Software optimization of binary elliptic curves arithmetic using modern processor architectures

Manuel Bluhm

June 17, 2013

Department of Mathematics, University of Haifa
Embedded Security Group, Ruhr University Bochum

Prof. Dr. Shay Gueron
Prof. Dr.-Ing. Christof Paar
Abstract This work provides an efficient and protected implementation of the binary elliptic curve point multiplication for the well-known standard curves suggested by NIST and SECG. In particular, the Intel SSE/AVX processor instruction set and the pclmulqdq instruction is used to enhance current state-of-the-art library OpenSSL. Additionally, we suggest improvements to the López-Dahab/Montgomery method on the algorithmic level. In comparison to the generic OpenSSL C implementation, we show that the proposed techniques sum up to a significant performance improvement of factor 6 to 9 for a side channel protected point multiplication in $GF(2^m)$, suggested by the NIST with $m \in \{163, 233, 283, 409, 571\}$. 
Contents

1. Introduction 1
   1.1. The Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.2. The Bottomline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2
   1.3. The Goal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

2. Cryptography on Binary Elliptic Curves 4
   2.1. Binary Finite Field Arithmetic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
       2.1.1. The Galois Field GF(2^m) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4
       2.1.2. Arithmetic in GF(2^m) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
       2.1.3. Field Element Representation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.2. Introduction to Binary Elliptic Curves . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
       2.2.1. Elliptic Curve Arithmetic . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
       2.2.2. Coordinate Systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
       2.2.3. EC Point Multiplication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
       2.2.4. Montgomery Point Multiplication . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
   2.3. Cryptography with Elliptic Curves . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
       2.3.1. Elliptic Curve Discrete Logarithm Problem (ECDLP) . . . . . . . . . . . . . . . 19
       2.3.2. Elliptic Curve Diffie-Hellman (ECDH) . . . . . . . . . . . . . . . . . . . . . . . 20
       2.3.3. Elliptic Curve Digital Signature Algorithm (ECDSA) . . . . . . . . . . . . . . . 21

3. Side Channel Protection and Algorithm Improvements 22
   3.1. Side Channel Protection . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
       3.1.1. Introduction to Side Channel Attacks . . . . . . . . . . . . . . . . . . . . . . . . 22
       3.1.2. Side Channel Attacks to ECC . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
       3.1.3. Protection Strategy . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
   3.2. Improvements of the 2P Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
       3.2.1. Improved Mdouble . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
       3.2.2. Improved Madd . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
       3.2.3. Complexity Results of Improved Method . . . . . . . . . . . . . . . . . . . . . . . . 29

4. Implementation 31
   4.1. The Intel Vector Extensions and PCLMUL . . . . . . . . . . . . . . . . . . . . . . . . . 31
       4.1.1. Architectures and History . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
       4.1.2. The Arsenal . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
   4.2. Binary Field Arithmetics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
       4.2.1. Addition . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
       4.2.2. Squaring . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
       4.2.3. Multiplication . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
       4.2.4. Reduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
       4.2.5. Inversion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
   4.3. OpenSSL Integration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
   4.4. Expected Implementation Benefits with AVX2 . . . . . . . . . . . . . . . . . . . . . . . . 46

5. Measurement and Results 48
   5.1. Measurement Guidelines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.2. Results .......................................................... 49
  5.2.1. Speed Results - Binary Field Arithmetic .................. 49
  5.2.2. Speed Results - Improved 2P Flow ...................... 53
  5.2.3. Speed Results - ECDH/ECDSA .......................... 54

5.3. Comparison .................................................... 57
  5.3.1. Comparison - Binary Field Arithmetic .................. 57
  5.3.2. Comparison - Point Multiplication ...................... 57
  5.3.3. Further Work ............................................. 58

6. Conclusion ....................................................... 59

A. Bibliography .................................................. 60

B. OpenSSL Patch ................................................. 64

C. XMM C Library .................................................. 139

D. XMM Test Utilities .............................................. 196

E. Declaration ..................................................... 202
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example for elliptic curves over an infinite field $K = \mathbb{R}$, $y^2 = x^3 - 4x + 3$</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Example for elliptic curves over a finite field $K = \mathbb{F}_{31}$, $y^2 = x^3 + 10x + 15 \mod 31$</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>Example for the EC Group Law: $P + \mathcal{O} = P$, $\mathcal{E}(\mathbb{R})$: $y^2 = x^3 - 4x + 1$</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Example for the EC Group Law: $P + Q = R$, $\mathcal{E}(\mathbb{R})$: $y^2 = x^3 - 4x + 1$</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Example for the EC Group Law: $2P = R$, $\mathcal{E}(\mathbb{R})$: $y^2 = x^3 - 4x + 1$</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>Simple ECDH Key Exchange</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Squaring technique in $GF(2^m)$ with $pshufb$</td>
<td>34</td>
</tr>
<tr>
<td>4.2</td>
<td>Reduction overflow without Inplace in $GF(2^{163})$</td>
<td>37</td>
</tr>
<tr>
<td>4.3</td>
<td>Example step for Shift&amp;Add Inplace Reduction in $GF(2^{163})$</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Example step of the re-combination strategy</td>
<td>40</td>
</tr>
<tr>
<td>4.5</td>
<td>Optimal operand shifting for multiplication</td>
<td>43</td>
</tr>
<tr>
<td>5.1</td>
<td>Speedups of point multiplications over binary curves</td>
<td>55</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.</td>
<td>Comparison of affine and projective versions</td>
<td>18</td>
</tr>
<tr>
<td>2.2.</td>
<td>Security parameter comparison for EC over (GF(2^m))</td>
<td>19</td>
</tr>
<tr>
<td>3.1.</td>
<td>X/Z Transformation</td>
<td>26</td>
</tr>
<tr>
<td>3.2.</td>
<td>Computational costs for the improved versions of M\texttt{double} and M\texttt{add}</td>
<td>29</td>
</tr>
<tr>
<td>3.3.</td>
<td>New numbers for the LD/Montgomery point multiplication \textit{without} SC countermeasures</td>
<td>29</td>
</tr>
<tr>
<td>3.4.</td>
<td>New numbers for the LD/Montgomery point multiplication \textit{with} SC countermeasures</td>
<td>30</td>
</tr>
<tr>
<td>4.1.</td>
<td>Load, Store and Memory - Intrinsics</td>
<td>32</td>
</tr>
<tr>
<td>4.2.</td>
<td>Arithmetic Intrinsics</td>
<td>32</td>
</tr>
<tr>
<td>4.3.</td>
<td>Memory Alignment Intrinsics</td>
<td>33</td>
</tr>
<tr>
<td>4.4.</td>
<td>Karatsuba multiplication strategies for different sizes</td>
<td>36</td>
</tr>
<tr>
<td>4.5.</td>
<td>NIST/SECG Reduction Polynomials</td>
<td>37</td>
</tr>
<tr>
<td>5.1.</td>
<td>Cycles for addition in (GF(2^m))</td>
<td>49</td>
</tr>
<tr>
<td>5.2.</td>
<td>Cycles for Reduction in (GF(2^m)) on Ivy Bridge</td>
<td>50</td>
</tr>
<tr>
<td>5.3.</td>
<td>Cycles for Reduction in (GF(2^m)) on Haswell</td>
<td>50</td>
</tr>
<tr>
<td>5.4.</td>
<td>Cycles for Squaring in (GF(2^m))</td>
<td>51</td>
</tr>
<tr>
<td>5.5.</td>
<td>Cycles for Multiplication in (GF(2^m))</td>
<td>51</td>
</tr>
<tr>
<td>5.6.</td>
<td>Cycles for Inversions and Division in (GF(2^m))</td>
<td>51</td>
</tr>
<tr>
<td>5.7.</td>
<td>Cycles for Lazy Reductions in (GF(2^m)), SQR + MUL</td>
<td>52</td>
</tr>
<tr>
<td>5.8.</td>
<td>Cycles for Lazy Reductions in (GF(2^m)), MUL + MUL</td>
<td>52</td>
</tr>
<tr>
<td>5.9.</td>
<td>Cost Saving for M\texttt{double}</td>
<td>53</td>
</tr>
<tr>
<td>5.10.</td>
<td>Data Veiling costs per round</td>
<td>53</td>
</tr>
<tr>
<td>5.11.</td>
<td>OpenSSL Point Multiplication in (GF(2^m))</td>
<td>54</td>
</tr>
<tr>
<td>5.12.</td>
<td>OpenSSL ECDH Operations over NIST curves</td>
<td>55</td>
</tr>
<tr>
<td>5.13.</td>
<td>OpenSSL ECDSA Sign/Verify in (GF(2^m))</td>
<td>56</td>
</tr>
<tr>
<td>5.14.</td>
<td>OpenSSL ECDSA Operations in (GF(2^m))</td>
<td>56</td>
</tr>
<tr>
<td>5.15.</td>
<td>Unknown Point Multiplication over NIST Curves</td>
<td>58</td>
</tr>
<tr>
<td>5.16.</td>
<td>Unknown Point Multiplication over Special Curves</td>
<td>58</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Binary Method (left-to-right) .................................................. 14
2. Montgomery Ladder method for binary EC .................................. 14
3. Montgomery's Method in affine coordinates ............................... 15
4. Doubling Algorithm $M_{\text{double}}$ ........................................ 16
5. Adding Algorithm $M_{\text{add}}$ .................................................. 16
6. Mxy Algorithm ........................................................................ 17
7. Montgomery’s Method in projective coordinates .......................... 17
8. ECDSA Signature generation ................................................... 21
9. ECDSA Signature verification .................................................. 21
10. Proposed data veiling method .................................................. 26
11. Point Doubling algorithm for random curves ............................. 28
12. Point Doubling algorithm for Koblitz curves ............................. 28
13. Algorithm for Point Addition .................................................. 29
14. Fast Squaring in $GF(2^{163})$ .................................................. 35
15. Reduction in $GF(2^{193})$ ....................................................... 39
16. Reduction in $GF(2^{233})$ ....................................................... 39
17. Reduction in $GF(2^{409})$ ....................................................... 39
18. Reduction in $GF(2^{409})$ ....................................................... 39
19. Reduction in $GF(2^{409})$ ....................................................... 39
20. Reduction in $GF(2^{409})$ ....................................................... 39
21. Reduction in $GF(2^{163})$ using PCLMULQDQ ......................... 43
22. Faster PCLMULQDQ-Reduction in $GF(2^{163})$ ....................... 44
23. Example inversion in $GF(2^{163})$ .......................................... 45
24. Mixed reduction in $GF(2^{163})$ .............................................. 47
1. Introduction

This chapter points out the need of secure web communication, the importance of side channel protected implementation and sums up the goals for this thesis.

1.1. The Motivation

As part of the challenges coming with the rise of computer technology, the internet presents a unique set of security threats and risks. Consumers exchange private information, purchase, trade and thus submit delicate information over webpages, an utterly insecure channel. In order to address these issues and allow trustworthy connections, the need for strong and reliable encryption is essential.

In 1994, Netscape\textsuperscript{1} established the Secure Sockets Layer (SSL) protocol using public key (asymmetric) cryptography for key establishment and authentication, symmetric encryption to assert confidentiality and message authentication codes for integrity. It uses certificates, issued by trusted authorities and distributed through web browsers and is located right below the application layer in the OSI\textsuperscript{2} model. Nowadays, SSL and its successor TLS (Transport Layer Security) are widely used in several application types such as web browsing, email, instant messaging and voice-over-ip. TLS was first defined in 1999 and last updated in 2008 (RFC5246\textsuperscript{1}) and 2011 (RFC6176\textsuperscript{2}) and is a standards track protocol of the Internet Engineering Task Force (IETF).

Without going too deep into detail, a simple TLS web connection establishment basically runs as follows.

1. A browser requests to connect to a website secured with TLS
2. Server and client agree upon a cipher suite
3. The server sends a certificate
4. The browser authenticates this certificate
5. Server and Client establish a shared secret
6. The following session is encrypted using this secret

In general, the public key cryptography used to establish the shared secret (step 5) is the computationally most expensive part. Depending on the cipher suite, this step usually involves a Diffie-Hellman key exchange based on a variety of the discrete logarithm problem.

Considering the use of Elliptic Curve Diffie Hellman (ECDH, see 2.3.2), both the client and server need to conduct an expensive operation called point multiplication (see 2.2.1). Although requiring less performance than for a comparable RSA-operation \textsuperscript{10}, this operation requires serious computational effort for both server and client, whereas the server is usually required to process a huge number of handshakes simultaneously. Therefore, increasing the performance of the point multiplication directly affects the ability to perform more handshakes at the same time and thus influences the overall performance of the server. This is not only beneficial in terms of costs and environmental issues but also decreases the servers vulnerability against Denial-of-Service (DOS) attacks.

In addition, the SSL/TLS server is a crucial part of a secure web architecture, making its application a major target for malicious adversaries. A secure implementation of the key exchange is an essential part of the design, since if an adversary is able to exploit side channels to gain information about the shared secret, the whole communication will be compromised. Recently, several works such as \textsuperscript{17} have

\textsuperscript{1}Netscape Communications (formerly known as Netscape Communications Corporation)
\textsuperscript{2}The Open Systems Interconnection (OSI) model (ISO/IEC 7498-1)
demonstrated that such attacks are not theoretical but practical threats and pointed out the relevance of taking implementation and design flaws seriously.

1.2. The Bottomline

The OpenSSL project is an Open-Source toolkit implementing the Secure Socket Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols, using a comprehensive library of cryptographic primitives. It is written in C and assembly, maintained by a worldwide community of volunteers and available for almost all Unix derivatives (e.g., Linux, Solaris, BSD, MAC OS X), OpenVMS, Windows and an IBM port for System i (OS/400). The root of OpenSSL is the former SSLeay library, implemented by Eric A. Young and Tim J. Hudson.

