORE_CR(14)

/* Store word from CP register 15. Does not care about sign since it's
   the status register */
void
ew_store_cr15(ecc_word_t* w)
{
    register int r1, r2;
    w->is_zero = true;
    /* Offset 0 */
    ECC_STORE_0(r1, r2, "15");
    EW_SET_0(r1, r2, w);
    /* Offset 1 */
    ECC_STORE_1(r1, r2, "15");
    EW_SET_1(r1, r2, w);
    /* Offset 2 */
    ECC_STORE_2(r1, r2, "15");
    EW_SET_2(r1, r2, w);
    /* Offset 3 */
    ECC_STORE_3(r1, r2, "15");
    EW_SET_3(r1, r2, w);
    /* Offset 4 */
    ECC_STORE_4(r1, r2, "15");
    EW_SET_4(r1, r2, w);
    /* Offset 5 */
    ECC_STORE_5(r1, r2, "15");
    EW_SET_5(r1, r2, w);
    /* Offset 6 */
    ECC_STORE_6(r1, r2, "15");
    EW_SET_6(r1, r2, w);
    /* Offset 7 */
    ECC_STORE_7(r1, r2, "15");
    EW_SET_7(r1, r2, w);
    /* Offset 8 */
    ECC_STORE_8(r1, r2, "15");
    EW_SET_8(r1, r2, w);
    /* Offset 9 */
    ECC_STORE_9(r1, r2, "15");
    EW_SET_9(r1, r2, w);
    /* Offset 10 */
    ECC_STORE_10(r1, r2, "15");
    EW_SET_10(r1, r2, w);
    /* Offset 11 */
    ECC_STORE_11(r1, r2, "15");
    EW_SET_11(r1, r2, w);
    /* Offset 12 */
    ECC_STORE_12(r1, r2, "15");
    EW_SET_12(r1, r2, w);
    /* Offset 13 */
    ECC_STORE_13(r1, r2, "15");
    EW_SET_13(r1, r2, w);
    /* Offset 14 */
    ECC_STORE_14(r1, r2, "15");
    EW_SET_14(r1, r2, w);
    /* Offset 15 */
    ECC_STORE_15(r1, r2, "15");
    EW_SET_15(r1, r2, w);
}
```

LISTING E.2: Source file for big number implementation of an ECCo word.

Appendix F

Benchmark & Test program

```

/****************************************************************
 *                                                              *
 *                        Control macros                        *
 *                                                              *
 ****************************************************************/

// #define ONLY_HELLOW  /* Only run a simple hello world */

/* Testing control macros */
// #define TEST_ARI         /* Test arithmetic module  */
// #define TEST_ARI_NOADD   /* Skip addition during arithmetic testing */
// #define TEST_ARI_NOMOD   /* Skip multiplication during arithmetic testing */
// #define TEST_ARI_NODIV   /* Skip division during arithmetic testing */
// #define TEST_ARI_NONEG   /* Skip negation during arithmetic testing */
// #define TEST_REGS        /* Test register bank reading/writing */

/* Benchmarking control macros */
#define BENCHMARK                              /* Disable anything but the benchmarking code */
// #define BENCHMARK_ECC_ADDITION              /* Perform additions with ECCo with minimal extra code */
// #define BENCHMARK_ANSSI_ADDITION            /* Perform additions with ANSSI lib with minimal extra code */
// #define BENCHMARK_ECC_MULTIPLICATION        /* Perform multiplication with ECCo with minimal extra code */
#define BENCHMARK_ANSSI_MULTIPLICATION  /* Perform multiplication with ANSSI lib with minimal extra code */
// #define BENCHMARK_ITERATIONS 1         /* Number of iterations during benchmarking */
// #define BENCHMARK_ITERATIONS 10         /* Number of iterations during benchmarking */
#define BENCHMARK_ITERATIONS 100          /* Number of iterations during benchmarking */

/* ANSSI libecc control macros */
#define ANSSI_LIBECC

/* Sanity checks of macros */
#if (defined(BENCHMARK_ECC_ADDITION)           && (defined(BENCHMARK_ANSSI_ADDITION) || defined(BENCHMARK_ECC_MULTIPLICATION) || defined(BENCHMARK_ANSSI_MULTIPLICATION))) || \
    (defined(BENCHMARK_ANSSI_ADDITION)         && (defined(BENCHMARK_ECC_ADDITION)   || defined(BENCHMARK_ECC_MULTIPLICATION) || defined(BENCHMARK_ANSSI_MULTIPLICATION))) || \
    (defined(BENCHMARK_ECC_MULTIPLICATION)     && (defined(BENCHMARK_ANSSI_ADDITION) || defined(BENCHMARK_ECC_ADDITION)         || defined(BENCHMARK_ANSSI_MULTIPLICATION))) || \
    (defined(BENCHMARK_ANSSI_MULTIPLICATION) && (defined(BENCHMARK_ANSSI_ADDITION) || defined(BENCHMARK_ECC_MULTIPLICATION) || defined(BENCHMARK_ECC_ADDITION)))
  #error("Only␣one␣BENCHMARK_␣macro␣can␣be␣defined␣at␣a␣time")
#endif
```

```

#if (defined(BENCHMARK_ANSSI_ADDITION) || defined(
    BENCHMARK_ANSSI_MULTIPLICATION)) && !defined(ANSSI_LIBECC)
 #error("ANSSI_LIBECC␣must␣be␣defined␣for␣ANSSI␣benchmarks")
#endif


/****************************************************************
 *                                                              *
 *                          Includes                            *
 *                                                              *
 ****************************************************************/

/* ARM CM33 */
#include <arm_cmse.h>
#include <cm4ss.h>
#include <ee_printf.h>
#include <cm33/secure/trustzone_util.h>

/* stdlib */
#include <stdbool.h>
#include <string.h>

/* Coprocessor */
#include "ecc.h"
#include "ecc_word.h"
#include "division_data.h"
#include "modular_addition_data.h"
#include "modular_multiplication_data.h"

/* ANSSI libecc */
#ifdef ANSSI_LIBECC
 #include "libarith.h"
#endif


/****************************************************************
 *                                                              *
 *                       Globals/Macros                         *
 *                                                              *
 ****************************************************************/

/* TZ_START_NS: Start address of non-secure application */
#ifndef TZ_START_NS
#define TZ_START_NS (0x80000U)
#endif

#define CPACR_ADDR ((unsigned*) 0xE000ED88U)


/****************************************************************
 *                                                              *
 *                         Test setup                           *
 *                                                              *
 ****************************************************************/

/* Arithmetic test functions */
bool test_ari_multiplication(char (*)[DATAMUL16_NUM_HEADERS][
    DATAMUL16_NUM_CHARS+1]);
bool test_ari_addition(char (*)[DATAADD16_NUM_HEADERS][DATAADD16_NUM_CHARS
    +1]);
bool test_ari_division(char (*)[DATADIV16_NUM_HEADERS][DATADIV16_NUM_CHARS
    +1]);

/* ANSSI libecc helpers */
#ifdef ANSSI_LIBECC
static void nn_import_from_hexbuf(nn_t out_nn, const char *hbuf, u32
    hbuflen);
#endif