The OpenSSL library is licensed in two different ways, the OpenSSL license, which is an Apache License 1.0, and the BSD-style SSLeay license. This _dual-license_ means that one is free to choose which license to use for both commercial and non-commercial usage.

OpenSSL is a common library, used by over 6 billion web servers on the internet[^15]. It is the basis for the mod_ssl module of the market leading Apache webserver. Regarding elliptic curve cryptography, OpenSSL implements the ECDHE-ECDSA and ECDHE-RSA, as well as the ECDH-ANON protocols. The EC library is generic and thus working for elliptic curves over both prime and binary fields.

In the following we list the binary curves we selected for this work to improve. These well known curves, standardized by NIST[^3] and SECG[^4], are already implemented in OpenSSL in a generic way, written in C and some assembly (e.g., for multiplication).

- sect163k1: NIST/SECG/WTLS curve over a 163 bit binary field
- sect163r1: SECG curve over a 163 bit binary field
- sect163r2: NIST/SECG curve over a 163 bit binary field
- sect193r1: SECG curve over a 193 bit binary field
- sect193r2: SECG curve over a 193 bit binary field
- sect233k1: NIST/SECG/WTLS curve over a 233 bit binary field
- sect233r1: NIST/SECG/WTLS curve over a 233 bit binary field
- sect239k1: SECG curve over a 239 bit binary field
- sect283k1: NIST/SECG curve over a 283 bit binary field
- sect283r1: NIST/SECG curve over a 283 bit binary field
- sect409k1: NIST/SECG curve over a 409 bit binary field
- sect409r1: NIST/SECG curve over a 409 bit binary field
- sect571k1: NIST/SECG curve over a 571 bit binary field
- sect571r1: NIST/SECG curve over a 571 bit binary field

These 14 curves offer a security level from about 80 to 256 bits and are therefore useful for a big utilization range from low to very high security applications.

1.3. The Goal

In this thesis, we decided to integrate our work into OpenSSL as a representative of “real world” libraries. This leads to performance losses, due to the overhead which necessarily comes along with a full strength library and makes comparison with other, solely academic works, more complicated. On the other hand, we will be able to see the actual impact of the improved techniques and methods used in this thesis without being limited to scholarly environments. Despite that, we will benefit from the generic implementation of the point arithmetic, in order to support many different binary fields with a fast implementation.

[^3]: The National Institute of Standards and Technology (NIST) is an agency of the U.S. Department of Commerce.
In this thesis, we are aiming to improve both performance and implementation security for the elliptic curve point multiplication for binary elliptic curves. By providing a library for the underlying binary field arithmetic for the most common fields, specified by the well known institutions NIST and SECG, we want to contribute our work to a real world application. Since our code is required to be fast, we use compiler intrinsics for the Intel SSE/AVX instruction set, together with the `pclmulqdq` instruction to enhance performance on general purpose CPU’s, commonly used in today’s server architectures. Additionally, we aim to eliminate serious side channels by data independent processing.
2. Cryptography on Binary Elliptic Curves

This chapter provides with the necessary mathematical background of binary extension fields, explains the discrete logarithm problem for elliptic curves and briefly introduces public key cryptography with elliptic curves.

2.1. Binary Finite Field Arithmetic

In this section, we make mathematical preliminaries and describe the arithmetic and representations for elements in binary extension fields.

2.1.1. The Galois Field \( GF(2^m) \)

Definition 1: Abelian Group

An algebraic group \( (G, \circ) \) is a set of elements \( G \) together with a group operation \( \circ : G \times G \to G \), satisfying the group axioms (1-4). Any group additionally holding condition (5) is called abelian group.

1. Algebraic Closure \( a \circ b \in G \) \hfill (2.1)
2. Identity Elements \( \exists e \in G : \ a \circ e = a \) \hfill (2.2)
3. Associativity \( (a \circ b) \circ c = a \circ (b \circ c) \) \hfill (2.3)
4. Inverse Elements \( \exists a^{-1} \in G : \ a \circ a^{-1} = e \) \hfill (2.4)
5. Commutativity \( a \circ b = b \circ a \) \hfill (2.5)

for all \( a, b, c \in G \).

Definition 2: Field

A field is a triple \( F = (F, +, \cdot) \) with any nonempty set \( F \), if it satisfies the axiom

6. Distributivity \( a \cdot (b + c) = a \cdot b + a \cdot c \) \hfill (2.6)

for all \( a, b, c \in F \) and form abelian groups \( (F, +) \) with \( e = 0 \), and \( (F \setminus \{0\}, \cdot) \) with \( e = 1 \).
Definition 3: Finite Field (Galois Field)

Further, we call a field \( \mathbb{F} \) finite if the order (number of field elements) is finite. The order of a finite field (or \textbf{Galois Field}) is always a power of a prime \( p^m \) with prime \( p \) called characteristic and a positive integer \( m \in \mathbb{N} \). In this thesis, we denote such a field \( GF(p^m) \) rather than \( \mathbb{F}_{p^m} \). Up to isomorphisms, there exists exactly one finite field \( GF(p^m) \) for each prime power \( p^m \).

A representation of a finite field \( GF(p^m) \) can be given with an irreducible polynomial \( f(x) \) of degree \( m \) over \( GF(p) \) as follows:

\[
GF(p^m) = GF(p)[X] / \langle f(x) \rangle
\]  

(2.7)

with

\[
f(x) = x^m + a_{m-1}x^{m-1} + \ldots + a_0, \quad a_{0 \leq i \leq m-1} \in GF(p).
\]  

(2.8)

Since the minimal reduction polynomial \( f(x) \) cannot be factored as a product of polynomials of degree less than \( m \) (Irreducibility), the ideal \( \langle f(x) \rangle \) generated by \( f(x) \) divides the polynomial ring \( GF(p)[X] \) into a residue class ring of order \( p^m \). For more detailed information about these terms or finite fields in general, we refer the reader to the standard literature such as [11].

Definition 4: Binary Extension Field \( GF(2^m) \)

The smallest finite field satisfying the field axioms given in (2.1.1) is \( GF(2) \), consisting of two elements 0 and 1. The addition, as well as the subtraction, is an exclusive OR (XOR) and multiplication equals the logical AND. A field \( GF(2^m) \) with \( m \geq 2 \) and order \( 2^m \) is called \textbf{binary extension field}. According to (2.7), the binary extension field is defined as

\[
GF(2^m) = \{ a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_0 \mid a_{0 \leq i \leq m-1} \in GF(2) \}.
\]  

(2.9)

2.1.2. Arithmetic in \( GF(2^m) \)

With the background from the previous section, we can now focus on binary field arithmetic, which is essential for the techniques used in the implementation. Hereby, a field element \( a(x) \) is denoted as \( a(x) = \sum_{i=0}^{m-1} a_i x^i \).

Addition: Since addition and subtraction are the same in \( GF(2) \), the addition of \( a(x), b(x) \in GF(2^m) \) is defined as

\[
c(x) = a(x) + b(x) = \sum_{i=0}^{m-1} a_i x^i + \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} (a_i \oplus b_i) x^i
\]  

(2.10)

Please note that addition in \( GF(2^m) \) always equals the exclusive OR and can be denoted as both “+” or “⊕” in this work.

Multiplication: The multiplication in binary extension fields is a multiplication, modulo the irreducible reduction polynomial. Let \( a(x), b(x) \in GF(2^m) \), the multiplication is then defined as

\[
c(x) = a(x) \cdot b(x) = \sum_{i=0}^{m-1} a_i x^i \cdot \sum_{i=0}^{m-1} b_i x^i = \sum_{i=0}^{m-1} \left( a_i x^i \otimes b(x) \right) \mod f(x)
\]  

(2.11)
Squaring: Rather than just multiplying a field element with itself, the square of a field element \( a(x) \in GF(2^m) \) can be computed as follows:

\[
a^2(x) = \left( \sum_{i=0}^{m-1} a_i x^i \right)^2 = \sum_{i=0}^{m-1} (a_i x^i)^2 = \sum_{i=0}^{m-1} a_i x^{2i} \mod f(x)
\]

(2.12)

This makes the squaring in \( GF(2^m) \) a linear operation and therefore much faster than the conventional multiplication. The result of a square is twice as long as a field element and needs to be reduced with the field polynomial \( f(x) \).

Inversion: The inverse of an element \( a(x) \in GF(2^m) \) is defined to hold the condition (2.2). The two most popular methods for inversion in binary fields are based on either using the

1. Euclidean algorithm or
2. properties of cyclic groups.

In this thesis, we are going to use the Itoh-Tsujii inversion algorithm, which is based on the second group because of its constant time properties.

The cyclic group property utilized for the inversion is based on Lagrange’s theorem. This theorem states that the order of every subgroup \( H \) in any finite group \( G \) divides the group order; the order of any element \( a \) also divides the order of the group. The order of an element \( a \) is the smallest integer \( n \) that satisfies \( a^n = e \) where \( e \) is the identity of the group. Therefore, for any member \( a \) of a cyclic group \( G \), it follows \( a^{\deg G} = e \) and \( a^{\deg G - 1} = a^{-1} \). Hence, the inverse of an element \( a(x) \in GF(2^m) \) can be computed as follows:

\[
a^{-1}(x) = a^{2^m-2}(x) = (a^2)^{2^{m-1}-1}(x) \mod f(x).
\]

(2.13)

Reduction: As we have seen, the result of a field multiplication and squaring has twice the size of a regular field element and therefore needs to be reduced.

Let \( f(x) = x^m + r(x) \) with degree \( \deg r = k \) be the reduction polynomial over \( GF(2^m) \). We split the polynomial \( c(x) = c_H(x) + c_L(x) \) with degree \( \deg c = m + t \) into two parts where

\[
c_H(x) = \sum_{i=m}^{t} c_i x^i \quad \text{and} \quad c_L(x) = \sum_{i=0}^{m-1} c_i x^i.
\]

The reduction of \( c(x) \) is then computed as

\[
c'(x) = c_L(x) \oplus c_H(x) \cdot r(x).
\]

(2.14)

The resulting polynomial \( c'(x) \) has degree \( d = \deg c' = \max(m - 1, t + k) \). In case of \( d \geq m \), the reduction is applied recursively.
2.1.3. Field Element Representation

Basically, there are two common ways to represent a field element \( a(x) \in GF(2^m) \), the polynomial and the normal basis representation.

**Polynomial Representation**  In the polynomial representation, each coefficient \( a_i \in GF(2) \) from the polynomial \( a(x) = \sum_{i=0}^{m-1} a_i x^i \) corresponds to a bit in a binary string of length \( m \) such that \( a(x) = (a_{m-1}, a_{m-2}, ..., a_1, a_0) \). Thus, with a processor word size of \( w \), the number of words to store one field element is \( W = \lceil \frac{m}{w} \rceil \).

**Normal Basis Representation**  It is known that there exists an element \( \beta \in GF(2^m) \) so that the set \( \{\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}\} \) forms a basis (called normal basis) of \( GF(2^m) \) over \( GF(2) \). A field element \( a \in GF(2^m) \) is then represented as \( \sum_{i=0}^{m-1} a_i \beta^{2^i} \) with \( a_i \in GF(2) \). The software representation follows as above.

**Comparison**  Both representations are equal in matter of storage and computational costs for addition of field elements (XOR). However, the costs for multiplication and squaring are far more important. Although squaring and square root operations are only a cyclic shift in normal basis, multiplication usually requires much more effort in software implementations [23]. As we have seen before (2.12), squaring as well as square roots in \( GF(2^m) \) are linear time operations. Normal basis representations are usually used in hardware implementations and allow the design of efficient **bit-serial multipliers** such as described by Massey and Omura in [35]. In this work though, field elements are represented in polynomial basis only due to the domination of multiplications (see 2.1) and the fast squaring mechanism.
2.2. Introduction to Binary Elliptic Curves

In this section we briefly introduce the most important definitions and properties of binary elliptic curves. For these and further explanations we refer the reader to [21](chapter 13) and [32](chapter 3).

Definition 5: Elliptic Curve

In general, an elliptic curve $E$ is a smooth, algebraic curve constructed over any field. Let $K$ be such a field. An elliptic curve over $K$ can then be defined as the set of solutions $(x,y) \in K \times K$ to the Weierstraß equation

$$y^2 + a_0 xy + a_1 y = x^3 + a_2 x^2 + a_3 x + a_4,$$  

where $a_i$ are some constants in $K$.

Definition 6: Elliptic Curve Isomorphism

Let $E_1$ and $E_2$ be elliptic curves over $K$ defined by the Weierstraß equation as

$$E_1(K) : y^2 + a_0 xy + a_1 y = x^3 + a_2 x^2 + a_3 x + a_4$$

$$E_2(K) : y^2 + \bar{a}_0 xy + \bar{a}_1 y = x^3 + \bar{a}_2 x^2 + \bar{a}_3 x + \bar{a}_4$$

(2.16)

The two curves $E_1$ and $E_2$ are called isomorphic, if there exist elements $u, r, s, t \in K$ such that the isomorphism

$$\phi : E_1 \rightarrow E_2$$

$$(x, y) \mapsto (u^2 x + r, u^3 y + u^2 s x + t)$$  

(2.17)

transforms $E_1$ into $E_2$.

Let $K$ be a finite field of characteristics $\text{char}(K) = 2$ and the discriminant $\Delta = b$ is nonzero, equation 2.15 can then be transformed with the linear variable transformation

$$(x, y) \mapsto \left( a_2^2 x + \frac{a_3}{a_1}, a_1^3 y + a_1^2 a_4 + \frac{a_2^3}{a_1^2} \right)$$

(2.18)

into

$$y^2 + xy = x^3 + ax^2 + b, \quad a, b \in GF(2^m), \ b \neq 0$$

(2.19)

which is called non supersingular (see 2.2.1) elliptic curve of nonzero j-invariant.