/* Benchmark value strings */
```

```
char add_op1_str[] = "0x63feb1ab67e6b315a2dea87e6547ba17e0daa6009366d19f14dbb427faee50ae";
char add_op1_buf[] = {0x63, 0xfe, 0xb1, 0xab, 0x67, 0xe6, 0xb3, 0x15, 0xa2, 0xde, 0xa8, 0x7e, 0x65, 0x47, 0xba, 0x17, 0xe0, 0xda, 0xa6, 0x00, 0x93, 0x66, 0xd1, 0x9f, 0x14, 0xdb, 0xb4, 0x27, 0xfa, 0xee, 0x50, 0xae};
char add_op2_str[] = "0x2f08337b7ae05e16b4fada1ebbb4c7bb56009e5c141dc5b487db427faee50ae0";
char add_op2_buf[] = {0x2f, 0x08, 0x33, 0x7b, 0x7a, 0xe0, 0x5e, 0x16, 0xb4, 0xfa, 0xda, 0x1e, 0xbb, 0xb4, 0xc7, 0xbb, 0x56, 0x00, 0x9e, 0x5c, 0x14, 0x1d, 0xc5, 0xb4, 0x87, 0xdb, 0x42, 0x7f, 0xae, 0xe5, 0x0a, 0xe0};
char add_mod_str[] = "0xa41a41a12a799548211c410c65d8133afde34d28bdd542e4b680cf2899c8a8c4";
char add_mod_buf[] = {0xa4, 0x1a, 0x41, 0xa1, 0x2a, 0x79, 0x95, 0x48, 0x21, 0x1c, 0x41, 0x0c, 0x65, 0xd8, 0x13, 0x3a, 0xfd, 0xe3, 0x4d, 0x28, 0xbd, 0xd5, 0x42, 0xe4, 0xb6, 0x80, 0xcf, 0x28, 0x99, 0xc8, 0xa8, 0xc4};
char mul_op1_str[] = "0x63feb1ab67e6b315a2dea87e6547ba17e0daa6009366d19f14dbb427faee50ae";
char mul_op1_buf[] = {0x63, 0xfe, 0xb1, 0xab, 0x67, 0xe6, 0xb3, 0x15, 0xa2, 0xde, 0xa8, 0x7e, 0x65, 0x47, 0xba, 0x17, 0xe0, 0xda, 0xa6, 0x00, 0x93, 0x66, 0xd1, 0x9f, 0x14, 0xdb, 0xb4, 0x27, 0xfa, 0xee, 0x50, 0xae};
char mul_op2_str[] = "0x02f08337b7ae05e16b4fada1ebbb4c7bb56009e5c141dc5b487db427faee50ae";
char mul_op2_buf[] = {0x02, 0xf0, 0x83, 0x37, 0xb7, 0xae, 0x05, 0xe1, 0x6b, 0x4f, 0xad, 0xa1, 0xeb, 0xbb, 0x4c, 0x7b, 0xb5, 0x60, 0x09, 0xe5, 0xc1, 0x41, 0xdc, 0x5b, 0x48, 0x7d, 0xb4, 0x27, 0xfa, 0xee, 0x50, 0xae};
char mul_mod_str[] = "0xa41a41a12a799548211c410c65d8133afde34d28bdd542e4b680cf2899c8a8c4";
char mul_mod_buf[] = {0xa4, 0x1a, 0x41, 0xa1, 0x2a, 0x79, 0x95, 0x48, 0x21, 0x1c, 0x41, 0x0c, 0x65, 0xd8, 0x13, 0x3a, 0xfd, 0xe3, 0x4d, 0x28, 0xbd, 0xd5, 0x42, 0xe4, 0xb6, 0x80, 0xcf, 0x28, 0x99, 0xc8, 0xa8, 0xc4};

#define BM_STR_LEN 67
#define BM_BUF_LEN 32
#define BM_NN_LEN ((BM_STR_LEN / 2) / WORD_BYTES)


/**************************************************************
 *                                                            *
 *                        Secure main                         *
 *                                                            *
 **************************************************************/

int
main(void)
{
 #ifndef BENCHMARK
    MSG(("C-code:␣Secure␣firmware␣booting\n"));
    MSG((">>>>>>>>␣Running␣ECC␣firmware␣test.\n"));
 #endif

    /* Enable coprocessor */
    *CPACR_ADDR ^= 0x01;

 #ifdef ONLY_HELLOW

    MSG(("HELLO␣EC␣WORLD!\n"));

 #else

    /***************************
     *  Test arithmetic module  *
     ***************************/

  #ifdef TEST_ARI
    /* Modular addition */
   #ifndef TEST_ARI_NOADD
    MSG((">>>>␣Testing␣addition\n"));
    if ( test_ari_addition(dataAdd16) )
        MSG(("Success!\n"));
   #endif
    /* Modular multiplication */
   #ifndef TEST_ARI_NOMUL
```

```
      MSG((">>>>␣Testing␣multiplication\n"));
      if ( test_ari_multiplication(dataMul16) )
          MSG(("Success!\n"));
    #endif
      /* Division */
    #ifndef TEST_ARI_NODIV
      MSG((">>>>␣Testing␣division\n"));
      if ( test_ari_division(dataDiv16) )
          MSG(("Success!\n"));
    #endif
  #endif

    /************************************
     *  Benchmark modular addition w/CP  *
     ************************************/

  #ifdef BENCHMARK_ECC_ADDITION
    ecc_word_t op1, op2, mod;
    /* Set parameter values */
    ew_set_str(&op1, add_op1_str);
    ew_set_str(&op2, add_op2_str);
    ew_set_str(&mod, add_mod_str);
    /* Load parameters to CP */
    ew_load_cr0(&op1);
    ew_load_cr1(&op2);
    EW_LOAD_MOD(&mod);
    /* Perform N number of additions */
    for ( int i = 0; i < BENCHMARK_ITERATIONS; ++i )
        ECC_ADD("0", "1", "0");
  #endif

    /******************************************
     *  Benchmark modular addition in software  *
     ******************************************/

  #ifdef BENCHMARK_ANSSI_ADDITION
    nn nn_op1, nn_op2, nn_mod;
    fp fp_op1, fp_op2;
    fp_ctx fp_ctx;  /* Finite field context - size of field etc. */
    /* Initialize and set parameter values */
    nn_init_from_buf(&nn_op1, add_op1_buf, BM_BUF_LEN);
    nn_init_from_buf(&nn_op2, add_op2_buf, BM_BUF_LEN);
    nn_init_from_buf(&nn_mod, add_mod_buf, BM_BUF_LEN);
    fp_ctx_init_from_p(&fp_ctx, &nn_mod);
    fp_init(&fp_op1, &fp_ctx);
    fp_init(&fp_op2, &fp_ctx);
    fp_op1.fp_val = nn_op1;
    fp_op2.fp_val = nn_op2;
    /* Perform N number of additions */
    for ( int i = 0; i < BENCHMARK_ITERATIONS; ++i )
        fp_add(&fp_op1, &fp_op1, &fp_op2);
  #endif

    /*****************************************
     *  Benchmark modular multiplication w/CP  *
     *****************************************/

  #ifdef BENCHMARK_ECC_MULTIPLICATION
    ecc_word_t op1, op2, mod;
    /* Set parameter values */
    ew_set_str(&op1, mul_op1_str);
    ew_set_str(&op2, mul_op2_str);
    ew_set_str(&mod, mul_mod_str);
    /* Load parameters to CP */
    ew_load_cr0(&op1);
    ew_load_cr1(&op2);
    EW_LOAD_MOD(&mod);
    /* Perform N number of additions */
    for ( int i = 0; i < BENCHMARK_ITERATIONS; ++i )
        ECC_MUL("0", "1", "0");
  #endif

```

```
    /*************************************************
     *  Benchmark modular multiplication in software  *
     *************************************************/

  #ifdef BENCHMARK_ANSSI_MULTIPLICATION
    nn nn_op1, nn_op2, nn_mod;
    fp fp_op1, fp_op2;
    fp_ctx fp_ctx;  /* Finite field context - size of field etc. */
    /* Initialize and set parameter values */
    nn_init_from_buf(&nn_op1, mul_op1_buf, BM_BUF_LEN);
    nn_init_from_buf(&nn_op2, mul_op2_buf, BM_BUF_LEN);
    nn_init_from_buf(&nn_mod, mul_mod_buf, BM_BUF_LEN);
    fp_ctx_init_from_p(&fp_ctx, &nn_mod);
    fp_init(&fp_op1, &fp_ctx);
    fp_init(&fp_op2, &fp_ctx);
    fp_op1.fp_val = nn_op1;
    fp_op2.fp_val = nn_op2;
    /* Perform N number of additions */
    for ( int i = 0; i < BENCHMARK_ITERATIONS; ++i )
        fp_mul(&fp_op1, &fp_op1, &fp_op2);
  #endif

 #endif

 #ifndef BENCHMARK
    MSG((">>>>>>>> Finished ECC firmware test.\n\n"));
 #endif

    finish_test(TEST_PASS);
    return 0; // This line will never execute as boot_nonsec_program never
    returns
}