Furthermore, the set of solutions to this equation plus an identity element $O$, called point at infinity, form together with an operation “+” a finite abelian group $G_E$. This group operation is called Point Addition and defined in the next section.

Definition 7: Elliptic Curve over $GF(2^m)$

Finally, we define the binary elliptic curve over $K = GF(2^m)$ as

$$E(K) = \{O\} \cup \{ (x, y) \in K \times K \mid y^2 + xy = x^3 + ax^2 + b \}$$

(2.20)

where $a, b \in K$ are constants and $b \neq 0$. 

8
Further, we call

$$E(K) = \begin{cases} 
Koblitz \text{ curve} & \text{if } a \in \{0, 1\} \text{ and } b = 1 \\
Random \text{ curve} & \text{otherwise.}
\end{cases} \quad (2.21)$$

Figure 2.1.: Example for elliptic curves over an infinite field $K = \mathbb{R}$, $y^2 = x^3 - 4x + 3$

Figure 2.2.: Example for elliptic curves over a finite field $K = \mathbb{F}_{31}$, $y^2 = x^3 + 10x + 15 \mod 31$
### 2.2.1. Elliptic Curve Arithmetic

In this section we will describe the basic arithmetic on elliptic curves and give examples for elliptic curves over real numbers to demonstrate its construction.

**Definition 8: The Group Law**

Let \( P = (x_p, y_p) \) and \( Q = (x_q, y_q) \) be points on the curve with \( P \neq O \). The inverse of a point \( P \) is defined as \(-P = (x_p, x_p + y_p)\). The operations \( P + Q \) (Point Addition) and \( P + P = 2P \) (Point Doubling) are further defined as

**Addition with \( \emptyset \)**

- \( Q = \emptyset \): \( P + \emptyset = P \)
- \( Q = (-P) \): \( P - P = \emptyset \)

**Figure 2.3.**: Example for the EC Group Law: \( P + \emptyset = P \), \( \mathcal{E}(\mathbb{R}) : y^2 = x^3 - 4x + 1 \)

**Point Addition**

- \( Q \neq P \): \( P + Q = R \)
  
  \[
  \begin{align*}
  \lambda &= \frac{y_q + y_p}{x_p + x_q} \\
  x' &= \lambda^2 + \lambda + x_p + x_q + a \\
  y' &= \lambda(x_p + x' + x'y_p)
  \end{align*}
  \]

(2.22)

**Figure 2.4.**: Example for the EC Group Law: \( P + Q = R \), \( \mathcal{E}(\mathbb{R}) : y^2 = x^3 - 4x + 1 \)
**Definition 9: Point Multiplication**

Finally, the *point* or *scalar multiplication* is the \( k \)-times repeated addition of a point \( P \in \mathcal{E} \) to itself, for any positive integer \( k \).

\[
Q = k \cdot P = P + P + \ldots + P = \sum_{i=0}^{k-1} P
\]  
(2.24)

**Definition 10: Elliptic Curves Group/Point Order**

The amount of points satisfying \( \mathcal{E}(K) \) is called *group order* and denoted as \( |\mathcal{E}(K)| \). Since the Weierstraf equation (2.15) has at most two solutions for each \( x \in K \), we know that \( |\mathcal{E}(K)| \leq 2p^m + 1 \). The following theorem from Hasse gives us tighter bounds.

**Hasse’s Theorem** Let \( \mathcal{E} \) be an elliptic curve over \( K = GF(p^m) \), then follows

\[
|\mathcal{E}(K)| \leq p^m + 1 - t, \quad |t| \leq 2\sqrt{p^m}
\]  
(2.25)

where \( t \) is the trace of \( \mathcal{E}(K) \).

**Supersingularity** Additionally, \( \mathcal{E}(K) \) is called *supersingular* if \( p \mid t \) and *non-supersingular* otherwise.

Furthermore, the curve order \( |\mathcal{E}(K)| \) is the product of its subgroups order \( |G_{\mathcal{E}}| \cdot |H_{\mathcal{E}}| \) (Langrange’s Theorem). In cryptographic applications, the order \( n = |G_{\mathcal{E}}| \) of the group generator \( G \) is normally prime. The cofactor \( h = |H_{\mathcal{E}}| \) is optimally one, but usually smaller than 5 for cryptographic purposes due to the vulnerability to Weil-Decent attacks ([29], [45]). For a similar reason, \( m \) should be prime as well, since \( |E(GF(2^l))| \) divides \( |E(GF(2^m))| \) if \( l \) divides \( m \), allowing to mount further attacks as small subgroup or invalid curve attacks [32]. Thus, we need to assert that \( P \in G_{\mathcal{E}} \) for point multiplication in cryptographic applications.
2.2.2. Coordinate Systems

Affine Coordinates

The representation of an elliptic curve point with two coordinates \( x \) and \( y \) is called **affine**. In general, its advantage is a low number of multiplications, its disadvantage a large number of inversions. Depending on the field element representation and the hardware architecture, the inversion is a very costly operation from a computational point of view. Therefore, it is often useful to switch to a different coordinate representation such as projective coordinates to decrease the number of inversions.

Projective Coordinates

Let \( K \) be a field and \( c, d \in \mathbb{N} \). One can define an equivalence relation \( \sim \) on the set \( K^3 \setminus \{(0,0,0)\} \) of nonzero triples over \( K \) by \((X_0, Y_0, Z_0) \sim (X_1, Y_1, Z_1)\) if \( X_0 = \lambda^c X_1, \ Y_0 = \lambda^d Y_1, \ Z_0 = \lambda Z_1, \ \lambda \in K^* \), where \( K^* = K \setminus \{0\} \).

The equivalence class containing the **representative** \((X,Y,Z) \in K^3 \setminus \{(0,0,0)\}\) is called **projective point** and denoted as

\[
(X : Y : Z) := \{(\lambda^c X, \lambda^d Y, \lambda Z) : \lambda \in K^*\}.
\]

(2.26)

Further, we denote the set of all projective points as \( \mathbb{P} = \mathbb{P}^* \cup \mathbb{P}^0 \) where

\[
\mathbb{P}^* = \{(X : Y : Z) : X,Y,Z \in K, Z \neq 0\}
\]

\[
\mathbb{P}^0 = \{(X : Y : Z) : X,Y,Z \in K, Z = 0\}
\]

(2.27)

Hence, the set of projective points \( \mathbb{P}^* \) corresponds to the set of affine points \((x, y) \in K \times K\) and the **line at infinity** \( \mathbb{P}^0 \) to the **point at infinity** \( \mathbb{O} \). The projective form of the elliptic curve \( E(K) \) is obtained by substituting the coordinates \( x \) by \( X/Z^c \) and \( y \) by \( Y/Z^d \) and clearing denominators.

Let \( K = GF(2^m) \) be a finite field and the elliptic curve \( E(K) \) as defined in (2.15). Different projective coordinates have been proposed in the past. We will briefly introduce and give detailed information about the point addition formulas for the López-Dahab coordinates [44] here. The sign “\( \equiv \)” indicates the correspondence to the affine coordinates.

**Standard Projective Coordinates** \((c = d = 1)\)

- **Projective Point**: \((X, Y, Z), Z \neq 0 \equiv (X/Z, Y/Z)\)
- **Projective Curve**: \( \mathcal{E}_{SP} : Y^2Z + XYZ = X^3 + aX^2Z + bZ^3 \)
- **Line at Infinity**: \((0, 1, 0) \equiv \mathbb{O} \)
- **Inverse Point**: \((-X, Y, Z) = (X, X + Y, Z)\)

**Jacobian Projective Coordinates** \((c = d = 2)\)

- **Projective Point**: \((X, Y, Z), Z \neq 0 \equiv (X/Z^2, Y/Z^2)\)
- **Projective Curve**: \( \mathcal{E}_{JP} : Y^2 + XYZ = X^3 + aX^2Z^2 + bZ^6 \)
- **Line at Infinity**: \((1, 1, 0) \equiv \mathbb{O} \)
- **Inverse Point**: \((-X, Y, Z) = (X, X + Y, Z)\)

**López-Dahab Projective Coordinates** \((c = 1 \text{ and } d = 2)\)

- **Projective Point**: \((X, Y, Z), Z \neq 0 \equiv (X/Z, Y/Z^2)\)
- **Projective Curve**: \( \mathcal{E}_{SP} : Y^2Z + XYZ = X^3Z + aX^2Z + bZ^4 \)
- **Line at Infinity**: \((1, 0, 0) \equiv \mathbb{O} \)
- **Inverse Point**: \((-X, Y, Z) = (X, X + Y, Z)\)
Point Addition in López-Dahab Coordinates Let $P = (X_p : Y_p : Z_p)$ and $Q = (X_q : Y_q : Z_q)$ be distinctive points, the sum of $P$ and $Q$, $R = (X_r : Y_r : Z_r)$ can be computed as follows.

$$
\begin{align*}
t_0 & \leftarrow Y_q \cdot Z_p^2 + Y_p \\
t_1 & \leftarrow X_q \cdot Z_p + X_p \\
t_2 & \leftarrow Z_p \cdot t_1 \\
t_3 & \leftarrow t_1 \cdot (t_2 + aZ_p^2) \\
& \triangleright Z_r \leftarrow t_3^2 \\
t_4 & \leftarrow t_0 \cdot t_2 \\
& \triangleright X_r \leftarrow t_0^2 + t_3 + t_4 \\
t_5 & \leftarrow X_r + X_q \cdot Z_r \\
t_6 & \leftarrow Z_r^2 \cdot (X_q + Y_q) \\
& \triangleright Y_r \leftarrow t_5 \cdot (t_4 + Z_r) + t_6 
\end{align*}
$$

Point Doubling in López-Dahab Coordinates Let $P = (X_p : Y_p : Z_p)$ and $Q = P$. The point doubling of $P$, $R = (X_r : Y_r : Z_r) = 2P$ can be computed as follows.

$$
\begin{align*}
& \triangleright Z_r \leftarrow X_p^2 \cdot Z_p^2 \\
& \triangleright X_r \leftarrow X_p^4 + b \cdot Z_p^4 \\
& \triangleright Y_r \leftarrow bZ_p^4 \cdot Z_r + X_r \cdot (aZ_r + Y_p^2 + bZ_p^4)
\end{align*}
$$

Please note that conversion from projective to affine coordinates needs an inversion but has only small computational relevance. These definitions and further details can be found at [32].

2.2.3. EC Point Multiplication

The elliptic curve point multiplication as described in 2.2.1 is the most crucial and important part of the elliptic curve cryptosystem. A lot of efforts have been made to find methods and algorithms for the efficient conduction of a point multiplication. At first, we want to briefly review techniques and give a target definition for this work.

The first point to consider is whether or not the techniques make use of precomputation. In this context, the term precomputation means to calculate intermediate multiples of a point in advance to speed up the point multiplication with table lookups. Several fast multiplication methods have been proposed in the past ([57], [25]), however, precomputation does require the knowledge about the points to multiply in advance to create the precomputation table. In order to protect the implementation against side channel attacks, it may be necessary to veil both table accesses and computations. Also, the memory consumption can be significant and often disqualifies these techniques especially for the use in embedded environments. Nonetheless, the speed up achieved by those methods is often significant.

Further, exploiting the form of the scalar $k$ from $Q = kP$ can increase the multiplication performance significantly, but also decrease the security as data dependent branches lead to insecure implementations. Please refer to section 3.1 for references and further details on this topic.

Also, the shape of the curves to implement is of great importance. If the implementation is made for a specific curve, many different methods and mathematical properties can be used to speed up the implementation (e.g., the Frobenius Automorphism for curves with special traces [9]), whereas generalizations, such as implementations for both Koblitz and Random curves, always come along with a trade-off.

As stated in the introduction, the aim of this work is to provide a fast, side channel protected point multiplication software implementation for Koblitz and Random curves without precomputation. In the following, we will introduce to state-of-the-art methods for the conduction of a point multiplication.
2.2.4. Montgomery Point Multiplication

Double&Add Method

Let us briefly re-consider the simple double&add approach to perform the scalar multiplication $Q = kP$ first. The basic idea of the binary double&add method is to scan the binary representation of the scalar $k$ either from left-to-right or right-to-left and perform one point addition ($Q = Q + P$) when the current bit is one, and one point doubling each iteration. This results in algorithm 1.

**Algorithm 1:** Binary Method (left-to-right)

**Input:** Point $P$, Integer $k = (1, k_{l-1}, ..., k_0)_2$

**Output:** Point $Q = kP$

1. $Q \leftarrow P$;
2. for $i \leftarrow (l - 1)$ to 0 do
3.     $Q \leftarrow 2Q$;
4.     if $k_i = 1$ then
5.         $Q \leftarrow Q + P$;
6.     end
7. end
8. return $Q = kP$

Montgomery Ladder Method

In [46], Peter L. Montgomery introduces a different approach to perform a scalar multiplication. Based on the binary method, it uses the relation $P_1 - P_0 = P$, where the sum of two points whose difference is known, can be evaluated with only involving the $x$-coordinate of these points. The resulting algorithm also has the convenient property of performing the same amount of additions and doubles for any key of the same length. Please note that the leading bit at position $l - 1$ is required to be one.

**Algorithm 2:** Montgomery Ladder method for binary EC

**Input:** Point $P$, positive integer $k = (1, k_{l-2}, ..., k_0)_2$

**Output:** Point $Q = kP$

1. $P_0 \leftarrow P$;
2. $P_1 \leftarrow 2P$;
3. for $i \leftarrow (l - 2)$ to 0 do
4.     if $k_i = 0$ then
5.         $P_1 \leftarrow P_0 + P_1$;
6.         $P_0 \leftarrow 2P_0$;
7.     else
8.         $P_0 \leftarrow P_0 + P_1$;
9.         $P_1 \leftarrow 2P_1$;
10. end
11. end
12. return $Q = P_0$
López-Dahab’s 2P Algorithm

In [43], López and Dahab introduced an improved version of the above algorithm for elliptic curves over \(GF(2^m)\).