/***************************************************************
 *                                                             *
 *                       Test functions                        *
 *                                                             *
 ***************************************************************/

/**********************
 *  Arithmetic module  *
 **********************/

/* Modular addition */
bool
test_ari_addition(char (*data)[DATAADD16_NUM_HEADERS][DATAADD16_NUM_CHARS
    +1])
{
    int i = 0;
    char (*entry)[DATAADD16_NUM_CHARS+1];
    ew_str_t   mod_s, op1_s, op2_s, sol_s, res_s;
    ecc_word_t mod, op1, op2, sol, res;

    while ( i++ < DATAADD16_NUM_ENTRIES ) {
        entry = *data++;
        /* Set parameter values from data strings */
        if ( !ew_set_str(&mod, entry[0]) ) goto error;
        if ( !ew_set_str(&op1, entry[1]) ) goto error;
        if ( !ew_set_str(&op2, entry[2]) ) goto error;
        if ( !ew_set_str(&sol, entry[3]) ) goto error;
        /* Load parameters into CP registers */
        ew_load_cr0(&op1);
        ew_load_cr1(&op2);
        EW_LOAD_MOD(&mod);
        /* Perform addition */
        ECC_ADD("0", "1", "2");
        /* Verify result */
        ew_store_cr2(&res);
        if ( !ew_eq(&res, &sol) )
            goto wrong;
        MSG(("Test entry %d passed.\n", i));
    }
    return true;

 wrong:
    ew_to_str(&mod, mod_s, EW_STR_LENGTH);
    ew_to_str(&op1, op1_s, EW_STR_LENGTH);
    ew_to_str(&op2, op2_s, EW_STR_LENGTH);
    ew_to_str(&res, res_s, EW_STR_LENGTH);
    ew_to_str(&sol, sol_s, EW_STR_LENGTH);
    MSG(("␣␣␣␣␣%s\n"
         "␣␣␣+␣%s\n"
         "(mod␣%s)\n"
         "␣␣␣=␣%s\n"
         "␣got␣%s\n",
         op1_s, op2_s, mod_s, res_s, sol_s));
 error:
    MSG(("Failed...\n"));
    return false;
}

/* Modular addition */
bool
test_ari_multiplication(char (*data)[DATAMUL16_NUM_HEADERS][
    DATAMUL16_NUM_CHARS+1])
{
    int i = 0;
    char (*entry)[DATAMUL16_NUM_CHARS+1];
    ew_str_t   mod_s, op1_s, op2_s, sol_s, res_s;
    ecc_word_t mod, op1, op2, sol, res;

    while ( i++ < DATAMUL16_NUM_ENTRIES ) {
        entry = *data++;
        /* Set parameter values from data strings */
        if ( !ew_set_str(&mod, entry[0]) ) goto error;
        if ( !ew_set_str(&op1, entry[1]) ) goto error;
        if ( !ew_set_str(&op2, entry[2]) ) goto error;
        if ( !ew_set_str(&sol, entry[3]) ) goto error;
        /* Load parameters into CP registers */
        ew_load_cr0(&op1);
        ew_load_cr1(&op2);
        EW_LOAD_MOD(&mod);
        /* Perform addition */
        ECC_MUL("0", "1", "2");
        /* Verify result */
        ew_store_cr2(&res);
        if ( !ew_eq(&res, &sol) )
            goto wrong;
        MSG(("Test␣entry␣%d␣passed.\n", i));
    }
    return true;

 wrong:
    ew_to_str(&mod, mod_s, EW_STR_LENGTH);
    ew_to_str(&op1, op1_s, EW_STR_LENGTH);
    ew_to_str(&op2, op2_s, EW_STR_LENGTH);
    ew_to_str(&res, res_s, EW_STR_LENGTH);
    ew_to_str(&sol, sol_s, EW_STR_LENGTH);
    MSG(("␣␣␣␣␣%s\n"
         "␣␣␣*␣%s\n"
         "(mod␣%s)\n"
         "␣␣␣=␣%s\n"
         "␣got␣%s\n",
         op1_s, op2_s, mod_s, res_s, sol_s));
 error:
    MSG(("Failed...\n"));
    return false;
}