Affine Version  In the following, we will give a summary of the improved algorithm in affine coordinates. The following lemmas show how to compute the \(x\)-coordinate for the addition of two points (Lemma 1) and how to extract the \(y\)-coordinate from a point \(P_1\) (Lemma 2).

Lemma 1:  Let \(P = (x_0, y_0)\), \(P_1 = (x_1, y_1)\) and \(P_2 = (x_2, y_2)\) be points on the elliptic curve \(E(GF(2^m))\) with \(P_0 = P_1 + P_2\). The \(x\)-coordinate \(x'\) of the sum \(P_1 + P_2\) can be computed as follows.

\[
\begin{align*}
\text{▶ } P_1 &\neq P_2: \\
x' &= x_0 + \left(\frac{x_1}{x_1 + x_2}\right)^2 + \left(\frac{x_1}{x_1 + x_2}\right) \\
(2.30)
\end{align*}
\]

\[
\begin{align*}
\text{▶ } P_1 &= P_2: \\
x' &= x_1^2 + \frac{b}{x_1^2} \\
(2.31)
\end{align*}
\]

Lemma 2:  Let \(P = (x_0, y_0)\), \(P_1 = (x_1, y_1)\) and \(P_2 = (x_2, y_2)\) be points on the elliptic curve \(E(GF(2^m))\) with \(P_0 = P_1 + P\) and \(x_0 \neq 0\). The \(y\)-coordinate of \(P_1\) can then be expressed as follows.

\[
y_1 = \frac{(x_0 + x_1) \cdot (x_0 + x_1)(x_0 + x_2) + x_0^2 + y_0}{x_0} + y_0 \\
(2.32)
\]

With the use of the Lemmas 1 and 2, the following algorithm shows the implementation of Montgomery’s method in affine coordinates.

---

Algorithm 3: Montgomery’s Method in affine coordinates

**Input:** Point \(P = (x, y)\), Integer \(k = (k_l-1, k_{l-2}, ..., k_0)\), \(k \geq 0\)

**Output:** Point \(Q = kP\)

1. if \(k = 0\) or \(x = 0\) then return \(Q = (0, 0)\);
2. \(x_0 \leftarrow x; \ x_1 \leftarrow x^2 + \frac{b}{x^2}\);
3. for \(i \leftarrow (l - 2)\) to 0 do
4. \(t \leftarrow \frac{x_0}{x_0 + x_1} ;\)
5. if \(k_i = 0\) then
6. \(x_0 \leftarrow x + t^2 + t;\)
7. \(x_1 \leftarrow x_1^2 + \frac{b}{x_1^2} ;\)
8. else
9. \(x_0 \leftarrow x_1^2 + \frac{b}{x_1^2} ;\)
10. \(x_1 \leftarrow x + t^2 + t;\)
11. end
12. end
13. \(t_1 \leftarrow x + x_0, t_2 \leftarrow x + x_1;\)
14. \(y_1 = t_1(t_1t_2 + x^2 + y)/x + y;\)
15. return \(Q = (x_1, y_1)\)
As we have seen in 2.2.2, we can transform the Point $P = (x, y)$ into the LD-projective form $P = (X/Z, Y/Z^2)$. The $x$-coordinate of the sum of $P_0 + P_1$ and $2P_i$ can be computed as the fraction $X'/Z'$ where

\[ X' = X_i^4 + b \cdot Z_i^4 \]
\[ Z' = X_i^2 \cdot Z_i^2 \]  

(2.33)

\[ Z' = \left( (Z_0 \cdot X_1) + (Z_1 \cdot X_0) \right) \]
\[ X' = x \cdot Z' + (Z_0 \cdot X_1) \cdot (Z_1 \cdot X_0) \]  

(2.34)

Algorithm 4 for point doubling (2.33) requires one general field multiplication, one field multiplication by the constant $c = b^{2^m - 1}$, four field squarings and one temporary variable $t$. In addition, it requires 6 field reductions in $GF(2^m)$.

Algorithm 4: Doubling Algorithm $\texttt{Mdouble}$

**Input:** $c \in GF(2^m)$ where $c^2 = b$, $x$-coordinate $X/Z$ for a point $P$

**Output:** $x$-coordinate $X/Z$ for the point $2P$

1. $t \leftarrow c$
2. $X \leftarrow X^2 \mod f(x)$
3. $Z \leftarrow Z^2 \mod f(x)$
4. $t \leftarrow Z \times t \mod f(x)$
5. $Z \leftarrow Z \times X \mod f(x)$
6. $t \leftarrow t^2 \mod f(x)$
7. $X \leftarrow X^2 \mod f(x)$
8. $X \leftarrow X + t$
9. return $X$, $Z$

Algorithm 5 for adding two points (2.34) requires three general field multiplications, one multiplication by $x$, which is fixed during the point multiplication, one field squaring, two temporary variables and five reductions.

Algorithm 5: Adding Algorithm $\texttt{Madd}$

**Input:** $x$-coordinates $X/Z$ of the point $P(x, y), P_0(X_0, Z_0), P_1(X_1, Z_1)$

**Output:** $x$-coordinate $X_1/Z_1$ for the point $P_0 + P_1$

1. $t_0 \leftarrow x$
2. $X_0 \leftarrow X_0 \times Z_1 \mod f(x)$
3. $Z_0 \leftarrow Z_0 \times X_1 \mod f(x)$
4. $t_1 \leftarrow X_0 \times Z_0 \mod f(x)$
5. $Z_0 \leftarrow Z_0 + X_0$
6. $Z_0 \leftarrow Z_0^2 \mod f(x)$
7. $X_0 \leftarrow Z_0 \times t_0 \mod f(x)$
8. $X_0 \leftarrow X_0 + t_1$
9. return $X_0, Z_0$

Furthermore, we have the final transformation Algorithm 6 to extract the $y$-coordinate. As this method is only called once after the point multiplication, it is much less important than $\texttt{Madd}$ or
\texttt{Mdouble} which are both invoked inside the key evaluation loop.

\textbf{Algorithm 6: Mxy Algorithm}

\begin{itemize}
  \item \textbf{Input:} $x$-coordinates $X/Z$ of the points $P(x,y)$, $X_0$, $X_1$, $Z_1$
  \item \textbf{Output:} Affine coordinates $(x',y')$ for the point $P_0$
\end{itemize}

1 if $Z_0 = 0$ then return $Q \leftarrow (0,0)$;
2 if $Z_1 = 0$ then return $Q \leftarrow (x,x+y)$;
3 $t_0 \leftarrow x$;
4 $t_1 \leftarrow y$;
5 $t_2 \leftarrow Z_0 \times Z_1 \mod f(x)$;
6 $Z_0 \leftarrow Z_0 \times t_0 \mod f(x)$;
7 $Z_0 \leftarrow Z_0 + X_0$;
8 $Z_1 \leftarrow Z_1 \times t_0 \mod f(x)$;
9 $X_0 \leftarrow Z_1 \times X_0 \mod f(x)$;
10 $Z_1 \leftarrow Z_1 \times Z_0 \mod f(x)$;
11 $t_3 \leftarrow t_0^2 \mod f(x)$;
12 $t_3 \leftarrow t_3 + t_1$;
13 $t_3 \leftarrow t_3 \times t_2 \mod f(x)$;
14 $t_3 \leftarrow t_3 + Z_1$;
15 $t_2 \leftarrow t_2 \times t_0 \mod f(x)$;
16 $t_2 \leftarrow (t_2)^{-1}$;
17 $t_3 \leftarrow t_3 \times t_2 \mod f(x)$;
18 $X_1 \leftarrow X_0 \times t_2 \mod f(x)$;
19 $Z_1 \leftarrow X_1 + t_0$;
20 $Z_1 \leftarrow Z_1 \times t_3 \mod f(x)$;
21 $Z_1 \leftarrow Z_1 + t_1$;
22 return $(x',y') \leftarrow (X_0, Z_0)$

With the results from above, the projective version of the LD/Montgomery multiplication is given in algorithm 7.

\textbf{Algorithm 7: Montgomery’s Method in projective coordinates}

\begin{itemize}
  \item \textbf{Input:} Point $P = (x,y)$, Integer $k = (k_{l-1}, k_{l-2}, ..., k_0)_2$, $k \geq 0$
  \item \textbf{Output:} Point $Q \leftarrow kP$
\end{itemize}

1 if $k = 0$ or $X = 0$ then return $Q \leftarrow (0,0)$;
2 $X_0 \leftarrow x$;
3 $Z_0 \leftarrow 1$;
4 $Z_1 \leftarrow x^2$;
5 $X_1 \leftarrow x^4 + b$;
6 for $i \leftarrow (l-2)$ to 0 do
7     if $k_i = 0$ then
8         Madd $(X_0,Z_0,X_1,Z_1)$;
9         Mdouble $(X_1,Z_1)$;
10     else
11         Madd $(X_1,Z_1,X_0,Z_0)$;
12         Mdouble $(X_0,Z_0)$;
13     end
14 end
15 return $Q \leftarrow Mxy (X_0, Z_0, X_1, Z_1)$
Comparison  The following table shows the complexity of both affine and projective versions. The first fact to mention is that because of its linearity, the costs for addition and squaring are considered to be small. As one can see, the choice of the version is highly dependent on the cost of the field inversion operation rather than the costs for field multiplication. If the inversion is relatively cheaper than multiplication, the affine version will be the better choice, whereas if the multiplication is much faster, the projective version will gain from the low number of inversions.

<table>
<thead>
<tr>
<th>Field operation</th>
<th>Affine</th>
<th>Projective</th>
</tr>
</thead>
<tbody>
<tr>
<td>#ADD</td>
<td>$4\lfloor \log_2 k \rfloor + 6$</td>
<td>$3\lfloor \log_2 k \rfloor + 7$</td>
</tr>
<tr>
<td>#SQR</td>
<td>$2\lfloor \log_2 k \rfloor + 2$</td>
<td>$5\lfloor \log_2 k \rfloor + 3$</td>
</tr>
<tr>
<td>#MUL</td>
<td>$2\lfloor \log_2 k \rfloor + 4$</td>
<td>$6\lfloor \log_2 k \rfloor + 10$</td>
</tr>
<tr>
<td>#INV</td>
<td>$2\lfloor \log_2 k \rfloor + 1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

Table 2.1.: Comparison of affine and projective versions

The costs for field inversion relates to field element representation (see 2.1.3) and to the choice of the inversion method (2.1.2). The multiplication cost is strongly related to the multipliers available on the target platform. In this work, we are targeting hardware with a random carryless multiplier and therefore a fast multiplication. We will see that with the use of the Itoh-Tsujii algorithm, the field inversion algorithm is going to perform at least nine field multiplications for the target curves. Hence, the projective version will obviously give us much better performance than the affine version.
2.3. Cryptography with Elliptic Curves

This section explains the discrete logarithm problem for elliptic curves and glances at the main cryptographic schemes using elliptic curves.

2.3.1. Elliptic Curve Discrete Logarithm Problem (ECDLP)

The *Discrete Logarithm Problem* (DLP) for an elliptic curve provides the basis for the elliptic curve cryptosystem. It is based on the assumption that calculating the inverse of a point multiplication is a hard problem.

**Definition 12: Discrete Logarithm Problem for EC**

Let \( E(K) \) be an elliptic curve over the finite field \( K \) and \( P, Q \) be points on this curve. Let \( Q = kP \) further be a point multiplication for an unknown \( k \in [1, n - 1] \) where \( n = \frac{|E(K)|}{h} \), as defined in 2.2.1. With the knowledge of the points \( P \) and \( Q \), finding integer \( k \) is then called the *discrete logarithm* over \( E(K) \).

Security of Elliptic Curve Cryptosystems

The security of an elliptic curve cryptosystem is mainly bounded by the expense of calculating the discrete logarithm as defined above. For the interested reader, surveys of generic attacks on ECDLP can be found in chapter 19 of [21]. In general, we can say that the most powerful generic attacks (e.g., Pohlig-Hellmann, Pollard Rho) decrease the expense of calculating the discrete logarithm in \( GF(q) \) to \( O(\sqrt{q}) \). Other, more powerful but non-generic attacks as the Index calculus method have been found and studied ([21], chapter 20).

In the following, we give a brief overview about the security parameter (key bit length) of elliptic curves in comparison to the well-known public key cryptosystem RSA [58]. As reference parameter, we use the numbers for an exhaustive key search (here called *Bit Strength*), as it is believed to apply for symmetric ciphers such as AES [54]. As of today, it is also believed that an exhaustive key search for keys with a bit size greater than 80 bit are infeasible. The numbers for the Elliptic Curve Cryptosystem (ECC) apply for elliptic curves over \( GF(2^m) \) and can be found in [56].

<table>
<thead>
<tr>
<th>Bit Strength</th>
<th>RSA</th>
<th>ECC</th>
</tr>
</thead>
<tbody>
<tr>
<td>80</td>
<td>1024</td>
<td>163</td>
</tr>
<tr>
<td>96</td>
<td>1536</td>
<td>193</td>
</tr>
<tr>
<td>112</td>
<td>2240</td>
<td>233</td>
</tr>
<tr>
<td>115</td>
<td>2304</td>
<td>239</td>
</tr>
<tr>
<td>128</td>
<td>3456</td>
<td>283</td>
</tr>
<tr>
<td>192</td>
<td>7680</td>
<td>409</td>
</tr>
<tr>
<td>256</td>
<td>15360</td>
<td>571</td>
</tr>
</tbody>
</table>

Table 2.2.: Security parameter comparison for EC over \( GF(2^m) \)
2.3.2. Elliptic Curve Diffie-Hellmann (ECDH)

The Elliptic Curve Diffie-Hellmann protocol is a variant of the Diffie-Hellman key agreement protocol using the discrete logarithm for elliptic curves (2.3.1). Usually, it is used to establish a mutual secret over an insecure channel, e.g., to derive a session-key for symmetric encryption for further communication. The next figure shows how this establishment works for two parties, called Alice and Bob.

\[
Q_A = d_A \cdot G \\
Q_B = d_B \cdot G \\
d_AQ_B = (X_k, Y_k) \\
d_BQ_A = (X_k, Y_k) \\
K_{AB} = \text{shared secret}
\]

Figure 2.6.: Simple ECDH Key Exchange

First, Alice and Bob need to agree upon the domain parameter \((m, f(x), a, b, G, n, h)\) as defined before with \(G\) a generator point of \(G_E\), the large subgroup of \(E(K)\). Additionally, both parties need to possess their own key pair, suitable for public key cryptography in the selected domain. In particular, this key pair consists of a public key \(Q\), which is a point on the curve, and a secret (private) key \(d\) with \(Q = dG\).

Let \((d_A, Q_A)\) be the key pair of Alice and \((d_B, Q_B)\) Bob’s key pair. The two public keys need to be shared with the other party and are usually exposed to anyone. If the public keys are known, both parties calculate the secret \((X_k, Y_k)\) with a point multiplication of their secret key with the public key point of the opposite party. Therefore, Alice computes \(d_AQ_B = (X_k, Y_k)\) and Bob \(d_BQ_A = (X_k, Y_k)\). Since \(d_AQ_B = d_Ad_BG = (X_k, Y_k) = d_Bd_AG = d_BQ_A\), Alice and Bob share the same point \((X_k, Y_k)\). Now both parties posses a secret, which can be used to derive another shared secret \(K_{AB}\). In fact, most standardized protocols use the \(x_k\) coordinate to derive this key with the use of cryptographic hash functions. Since no participant exposes more than the public key, no other party can calculate the shared secret without solving the discrete logarithm problem for elliptic curves.

The public keys can either be static or ephemeral. In order to avoid man-in-the-middle attacks, at least one of the public keys must be static and trusted. Please note that this is only the very basic protocol, further security goals such as forward secrecy or entity authentication require additional mechanisms and lead to more sophisticated protocols.
2.3.3. Elliptic Curve Digital Signature Algorithm (ECDSA)

The **Elliptic Curve Digital Signature Algorithm** is a variant of the **Digital Signature Algorithm (DSA)**[49], which in turn can be seen as a variant of the ElGamal signature scheme[26] using elliptic curve cryptography.

ECDSA is used to ensure data integrity, data origin/entity authentication and non-repudiation. Data integrity assures that the data has not been altered by an unknown party, whereas data origin and entity authentication assert that the data source or communication partner is as claimed. Non-repudiation is the assurance that an entity cannot deny actions or commitments. We refer the reader to the work about ECDSA[36] by D.Johnson and A.Menezes for more information about this scheme.

**ECDSA Signature generation**

To create a signature, we first need a key pair \((d_A, Q_A)\) for a domain \((m, f(x), a, b, G, n, h)\) where \(Q_A = d_A G\) for a \(d_A \in \mathbb{Z}_{1,n-1}\) and a cryptographic hash function \(\text{hash}\) such as of the SHA-family[50].

The signature generation process follows in algorithm 8.

**Algorithm 8: ECDSA Signature generation**

**Input:** Message \(m\), key pair \((d_A, Q_A)\)

**Output:** Signature \((r, s)\)

1. \(e \leftarrow \text{hash}(m)\);
2. \(z \leftarrow \lceil \log_2 n \rceil\) leftmost bits of \(e\);
3. \(k \leftarrow k \in \mathbb{Z}_{1,n-1}\);
4. \((x, y) \leftarrow k \cdot G\);
5. \(r \leftarrow x \mod n\);
6. if \(r = 0\) then goto step 3;
7. \(s \leftarrow k^{-1}(z + r \cdot d_A) \mod n\);
8. if \(s = 0\) then goto step 3;
9. return \((r, s)\)

**ECDSA Signature verification**

In order to verify a signature, one must know the public key point \(Q_A\) from the communication partner. To ensure \(Q_A\) is a valid in the agreed domain (and thus avoiding special attacks), one should ensure that \(Q_A = (x, y)\) has valid coordinates, \(Q_A \neq \emptyset\) and that \(Q_A\) lies on the curve, \(n \cdot Q_A\) equals \(\emptyset\) and that \(Q_A \notin H_E\). If \(Q_A\) is a valid point, one can verify the signature with algorithm 9.

**Algorithm 9: ECDSA Signature verification**

**Input:** Message \(m\), Signature \((r, s)\), public key point \(Q_A\)

**Output:** \(\text{ret} \in \{\text{valid}, \text{invalid}\}\)

1. if \(r, s \notin [1,n-1]\) then return \text{invalid};
2. \(e \leftarrow \text{hash}(m)\);
3. \(z \leftarrow \lceil \log_2 n \rceil\) leftmost bits of \(e\);
4. \(w \leftarrow s^{-1}\);
5. \(u_0 \leftarrow zw \mod n\);
6. \(u_1 \leftarrow rw \mod n\);
7. \((x, y) \leftarrow u_0 \cdot G + u_1 \cdot Q_A\);
8. if \(r = x \mod n\) then \(\text{ret} = \text{valid}\);
9. else \(\text{ret} = \text{invalid}\);
10. return \(\text{ret}\)
3. Side Channel Protection and Algorithm Improvements

In this chapter we present our ideas on how to improve the elliptic curve point multiplication in terms of performance and security. Therefore, we show a reasonable attacker model, sum up possible side channel threats and countermeasures and suggest a protection strategy. Additionally, we provide improvements on the algorithmic level for techniques presented in the previous chapter and show the impact of our improvements to the EC point multiplication.

3.1. Side Channel Protection

In this section, we discuss possible side channel attacks for the implementation of the elliptic curve point multiplication. As we have seen in the past, it is crucial for software implementations of cryptographic primitives to take side channel attacks into account. This analysis applies to the Montgomery/LD point multiplication only, not to the entire EC cryptosystem.

3.1.1. Introduction to Side Channel Attacks

Nowadays, the actual implementation of a cryptographic primitive gets more and more into the focus of attackers. Besides the view of cryptographic primitives as mathematical objects and the theoretic approach to break it (classical cryptanalysis), one can also approach it as an application running on a device having different channels leaking secret information. This approach is called side channel analysis and became very popular in the last decades due to its outstanding, practical results in breaking ciphers and other primitives.

Since side channel attacks focus on the actual design in hard- and software, the amount of possible attacks is tremendous. While an adversary is able to pick one single attack, the implementer must consider every possible attack and apply proper countermeasures. Unfortunately, it turns out that certain countermeasures themselves cause further weaknesses, making the challenge even greater, thus requiring highly complex solutions. The amount of countermeasures applied to a product affects not only the application’s performance, but also the budget. To achieve the best side channel resistance, it is highly recommended to apply as many countermeasures and thus being as paranoid as possible.

3.1.2. Side Channel Attacks to ECC

As for many public key cryptographic schemes, lots of different side channel attacks to elliptic curve cryptography have been found in the last years. We want to briefly introduce techniques that can be used to gain secret information through side channels in elliptic curve cryptosystems on server architectures. For more detailed information about side channel attacks on ECC please refer to the excellent surveys [27] by Fan and Verbauwhede, [11] by R.M. Avanzi, the Guide to Elliptic Curve Cryptography [32], the Side Channel Cryptanalysis Lounge[48] and the online database for side channel attacks [55].

In [17], Brumley and Tuveri showed that remote timing attacks are still practical and applicable to (here: binary) ECC implementations. Exploiting not even a real implementation error but an inter component design flaw, it also shows how fragile and prone a cryptographic implementation can be to side channel attacks.
Attacker Model

In this subsection we want to define an attacker model in order to determine the attackers strength and abilities and thus the probability of certain side channel attacks.

As mentioned before, we aim to provide a fast and protected point multiplication implementation, running on a server architecture such as a webserver. Therefore, we assume our implementation to run on a server with no physical access to an adversary, whilst the attacker is able to run a user mode process. This assumption makes sense, since the attackers strength should be as great as possible without having access to neither the application’s content nor any part on the system level which would make an attack quite trivial. Having root rights, the attacker could easily access application memory and CPU and thus examine the secret keys, modify code and compromise the system in any possible way. Further, we assume that the adversary is able to perform SSL/TLS handshakes in order to start a secured session. In the following, we will list a summary of the attackers abilities.

The adversary...

▶ is able to perform an unlimited amount of SSL/TLS handshakes,
▶ runs his own spy process in user mode on the server,
▶ has neither root rights nor physical access to the server,
▶ has limited computational power (e.g., for key recovering).

Attacks and Countermeasures

We want to list the basic ideas for attacks possible for an adversary with the capabilities stated in the previous section. In particular, this means we are focusing on passive attacks only. This overview shows the main threats, however, other techniques may be found and successfully applied to the ECC implementation. Furthermore, our analysis is constrained to the elliptic curve LD point multiplication implementation only - countermeasures for other possible attacks on the higher level primitives such as the ECDH or ECDSA protocols are not included. Since the point multiplication is the most crucial part for the ECC, the attacks and countermeasures affect the security of the whole cryptosystem though.

Timing Attacks

exploit the occurrence of variant execution times for different inputs. This kind of attack can be mounted whenever any correlation between the key and the execution time can be found. Thus, to avoid timing attacks in the LD/Montgomery point multiplication one must ensure that

1. all inputs \((k,P)\) to the point multiplication function are valid, meaning

   ◦ Integer \(k\) is in the interval \([1,n]\) with \(n = |\langle P \rangle|\)
   ◦ \(k\) has constant bit length (for the same group)
   ◦ \(P\) is a valid point on the curve \((P \in G_E \Rightarrow |\langle P \rangle| = n = |G_E|)\)

2. the point multiplication itself is executed in constant time (and therefore all sub-functions).

Actually, the input validation of the first point should be done before calling the multiplication function. Anyhow, it turns out to be useful to fix the bit length of the scalar \(k\) by adding multiples of the point order \(n\) to \(k\), to assert a constant running time for any valid input.

Cache Timing Attacks

are attacks based on the cache architecture of the target hardware. Since modern CPU’s are using a cache pipeline to reduce slow memory accesses, a spy process with the ability to measure cache latencies may be able to find out whether the targeted application (here called crypto process) uses certain memory or function addresses. This technique can be applied to both data and instruction caches where the spy process inserts dummy instructions or data. If spy and the monitored process crypto are running on the same (physical) core, the crypto process changes the cache state
and therefore “evicts” spy’s dummies. In turn, spy measures the time to execute the chased function or access a memory location.

Cache Attacks against several implementations of cryptographic primitives have been presented in the past, e.g., AES [53], RSA [19], and ECC [16]. For further information about cache attacks and countermeasures, we refer to [15]. Here we want to sum up the most common countermeasures to demonstrate how one can protect the point multiplication at all and point out the (im-)practicalness of the particular methods.

**Countermeasures against Cache Attacks**

- **Disabled caches** (fully or partially)
  On x86 architectures, the programmer can disable caches fully or partially. A simple solution which obviously has a devastating performance impact.

- **flushing/Flooding caches** (fully or partially)
  Flushing the cache before each context switch may sound a good idea, however, it is hardly practicable on the target architectures.

- **Avoiding memory accesses**
  Another idea is to avoid memory accesses in general, which obviously protects an application from being spied on but is most often impractical. More promising is to avoid key dependent memory accesses such as table lookups.

- **Oblivious access pattern**
  Rather than avoiding memory access, one can make sure that the pattern of accessing is oblivious so that all memory entries are read in a fixed order.

**Power Consumption/EM Analysis Attacks** exploit the power consumption of the device during a key dependent operation and examine the key with correlation techniques. The most common attacks are the Simple Power Attack (SPA), Differential Power Attack (DPA) and Comparative Side Channel Attacks. Additionally, further sophisticated attacks, such as the Zero-Point Attack (ZPA) [5] or the Refined Power Attack (RPA) [30], use special properties of the EC cryptosystem in order to mount the attack and are immune against most SPA and DPA countermeasures.

Referring to our attacker model, most attacks of this kind seem to be out of our attackers capabilities due to the lack of sufficient methods to gain the required power data. Although simple attacks such as SPA might be applicable - to get useful power traces for a DPA with neither physical access nor root rights on the target machine is a quite challenging task. However, we will briefly list countermeasures for the presented attacks. Please see [27] for a more detailed discussion about this topic.

**Countermeasures against Power Consumption Attacks**

- **SPA**:
  The simplest solution to avoid simple power attacks is to calculate the point multiplication $kP$ with a fixed pattern of operations, independent from a specific bit $k_i$ of the scalar $k$ (as the LD/Montgomery point multiplication does).

- **DPA**:
  - **Scalar randomization** [22] [51] [28] [6]
    Blinding the scalar $k$ by adding a multiple of the group order $n$, such as $k' = k + r \cdot n$ for a random number $r$.
  - **Base point blinding** [22] [6]
    The blinding of the base point $P$ by performing $kP = k(P + R) - (kR)$ with securely stored and updated $(R, kR)$ per iteration.
Another similar side channel is the electromagnetic emanation (EM) from CMOS devices. EM signals broadcast via both radiation and conduction, and can be measured without physical contact. Due to the similarity to the power consumption side channel, many techniques as SPA and DPA can be adopted to build EM attacks. In some cases, even if power consumption attacks like DPA failed, EM attacks would still be possible.