/* Modular addition */
bool
test_ari_division(char (*data)[DATADIV16_NUM_HEADERS][DATADIV16_NUM_CHARS
    +1])
```

```
{
    int i = 0;
    char (*entry)[DATADIV16_NUM_CHARS+1];
    ew_str_t   op1_s, op2_s, sol_s, res_s;
    ecc_word_t op1, op2, sol, res;

    while ( i++ < DATADIV16_NUM_ENTRIES ) {
        entry = *data++;
        /* Set parameter values from data strings */
        if ( !ew_set_str(&op1, entry[0]) ) goto error;
        if ( !ew_set_str(&op2, entry[1]) ) goto error;
        if ( !ew_set_str(&sol, entry[2]) ) goto error;
        /* Load parameters into CP registers */
        ew_load_cr0(&op1);
        ew_load_cr1(&op2);
        /* Perform addition */
        ECC_DIV("0", "1", "2");
        /* Verify result */
        ew_store_cr2(&res);
        if ( !ew_eq(&res, &sol) )
            goto wrong;
        MSG(("Test entry %d passed.\n", i));
    }
    return true;

 wrong:
    ew_to_str(&op1, op1_s, EW_STR_LENGTH);
    ew_to_str(&op2, op2_s, EW_STR_LENGTH);
    ew_to_str(&res, res_s, EW_STR_LENGTH);
    ew_to_str(&sol, sol_s, EW_STR_LENGTH);
    MSG(("     %s\n"
         "   / %s\n"
         "   = %s\n"
         " got %s\n",
         op1_s, op2_s, res_s, sol_s));
 error:
    MSG(("Failed...\n"));
    return false;
}
```

LISTING F.1: C main of test and benchmark program.

References

[1] N. Koblitz, “Elliptic curve cryptosystems”, *Math. Comp.*, vol. 48, pp. 203–209, 1987, ISSN: 0025-5718. DOI: 10.1090/S0025-5718-1987-0866109-5.

[2] V. S. Miller, “Use of elliptic curves in cryptography”, in *Advances in Cryptology — CRYPTO ’85 Proceedings*, H. C. Williams, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 417–426, ISBN: 978-3-540-39799-1.

[3] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996, ISBN: 0849385237.

[4] W. Diffie and M. Hellman, “New directions in cryptography”, *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976, ISSN: 0018-9448. DOI: 10.1109/TIT.1976.1055638.

[5] Y. Kumar, R. Munjal, and H. Sharma, “Comparison of symmetric and asymmetric cryptography with existing vulnerabilities and countermeasures”, *International Journal of Computer Science and Management Studies*, vol. 11, no. 03, 2011.

[6] R. Tripathi and S. Agrawal, “Comparative study of symmetric and asymmetric cryptography techniques”, *International Journal of Advance Foundation and Research in Computer (IJAFRC)*, vol. 1, no. 6, pp. 68–76, 2014.

[7] E. Rescorla. (2018). The transport layer security (tls) protocol version 1.3, [Online]. Available: https://tools.ietf.org/html/rfc8446 (visited on 11/09/2018).

[8] IEEE. (2017). Why we need low-power, low-latency devices, [Online]. Available: https://innovationatwork.ieee.org/why-we-need-low-power-low-latency-devices/ (visited on 06/26/2019).

[9] M. Guerra. (2017). The power of iot devices, [Online]. Available: https://www.electronicdesign.com/power/power-iot-devices (visited on 06/26/2019).

[10] N. Shields. (2017). Here’s how 5g will revolutionize the internet of things, [Online]. Available: https://www.businessinsider.com/how-5g-will-revolutionize-the-internet-of-things-2017-6?r=US&IR=T (visited on 06/26/2019).

[11] M. Hirth, *Hardware acceleration of asymmetric elliptic curve cryptography*, 2018.

[12] P. B. Bhattacharya, S. K. Jain, and S. Nagpaul, *Basic abstract algebra*, 2nd. Cambridge University Press, 1994, ISBN: 0521460816.

[13] B. Lynn. (). Modular arithmetic, [Online]. Available: https://crypto.stanford.edu/pbc/notes/numbertheory/arith.html (visited on 11/14/2018).

[14] Wikipedia. (2018). Extended euclidaen algorithm, [Online]. Available: https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm (visited on 11/14/2018).

[15] ——, (2018). Euclidaen algorithm, [Online]. Available: https://en.wikipedia.org/wiki/Euclidean_algorithm (visited on 11/14/2018).