### 3.1.3. Protection Strategy

Finally, we want to describe the protection strategy for the implementation and discuss possible vulnerabilities to side channel attacks. As we have seen, the large variety of different attacks makes it hard to protect an implementation against all side channel threats. Through defining the attackers model though, we could already point out that the probability for certain threats is low. In the following, we will show the mechanism implemented for resistance against the main threat for server implementations, the timing attacks.

**Constant key length** is a first step to assert a constant running time of the point multiplication \( kP \). Although, the input validation of the scalar \( k \) and the point \( P \) should be performed prior to the point multiplication, we provide a bit length fix for all valid \( k \in [1, n] \) with \( n \) order of the large subgroup \( G_E \) of the elliptic curve \( E \).

At first, we always add \( n \) to \( k \), so that \( k' = k + n \) and \( kP = k'P \). The bit length of \( k' \) is now either

\[
\lfloor \log_2(n) \rfloor \leq \log_2(k') \leq \lfloor \log_2(n) \rfloor + 1
\]  

(3.1)

since the result of an addition of two integers \( n \) and \( k \leq n \) is not longer than \( \lfloor \log_2(n) \rfloor + 1 \) bits.

The group order \( n \) of all target NIST/SECG curves over \( GF(2^m) \) is either of the form

1. \( l \approx \frac{n}{2} \) leading 1’s with the most significant bit at position \( m - 1 \) or
2. most significant bit is set at position \( m \), followed by \( l \approx \frac{n}{2} \) consecutive 0’s.

Thus, one addition with the group order does not guarantee a fixed length of \( k \). In the first case, \( k' = k + n \) only has a bit length \( \lfloor \log_2(n) \rfloor \) if the bit length of \( k \) is smaller than \( m - l - 1 \) (and therefore has no carry bit at the top), in the second case \( k' = k + n \) has a bit length \( \lfloor \log_2(n) \rfloor + 1 \) if the most significant bit is at position \( m \) or \( k' \)'s bits in range \( [m - 1, m - l] \) are 1’s. This fixes the length of \( k' \) with a high probability on either \( \lfloor \log_2(n) \rfloor \) or \( \lfloor \log_2(n) \rfloor + 1 \).

To make the bit length fixed for all cases, one can repeat the addition of \( n \), if and only if the bit at position \( \lfloor \log_2(n) \rfloor + 1 \) is NOT set. To ensure that the implementation does not include any branches, one should create a mask in a similar way as suggested in Algorithm 10 and then add the masked value, which is either the order \( n \) or an equivalent sized number with zero value. Hence, the bit length of \( k' \) is always \( \lfloor \log_2(n) \rfloor \). Please take into account that \( k \) needs to be a valid input with \( k \in [1, n] \) in order to assert the fixed bit length, which is expected to be evaluated outside the point multiplication.

Additionally, this countermeasure adds vulnerability to Carry-based attacks, attacking not the point multiplication but this particular countermeasure itself [28], considering our attacker model, however, this is no reasonable threat.

**Constant memory access pattern** and the **elimination of key dependent branches** are the main countermeasure against cache based side channel attacks in our implementation. Since cache attacks are most likely with our attacker model, we want to assure not only a fixed memory access pattern but also avoid all key dependent branches. Therefore, we suggest the following technique to veil the data processing.
Please see the proposed data veiling method below. Depending on $k_i$, a mask is created and then successively applied with the AND and NAND operation to the two possible values, satisfying the rules in the table to the left. Therefore, at first a masking word $t_0$ is created with either all bits set to zero, or all bits set to one, depending on the value of $k_i$. This word is then used to create the mask for field elements. Please note that this transition is self-inverse and needs to be applied again after the execution of $\text{Madd}$ and $\text{Mdouble}$.

Let $tx_1, tx_2, tz_1$ and $tz_2$ be temporary variables with a fixed size for a $GF(2^m)$ and $k_i$ the current bit a position $i \in [0, [\log_2(k)]]$. Table 3.1 shows the transition of the coordinates involved in the double/add process inside the LD/Montgomery point multiplication’s ($kP$) key evaluation loop.

<table>
<thead>
<tr>
<th></th>
<th>$k_i = 0$</th>
<th>$k_i = 1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tx_1$</td>
<td>$x_2$</td>
<td>$x_1$</td>
</tr>
<tr>
<td>$tx_2$</td>
<td>$x_1$</td>
<td>$x_2$</td>
</tr>
<tr>
<td>$tz_1$</td>
<td>$z_2$</td>
<td>$z_1$</td>
</tr>
<tr>
<td>$tz_2$</td>
<td>$z_1$</td>
<td>$z_2$</td>
</tr>
</tbody>
</table>

Table 3.1.: X/Z Transformation

Algorithm 10: Proposed data veiling method

**Input:** Keybit $k_i$, EC coordinates $x_1, x_2, z_1, z_2 \in GF(2^m)$

**Output:** Temporary coordinates $tx_1, tx_2, tz_1, tz_2 \in GF(2^m)$

1. $t_0 \leftarrow (0x00...0 - k_i)$;
   
2. for $(j = 0; j < \lceil \frac{m}{W} \rceil; j++)$ do $\text{mask}[j] \leftarrow t_0$;

3. $tx_1 \leftarrow (\text{mask} \land x_1) \oplus (\text{mask} \landbar x_2)$;

4. $tx_2 \leftarrow (\text{mask} \landbar x_1) \oplus (\text{mask} \land x_2)$;

5. $tz_1 \leftarrow (\text{mask} \land z_1) \oplus (\text{mask} \landbar z_2)$;

6. $tz_2 \leftarrow (\text{mask} \landbar z_1) \oplus (\text{mask} \land z_2)$;

7. return $tx_1, tx_2, tz_1, tz_2$

In summary, the proposed technique requires 4 field XORs, NANDs and ANDs, and, negligible in proportion, one 64 bit subtraction and one constant load each. Our results (see 5.10) show that this method has acceptable costs, comparable to reduction over $GF(2^m)$.

**Constant function calls** inside the key evaluation loop are naturally provided by the Montgomery/LD point multiplication. In addition, we must ensure that all functions called inside the loop are executed in constant time and thus are not depending on the data through branches.

**Constant time implementation for binary field arithmetic** is the necessary consequence from the previous point. If we want to ensure that the running time is constant, the binary field arithmetic must be totally data independent. This does not only mean that we cannot include any shortcuts for special values and data dependent branches at all, but it also biases the choice of algorithm.

**Discussion**

Due to the constant time implementation, the Montgomery/LD point multiplication is protected against timing and simple power analysis attacks. Since most practical attacks are based on timing, we can exclude most of them. Assuming our attacker model is valid, we can also exclude differential power and electromagnetic analyses since the attacker needs physical access or, in case of EM attacks, at least be able to place eavesdrop equipment into a few meters distance.
Therefore, the remaining threats are the cache timing attacks. The nature of the Montgomery/LD point multiplication algorithm is to have a key evaluation loop with a key dependent branch and calls of the functions $\texttt{Madd}$ and $\texttt{Mdouble}$ with a different parameter order. We proposed a method to eliminate the branch and veil the processed data by loading temporary variables in a fixed pattern but masking it in dependence on the current key bit.

Anyhow, we can of course not guarantee total resistance against side channel attacks. Furthermore, due to the branches inside $\texttt{Mxy}$, the running time for the point multiplication is not constant, if the result is invalid, the inverse $-P$ of a point $P$ or the point at infinity (and thus the scalar $k$ a multiple of $n$).
3.2. Improvements of the 2P Algorithm

In this section we will show how to reduce the number of squares, multiplications and reductions in the point multiplication process on the algorithmic level.

As we have seen on page 15, the computational cost of algorithm 7 is mainly determined by the functions $M\text{add}$ and $M\text{double}$. Originally, both functions are executed exactly $\lceil \log_2(k) \rceil$ times during a point multiplication $kP$. Hence, even small changes may lead to a significant performance improvement. In the following we propose an improved version for the point doubling.

3.2.1. Improved $M\text{double}$

By re-ordering the flow from algorithm 4, we first multiply $Z^2$ with the precomputation value $c$ and then add the result to $X^2$, square this again and obtain the final result $X \leftarrow (X^2 + c \cdot Z^2)^2 = X^4 + b \cdot Z^4$.

<table>
<thead>
<tr>
<th>Algorithm 11: Point Doubling algorithm for random curves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: $c \in GF(2^m)$ where $c^2 = b$, x-coordinate $X/Z$ for a point $P$</td>
</tr>
<tr>
<td><strong>Output</strong>: x-coordinate $X/Z$ for the point $2P$</td>
</tr>
<tr>
<td>$1 \ t \leftarrow c;$</td>
</tr>
<tr>
<td>$2 \ X \leftarrow X^2 \mod f(x);$</td>
</tr>
<tr>
<td>$3 \ Z \leftarrow Z^2 \mod f(x);$</td>
</tr>
<tr>
<td>$4 \ t \leftarrow Z \times t \mod f(x);$</td>
</tr>
<tr>
<td>$5 \ Z \leftarrow Z \times X \mod f(x);$</td>
</tr>
<tr>
<td>$6 \ X \leftarrow X + t;$</td>
</tr>
<tr>
<td>$7 \ X \leftarrow X^2 \mod f(x);$</td>
</tr>
<tr>
<td>$8 \ \text{return } X, Z$</td>
</tr>
</tbody>
</table>

This new version of the doubling algorithm saves a squaring and one reduction per execution. In the field $GF(2^{283})$, for example, it saves approximately about 283 field squarings and reductions. Furthermore, we know that $b = 1$ for Koblitz curves and thus $c = 2^{m-1} = 1$. Since we are going to multiply $Z$ by a value of one in step 4, we can save an additional multiplication and reduction here. Algorithm 12 shows the modified version for Koblitz curves.

<table>
<thead>
<tr>
<th>Algorithm 12: Point Doubling algorithm for Koblitz curves</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: x-coordinate $X/Z$ for a point $P$</td>
</tr>
<tr>
<td><strong>Output</strong>: x-coordinate $X/Z$ for the point $2P$</td>
</tr>
<tr>
<td>$1 \ X \leftarrow X^2 \mod f(x);$</td>
</tr>
<tr>
<td>$2 \ Z \leftarrow Z^2 \mod f(x);$</td>
</tr>
<tr>
<td>$3 \ t \leftarrow X + Z;$</td>
</tr>
<tr>
<td>$4 \ Z \leftarrow Z \times X \mod f(x);$</td>
</tr>
<tr>
<td>$5 \ X \leftarrow t^2 \mod f(x);$</td>
</tr>
<tr>
<td>$6 \ \text{return } X, Z$</td>
</tr>
</tbody>
</table>
3.2.2. Improved Madd

The Madd function can also be improved by applying the so-called lazy reduction technique[62]. Here, the basic idea is to save a reduction by adding the two un-reduced results of the multiplications in steps 4 and 8 and reduce the result afterwards.

Depending on the field, the costs for a double sized addition is about factor 2-3 smaller than a reduction, so we can save a few cycles per round here. In [13] we show the algorithm for the improved Madd function.

Algorithm 13: Algorithm for Point Addition

\begin{algorithm}
\begin{verbatim}
Input:  x-coordinates $X/Z$ of the point $P(x, y), P_0(X_0, Z_0), P_1(X_1, Z_1)$
Output: x-coordinate $X_1/Z_1$ for the point $P_0 + P_1$
1 $t_0 \leftarrow X_1$;
2 $X_0 \leftarrow X_0 \times Z_0 \mod f(x)$;
3 $Z_0 \leftarrow Z_0 \times t_0 \mod f(x)$;
4 $t_4 \leftarrow X_0 \times Z_0$ ;  // mult. w/o reduction, result double sized
5 $Z_0 \leftarrow Z_0 + X_0$;
6 $Z_0 \leftarrow Z_0^2 \mod f(x)$;
7 $t_0 \leftarrow x$;
8 $t_2 \leftarrow Z_0 \times t_0$ ;  // mult. w/o reduction, result double sized
9 $t_2 \leftarrow t_2 + t_1$ ;  // double sized addition
10 $X_0 \leftarrow t_2 \mod f(x)$ ;
11 return $X_0, Z_0$
\end{verbatim}
\end{algorithm}

3.2.3. Complexity Results of Improved Method

In the next table we show the new numbers for Mdouble and Madd, listing the arithmetic functions ordered from the slowest to the fastest field operation.

\begin{table}[h]
\centering
\begin{tabular}{c||c|c|c|c|c}
\hline
Field operation & Mdouble(K) & Mdouble(B) & Mdouble(Old) & Madd(Improved) & Madd(Old) \\
\hline
#MUL & $[\log_2(k)]$ & $2[\log_2(k)]$ & $2[\log_2(k)]$ & $4[\log_2(k)]$ & $4[\log_2(k)]$ \\
#RED & $4[\log_2(k)]$ & $5[\log_2(k)]$ & $6[\log_2(k)]$ & $4[\log_2(k)]$ & $5[\log_2(k)]$ \\
#SQR & $3[\log_2(k)]$ & $3[\log_2(k)]$ & $4[\log_2(k)]$ & $[\log_2(k)]$ & $[\log_2(k)]$ \\
#ADD & $[\log_2(k)]$ & $[\log_2(k)]$ & $[\log_2(k)]$ & $3[\log_2(k)]$ & $2[\log_2(k)]$ \\
\hline
\end{tabular}
\caption{Computational costs for the improved versions of Mdouble and Madd}
\end{table}

The results from above sum up to the following table, which gives new total numbers for the LD/-Montgomery point multiplication.

\begin{table}[h]
\centering
\begin{tabular}{c||c|c|c}
\hline
Field operation & MONT(Koblitz) & MONT(Random) & MONT(Old) \\
\hline
#INV & 1 & 1 & 1 \\
#MUL & $5[\log_2(k)]+10$ & $6[\log_2(k)]+10$ & $6[\log_2(k)]+10$ \\
#RED & $8[\log_2(k)]+13$ & $9[\log_2(k)]+13$ & $11[\log_2(k)]+13$ \\
#SQR & $4[\log_2(k)]+3$ & $4[\log_2(k)]+3$ & $5[\log_2(k)]+3$ \\
#ADD & $4[\log_2(k)]+7$ & $4[\log_2(k)]+7$ & $3[\log_2(k)]+7$ \\
\hline
\end{tabular}
\caption{New numbers for the LD/Montgomery point multiplication without SC countermeasures}
\end{table}
Nevertheless, we need to consider the effect of the side channel countermeasures. Therefore, we count the field ANDs and NANDs as field addition as their costs are the same in the implementation. Table 3.4 shows the new total numbers for the LD/Montgomery point multiplication. Please note that the running time now depends on the length of the curve order $n$ instead of the scalar $k$.

<table>
<thead>
<tr>
<th>Field operation</th>
<th>MONT (Koblitz)</th>
<th>MONT (Random)</th>
<th>MONT (Old)</th>
</tr>
</thead>
<tbody>
<tr>
<td>#INV</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>#MUL</td>
<td>$5 \lceil \log_2(n) \rceil + 10$</td>
<td>$6 \lceil \log_2(n) \rceil + 10$</td>
<td>$6 \lceil \log_2(n) \rceil + 10$</td>
</tr>
<tr>
<td>#RED</td>
<td>$8 \lceil \log_2(n) \rceil + 13$</td>
<td>$9 \lceil \log_2(n) \rceil + 13$</td>
<td>$11 \lceil \log_2(n) \rceil + 13$</td>
</tr>
<tr>
<td>#SQR</td>
<td>$4 \lceil \log_2(n) \rceil + 3$</td>
<td>$4 \lceil \log_2(n) \rceil + 3$</td>
<td>$5 \lceil \log_2(n) \rceil + 3$</td>
</tr>
<tr>
<td>#ADD</td>
<td>$16 \lceil \log_2(n) \rceil + 7$</td>
<td>$16 \lceil \log_2(n) \rceil + 7$</td>
<td>$3 \lceil \log_2(n) \rceil + 7$</td>
</tr>
</tbody>
</table>

Table 3.4: New numbers for the LD/Montgomery point multiplication with SC countermeasures

Further discussions of these numbers can be found in chapter 5.
4. Implementation

In this chapter we develop a library for the binary field arithmetic for Koblitz and Random curves suggested by NIST and SECG with 163, 193, 233, 239, 283, 409 and 571 bit. Additionally, we will show how to integrate this implementation into the OpenSSL library with the proposed improvements from the previous chapters and discuss benefits, challenges, and limitations.

4.1. The Intel Vector Extensions and PCLMUL

In this subsections we introduce the Intel Vector Extensions and the carry-less multiplication instruction and describe its features and opportunities towards an efficient and fast EC point multiplication.

4.1.1. Architectures and History

The Streaming SIMD Extensions (SSE) instruction set was designed and introduced by Intel in 1999. Subsequently expanded up to SSE4, it is mainly used to speed up digital signal- and graphics processing. It uses 128 bit XMM registers (originally 8, now 16 on x86-64 architectures) which can, for SSE2 or greater, be used to hold
- two 64 bit floating point numbers (double precision),
- two 64 bit integers,
- four 32 bit integers,
- eight 16 bit integers or
- sixteen 8 bit integers.

A huge variety of instructions for data management, comparison, shuffling and packing, logical and arithmetic operations and more are available. For a complete list please refer to the corresponding manuals for the specific hardware and version.

The AVX (Advanced Vector Extension) Instruction Set was proposed by Intel in March 2008 and introduced in 2011 with the Sandy Bridge (Intel) and Bulldozer (AMD) architectures. It introduces a larger register set of 256 bit (named YMM instead of XMM) and a non-destructive three-operand form such that $c = a + b$, and thus neither destroying $b$ nor $c$. Unfortunately, the first version of AVX, available on the current architectures at the time of writing, only supports floating point arithmetics. Accordingly, the integer extensions will be introduced with AVX2 on the new Intel Haswell architecture.

The $\text{pclmulqdq}$ processor instruction finally performs a carry-less multiplication of two 64 bit integers and was introduced together with the AES instruction set with the Intel Westmere architecture in early 2010.

4.1.2. The Arsenal

In this work we use the SSE4 instruction set and the $\text{pclmulqdq}$ instruction. In order to provide with small code size, but also efficiency for the target hardware, we decided to use C compiler intrinsics. In this section, we list the used intrinsics and the corresponding instructions in particular and explain their capabilities. Therefore, we separate the instructions into several different kinds and give the cycles for latency (L) and throughput (T). These numbers will be for Sandy Bridge only (CPUID: 06_2A), we
refer to the Intel manuals and the “Intrinsics Guide for Intel Advanced Vector Extensions 2” for other CPUs and additional information about the commands.

The first kind of instructions are those to load/store values from/into memory locations for 128 bit vectors. The intrinsics, which are using the CPU instruction movq, load or store a 64 bit value from one memory location to another, while zeroing the upper 64 bits. The instruction movqda moves 128 bit operands. The last intrinsic _mm_set_epi64x creates a sequence of instructions and sets two 64 bit integers. Since we will see the load and store operations on the algorithmic level, we will use the mnemonic ← for general load/store operations and ←_64 for SET64.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic</th>
<th>Instruction</th>
<th>Latency/Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD_64</td>
<td>_mm_loadl_epi64</td>
<td>movq</td>
<td>1/0.33</td>
</tr>
<tr>
<td>LOAD128</td>
<td>_mm_load_si128</td>
<td>movqda</td>
<td>1/0.33</td>
</tr>
<tr>
<td>STORE_64</td>
<td>_mm_storel_epi64</td>
<td>movq</td>
<td>1/0.33</td>
</tr>
<tr>
<td>STORE128</td>
<td>_mm_store_si128</td>
<td>movqda</td>
<td>1/0.33</td>
</tr>
<tr>
<td>SET64</td>
<td>_mm_set_epi64x</td>
<td>(several)</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Table 4.1.: Load, Store and Memory - Intrinsics

The second instruction group is used for the arithmetic/logical operations. We start with the most powerful instruction, the pclmulqdq, which performs a carryless multiplication of two quadwords and stores the result to a _m128i location. An additional 8 bit immediate allows to select from the two 64 bit operands of each 128 bit input. Compared to general multiplication, the multiplier is relatively slow on architectures before Haswell, which introduces a much faster version of this instruction.

Another powerful tool is the pshufb instruction, which allows one to implement any 4 bit → 8 bit extension function on a 128 bit range and is therefore optimal for the usage as lookup function for small tables. It takes two arguments a and b and performs inplace shuffles of bytes in a per control mask in b, taking the lowest 4 bit of each 8 bit value for addressing, while leaving b unchanged.

The operators XOR, AND, NAND and OR perform these very same logical operations on their two arguments and save the result to a new memory location. The psllq and psrlq instructions shift the two 64 bit integers of the argument vector to the left/right while shifting in zeros. The second argument determines the amount of shifted bits, which can be an immediate between 0 and 64. The pslldq and psrldq are only capable to shift the argument by multiples of 8 but allow to shift the complete 128 bit input.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic</th>
<th>Instruction</th>
<th>Mnemonic</th>
<th>L/T</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCLMUL</td>
<td>_mm_clmulepi64_si128</td>
<td>pclmulqdq</td>
<td>×</td>
<td>14/8</td>
</tr>
<tr>
<td>SHUFFLE</td>
<td>_mm_shuffle_epi8</td>
<td>pshufb</td>
<td>shuffle()</td>
<td>1/0.5</td>
</tr>
<tr>
<td>XOR</td>
<td>_mm_xor_si128</td>
<td>pxor</td>
<td>⊕</td>
<td>1/0.33</td>
</tr>
<tr>
<td>AND</td>
<td>_mm_and_si128</td>
<td>pand</td>
<td>∧</td>
<td>1/0.33</td>
</tr>
<tr>
<td>NAND</td>
<td>_mm_andnot_si128</td>
<td>pandn</td>
<td>¬</td>
<td>1/0.33</td>
</tr>
<tr>
<td>OR</td>
<td>_mm_or_si128</td>
<td>por</td>
<td>∨</td>
<td>1/0.33</td>
</tr>
<tr>
<td>SHL</td>
<td>_mm_slli_epi64</td>
<td>psllq</td>
<td>≪_64</td>
<td>1/1</td>
</tr>
<tr>
<td>SHR</td>
<td>_mm_srl_epi64</td>
<td>psrlq</td>
<td>≫_64</td>
<td>1/1</td>
</tr>
<tr>
<td>SHL128</td>
<td>_mm_slli_si128</td>
<td>pslldq</td>
<td>≪_128</td>
<td>1/0.5</td>
</tr>
<tr>
<td>SHR128</td>
<td>_mm_srl_si128</td>
<td>psrldq</td>
<td>≫_128</td>
<td>1/0.5</td>
</tr>
</tbody>
</table>

Table 4.2.: Arithmetic Intrinsics

The last group of instructions used in this work are the ones for memory alignment. These are separated from the load and store instructions because of their different usage in the arithmetics. One very powerful instruction is the palignt instruction, which enables the programmer to concatenate the two 128 bit arguments and shift this intermediate value with byte precision to the right. The _mm_move_epi64 intrinsic uses the movq instruction we have already introduced before to clear the upper 64
bits but takes a _m128i value instead of a memory pointer. The punpcklbw and punpckhbw packing instructions have two 128 bit arguments a and b and interleave either the lower or upper 8 bit values, respectively. The instructions punpcklqdq and punpckhqdq do the same for 64 bit values. The pandn performs the operation $a \land b \leftrightarrow \neg\not(a) \AND b$ for given input words a and b. The last instruction we want to mention here is the _mm_setzero_si128, which uses pxor in order to set any 128 bit value to zero.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic</th>
<th>Instruction</th>
<th>Mnemonic</th>
<th>L/T</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNR</td>
<td>_mm_alignr_epi8</td>
<td>palignr</td>
<td>$\gg_8$</td>
<td>1/0.5</td>
</tr>
<tr>
<td>MOVE64</td>
<td>_mm_move_epi64</td>
<td>movq</td>
<td>$\gg_{64}$</td>
<td>1/0.33</td>
</tr>
<tr>
<td>UNPACKLO8</td>
<td>_mm_unpacklo_epi8</td>
<td>punpcklbw</td>
<td>unpacklo8()</td>
<td>1/0.5</td>
</tr>
<tr>
<td>UNPACKHI8</td>
<td>_mm_unpackhi_epi8</td>
<td>punpckhbw</td>
<td>unpackhi8()</td>
<td>1/0.5</td>
</tr>
<tr>
<td>UNPACKLO64</td>
<td>_mm_unpacklo_epi64</td>
<td>punpcklqdq</td>
<td>unpacklo64()</td>
<td>1/0.5</td>
</tr>
<tr>
<td>UNPACKHI64</td>
<td>_mm_unpackhi_epi64</td>
<td>punpckhqdq</td>
<td>unpackhi64()</td>
<td>1/0.5</td>
</tr>
<tr>
<td>ZERO</td>
<td>_mm_setzero_si128</td>
<td>pxor</td>
<td>ZERO</td>
<td>1/0.33</td>
</tr>
</tbody>
</table>

Table 4.3.: Memory Alignment Intrinsics
4.2. Binary Field Arithmetics

In this section we explain the techniques for the implementation of the binary field arithmetic.

4.2.1. Addition

The addition of two elements \( c(x) = a(x) \oplus b(x) \) is the cheapest field operation in \( GF(2^m) \), implemented as a simple exclusive OR with the XOR intrinsic of SSE. If AVX is enabled though, the \texttt{vxorpd} instruction can be used, which gives us better results for more than 3 quadwords. This instruction works on 256 bit floating point values, so that we need additional load/store instructions for the conversion to 256 bit floats. In the end, however, the non-destructive form and the double sized XOR operation pays off against the conversion overhead and the larger throughput (1 cycle instead of 0.33). In fact, the \texttt{vxorpd} is the only true benefit we get from AVX.

4.2.2. Squaring

As stated in Section 2.1.2, the squaring is a linear function in \( GF(2^m) \) and can be performed with small table lookups. This table can be stored in only one vector and is therefore accessed in a fixed pattern. Since the operation consists of the operands expansion by successively inserting zeros, it can even be computed in parallel. Therefore, we split the operand into two parts by applying a 4 bit mask. The resulting operands have the form

\[
a_{\text{High}}(x) = \sum_{i=0}^{m-1} a_i x^{i-4}, \quad \forall \quad i \mod 8 \leq 3
\]

\[
a_{\text{Low}}(x) = \sum_{i=0}^{m-1} a_i x^i, \quad \forall \quad i \mod 8 \geq 4
\]

This splitting can be performed with two AND maskings and one 4 bit right-shift. Now we apply the \texttt{pshufb} instruction twice, while taking \( a_{\text{High}} \) and \( a_{\text{Low}} \) as control mask inputs and thereby perform the table lookup. Finally, the result is combined by consecutive interleaving loads from both results.

![Figure 4.1.: Squaring technique in \( GF(2^m) \) with \texttt{pshufb}](image-url)
This method was originally proposed by Aranha et al. in [10]. Algorithm 14 shows generic squaring in \( GF(2^m) \) without the final reduction.