[16] S. for Efficient Cryptography. (2009). Sec 1: Elliptic curve cryptography, [Online]. Available: http://www.secg.org/sec1-v2.pdf (visited on 12/19/2018).

[17] J. Balasch, B. Gierlichs, K. Ja¨rvinen, and I. Verbauwhede, "Hardware/-software co-design flavors of elliptic curve scalar multiplication", in *2014 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, Aug. 2014, pp. 758–763. DOI: 10.1109/ISEMC.2014.6899070.

[18] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates", in *Advances in Cryptology — ASIACRYPT'98*, K. Ohta and D. Pei, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 51–65, ISBN: 978-3-540-49649-6.

[19] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Berlin, Heidelberg: Springer-Verlag, 2003, ISBN: 038795273X.

[20] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. [Online]. Available: http://doi.acm.org/10.1145/359340.359342.

[21] A. P. Fournaris, I. Zafeirakis, C. Koulamas, N. Sklavos, and O. Koufopavlou, "Designing efficient elliptic curve diffie-hellman accelerators for embedded systems", in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2015, pp. 2025–2028. DOI: 10.1109/ISCAS.2015.7169074.

[22] Mentor. (2019). Questa® advanced simulator, [Online]. Available: https://www.mentor.com/products/fv/questa/ (visited on 06/19/2019).

[23] ——, (2019). Mentor, [Online]. Available: https://www.mentor.com/ (visited on 06/19/2019).

[24] "Ieee standard vhdl language reference manual", *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. c1–626, Jan. 2009. DOI: 10.1109/IEEESTD.2009.4772740.

[25] "Ieee standard for verilog hardware description language", *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, Apr. 2006. DOI: 10.1109/IEEESTD.2006.99495.

[26] "Ieee standard for systemverilog–unified hardware design, specification, and verification language", *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, pp. 1–1315, Feb. 2018. DOI: 10.1109/IEEESTD.2018.8299595.

[27] ARM. (2019). Cortex-m33, [Online]. Available: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m33 (visited on 06/19/2019).

[28] ——, (2019). Arm, [Online]. Available: https://www.arm.com/ (visited on 06/19/2019).

[29] ——, (2016). Armv8-m architecture reference manual, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0553a.d/index.html (visited on 06/26/2019).

[30] Wikipedia. (2019). Hardware acceleration, [Online]. Available: https://en.wikipedia.org/wiki/Hardware_acceleration (visited on 06/19/2019).

[31] R. Benadjila, A. Ebalard, and J.-P. Flori. (2017). Libecc project, [Online]. Available: https://github.com/ANSSI-FR/libecc (visited on 10/11/2018).

[32] Python Software Foundation. (2018). Python, [Online]. Available: https://www.python.org/ (visited on 11/21/2018).

[33] Python Docs. (2018). Python data model, [Online]. Available: https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy (visited on 11/20/2018).

[34] C. Koc, *Rsa hardware implementation, rsa laboratories, rsa data security, inc. august 1995*.

[35] J. K. Omura, "A public key cell design for smart card chips", *ISITA'90*, pp. 983–985, 1990.

[36] P. L. Montgomery, "Modular multiplication without trial division", *Math. Comp*, vol. 44, pp. 519–521, 1985. DOI: 10.1090/S0025-5718-1985-0777282-X.

[37] N. I. of Standards and Technology. (2013). Digital signature standards, [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf (visited on 09/09/2018).

[38] S. for Efficient Cryptography. (2010). Sec 2: Recommended elliptic curve domain parameters, [Online]. Available: http://www.secg.org/sec2-v2.pdf (visited on 09/09/2018).

[39] OpenCores. (2019). Opencores, [Online]. Available: https://opencores.org/ (visited on 07/01/2019).