Algorithm 14: Fast Squaring in \( GF(2^m) \)

Input: \( a(x) = (A[t-1], ..., A[0]), \quad t = \lceil \frac{m}{128} \rceil \)

Output: \( z(x) = a^2(x) \)

1. \( \text{mask} \leftarrow 64 \cdot 0x0F0F0F0F0F0F0F0F \)
2. \( \text{sqrT} \leftarrow 64 \cdot 0x5554515045444140, 0x1514111005040100 \)
3. for \( i = 0; i < t; i++ \) do
   4. \( x_0 \leftarrow A[i] \land \text{mask} \)
   5. \( x_1 \leftarrow A[i] \gg 64 \)
   6. \( x_1 \leftarrow x_1 \land \text{mask} \)
   7. \( Z[2 \cdot i] \leftarrow \text{unpacklo}_8(x_0, x_1) \)
   8. \( Z[2 \cdot i + 1] \leftarrow \text{unpackhi}_8(x_0, x_1) \)
4. return \( z(x) = a^2(x) \)

Please note that we unrolled the loop in our implementation and save the very last \( \text{unpackhi} \) if \( \lceil \frac{m}{128} \rceil \) is odd.

4.2.3. Multiplication

The multiplication is usually an expensive basic operation in \( GF(2^m) \). The \texttt{pclmulqdq} instruction brings us to the fortunate position of having a 64 bit carry-less multiplier at hand. This work implements the multiplication with the Karatsuba-Ofman (from now on: Karatsuba) trick, which enables one to perform the operation in sub-quadratic time. In general, multiplication with Karatsuba is not the fastest technique known, however, in our application case it is the most suitable, as will be seen. In addition, the random number multiplier gives the benefit of low memory consumption, which has the advantage of saving memory and thus enhancing performance.

Multiplication Technique Comparison

As of today, a lot sub-quadratic multiplication schemes are known. The asymptotically fastest known technique is F"urer’s Algorithm, which however only works for enormously large numbers. The next option is the Schönhage-Strassen Algorithm with complexity \( O(n \log(n) \log(\log(n))) \), but it also starts to gain for numbers beyond \( 2^{225} \) and is not of interest here, since our largest multiplication case is 571 bit (this is \( \approx 2^{23} \)). In fact, the asymptotically best algorithm for this case is the Toom-k Algorithm, originally invented by Toom in 1963 and improved by Cook (therefore called Toom-Cook), which performs a multiplication in \( \Theta(e(k) n^e) \), with \( e = \frac{\log(2k-1)}{\log(k)} \) and \( e(k) \) a linear function. The Toom-3 variant for example runs in \( \Theta(n^{\log(3)/\log(5)}) = \Theta(n^{1.465}) \) compared to \( \Theta(n^{\log(2)/\log(3)}) = \Theta(n^{1.585}) \) for the simple (2x2) Karatsuba, which is actually a special case of Toom-2. Although the Toom-Cook Algorithm works in binary fields [12], it requires many linear operations (summed up in \( c(k) \)), resulting in a large constant factor.

The Karatsuba-Ofman Algorithm

The basic idea of the Karatsuba Algorithm is to split up both numbers into two parts, so that

\[
\begin{align*}
a(x) &= a_H(x) \cdot x^M + a_L(x) \\
b(x) &= b_H(x) \cdot x^M + b_L(x)
\end{align*}
\]
perform three intermediate results

\[ c_0(x) = a_L(x) \cdot b_L(x) \]
\[ c_1(x) = a_H(x) \cdot b_H(x) \]
\[ c_2(x) = (a_L(x) + a_H(x)) \cdot (b_L(x) + b_H(x)) \]

and conduct the multiplication as

\[ c(x) = a(x) \cdot b(x) = c_1(x) \cdot x^{2M} + [c_2(x) - c_0(x) - c_1(x)] \cdot x^M + c_0(x) = c_1(x) \cdot x^{2M} \oplus [c_2(x) \oplus c_0(x) \oplus c_1(x)] \cdot x^M \oplus c_0(x). \quad \text{(in } GF(2^m)) \]

This formula is correct, since

\[
\begin{align*}
a_L(x)b_H(x) + a_H(x)b_L(x) &= (a_L(x) + a_H(x)) \cdot (b_L(x) + b_H(x)) - a_L(x)b_L(x) - a_H(x)b_H(x) \\
 &= c_2(x) - c_0(x) - c_1(x) \quad \text{(in } GF(2^m))
\end{align*}
\]

This strategy can be applied recursively and generalized\[63\] but it always pays off to use customize formulas for different sizes, such as suggested by Montgomery in \[47\]. An additional benefit of Karatsuba is the constant memory consumption without table lookups, which makes it a good choice for our application, considering the need of a constant time implementation.

**Multiplication Strategy**

In this work we use both explicit and recursive versions of Karatsuba for different sizes. For a comparison of Karatsuba implementations with vector extensions, please see the work by Su and Fan\[59\]. We chose explicit forms for 2-, 3- and 5-term Karatsuba and recursive forms for 4-, 7- and 9-terms. This decision was made by balancing performance gain against code size. In the following table we will state the number of 64 bit multiplications and low level operations such as shift and xor for the corresponding Karatsuba variant.

<table>
<thead>
<tr>
<th>Karatsuba</th>
<th>CL-MULs</th>
<th>Low Level OP</th>
<th>Recursive Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>2×2</td>
<td>3</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td>3×3</td>
<td>6</td>
<td>19</td>
<td>-</td>
</tr>
<tr>
<td>4×4</td>
<td>9</td>
<td>38</td>
<td>2-2</td>
</tr>
<tr>
<td>5×5</td>
<td>13</td>
<td>42</td>
<td>4-4-3</td>
</tr>
<tr>
<td>7×7</td>
<td>24</td>
<td>90</td>
<td>5-5-4</td>
</tr>
<tr>
<td>9×9</td>
<td>35</td>
<td>154</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.4.: Karatsuba multiplication strategies for different sizes

**4.2.4. Reduction**

The reduction \[(2.1.2)\] is a very crucial part of the implementation. The two most important field operations, squaring and multiplication, both require a reduction with the field polynomial \(f(x)\). In fact, the reduction is even more expensive than squaring in our implementation and can be seen as a multiplication with the reduction polynomial. In this thesis we deal with the NIST polynomials and two polynomials from the SECG standard\[56\]. These polynomials are either trinomials (three components) or pentanomials (five components). In this section we will describe our strategy for each polynomial individually and discuss a special reduction scheme for pentanomials, using pclmulqdq.
As we have seen, a reduction is basically a multiplication of the upper half of the remainder with the reduction polynomial. Since a multiplication by one bit is a simple shift, this multiplication can also be viewed as a series of shifts and adds. This makes especially sense, if we remember that the amount of bits in \( r(x) \) is very small, here either 2 (for trinomials) or 4 (for pentanomials).

In order to avoid unnecessary duplication of work, one should execute the shift and add from the left to the right since the result of \( c'(x) = c_L(x) \oplus c_H(x) \cdot r(x) \) has a degree larger than \( m \) and thus needs another reduction step.

\[
c'(x) = c_L(x) \oplus c_H(x) \cdot r(x)
\]

As we have seen, a reduction is basically a multiplication of the upper half of the remainder with the reduction polynomial. Since a multiplication by one bit is a simple shift, this multiplication can also be viewed as a series of shifts and adds. This makes especially sense, if we remember that the amount of bits in \( r(x) \) is very small, here either 2 (for trinomials) or 4 (for pentanomials).

In order to avoid unnecessary duplication of work, one should execute the shift and add from the left to the right since the result of \( c'(x) = c_L(x) \oplus c_H(x) \cdot r(x) \) has a degree larger than \( m \) and thus needs another reduction step.

\[
c'(x) = c_L(x) \oplus c_H(x) \cdot r(x)
\]

Going from the left to the right though, this additional reduction can be saved. The next picture demonstrates how the reduction is consecutively done by adding the intermediate results to the remainder.
Figure 4.3.: Example step for Shift&Add Inplace Reduction in $GF(2^{163})$

As we see, the topmost bits responsible for the overlay are already added to the remainder now. If this step is applied again for the next chunk, we will save the additional reduction step.

**Fast Vector Shifting**  As explained above, we require many shift operations for the reduction. Unfortunately, we lack a 128 bit shift instruction with bit precision and thus perform a 128 bit shift with two 64 bit shifts, one to the left and one to the right and re-combine the relocated results afterwards. Anyhow, we can use the 128 bit shift instruction ($\text{pslldq}/\text{psrldq}$), if the difference of $m$ and the component to reduce is a multiple of 8. With the polynomials from 4.5, we can apply this trick only for the polynomial $f(x) = x^{163} + x^7 + x^6 + x^3 + 1$ where $163 - 3 = 160$. A similar trick is to exploit the case when the difference between two components in the polynomial is a multiple of 8, such as $x^{10}$ and $x^2$ in the reduction polynomial for $GF(2^{571})$. Assuming that we have already shifted the current word $w$ for component $x^2$, we can get the intermediate result for component $x^{10}$ by shifting $w$ 8 bits to the left.

**Trinomial Reduction**

The reduction for trinomials is very straightforward - we successively shift the top word by the components of $r(x)$ to the right and add the result to the remainder, while skipping a multiple of 64 and 128 bit with memory alignments. With the assumption that we are about to reduce a double quadword $w$ by component $x^k$, we shift $w$ exactly $(m - k \mod 64)$ bits to the right and add it $l$ times 128 bits to the right, where $l = \lfloor \frac{m-k}{128} \rfloor$. If $(m - k \mod 128) \geq 64$, however, we need to add the lower half of $w$ to word $l - 1$ and the upper part to $l$. In the following, we list the reduction schemes for the NIST/SECG trinomial polynomials with 193, 233 and 409 bits.
**Algorithm 15**: Reduction in $GF(2^{193})$

**Input**: $a(x) = \sum_{i=0}^{193} a_i x^i = (A[3], A[2], A[1], A[0])$

**Output**: $z(x) = a(x) \mod x^{193} + x^{15} + 1$

1. $t_0 \leftarrow (A[3] \ll \text{shift}) 14; t_1 \leftarrow (A[3] \gg \text{shift} 1); t_2 \leftarrow (A[3] \ll \text{shift} 63);$
2. $Z[1] \leftarrow [A[1]] \oplus t_2; t_4 \leftarrow t_0 \oplus t_1;$
3. $t_0 \leftarrow (A[2] \gg \text{shift} 50); t_1 \leftarrow (A[2] \ll \text{shift} 14); t_2 \leftarrow (A[2] \gg \text{shift} 1); t_3 \leftarrow (A[2] \ll \text{shift} 63);$
4. $Z[1] \leftarrow Z[1] \oplus t_0; Z[0] \leftarrow A[0] \oplus t_3;$
5. $t_0 \leftarrow t_1 \oplus t_2; t_4 \leftarrow (t_4, t_0) \gg \text{shift} 8;$
7. $t_3 \leftarrow \text{shift} (0x0000000000000001, 0xFFFFFFFFFFFFF); t_4 \leftarrow t_3 \land Z[1]; Z[1] \leftarrow Z[1] \land t_3;$
8. $t_1 \leftarrow t_4 \gg \text{shift} 1; t_0 \leftarrow (t_0, t_1) \gg \text{shift} 8; t_1 \leftarrow \text{shift} t_0;$
9. $Z[0] \leftarrow Z[0] \gg t_0; t_1 \leftarrow (t_1 \ll \text{shift} 15);$
10. $Z[0] \leftarrow (Z[0] \gg t_1; t_2 \leftarrow t_4 \gg \text{shift} 50; Z[0] \leftarrow Z[0] \gg t_2;$
11. return $z(x) = (Z[1], Z[0])$

**Algorithm 16**: Reduction in $GF(2^{233})$

**Input**: $a(x) = \sum_{i=0}^{233} a_i x^i = (A[3], A[2], A[1], A[0])$

**Output**: $z(x) = a(x) \mod x^{233} + x^{23} + 1$

1. $t_0 \leftarrow (A[3] \ll \text{shift} 33); t_1 \leftarrow (A[3] \gg \text{shift} 31); t_2 \leftarrow (A[3] \ll \text{shift} 23); t_3 \leftarrow (A[3] \gg \text{shift} 41);$
2. $t_4 \leftarrow t_0 \gg t_3; t_3 \leftarrow t_4 \gg \text{shift} 8;$
4. $t_0 \leftarrow (A[2] \gg \text{shift} 33); t_1 \leftarrow (A[2] \gg \text{shift} 31); t_2 \leftarrow (A[2] \gg \text{shift} 23); t_3 \leftarrow (A[2] \gg \text{shift} 41);$
5. $t_5 \leftarrow t_0 \gg t_3; t_3 \leftarrow (t_4, t_5) \gg \text{shift} 8;$
6. $Z[0] \leftarrow A[0] \gg t_2; Z[1] \leftarrow Z[1] \gg t_1 \gg t_5;$
7. $t_0 \leftarrow \text{shift} (0x0000001FFFFFFFF, 0xFFFFFFFFFFFF); t_0 \leftarrow t_2 \ll Z[1]; Z[1] \leftarrow Z[1] \ll t_2;$
8. $t_1 \leftarrow t_0 \gg \text{shift} 41; t_1 \leftarrow (t_5, t_0) \gg \text{shift} 8;$
9. $Z[0] \leftarrow Z[0] \gg t_1; t_1 \leftarrow t_0 \gg \text{shift} 10; Z[0] \leftarrow Z[0] \gg t_1;$
10. return $z(x) = (Z[1], Z[0])$

**Algorithm 17**: Reduction in $GF(2^{409})$

**Input**: $a(x) = \sum_{i=0}^{409} a_i x^i = (A[6], A[5], A[4], A[3], A[2], A[1], A[0])$

**Output**: $z(x) = a(x) \mod x^{409} + x^{57} + 1$

1. $m_0 \leftarrow (A[6] \gg \text{shift} 2); m_1 \leftarrow (A[6] \ll \text{shift} 62); m_2 \leftarrow (A[6] \gg \text{shift} 25); m_3 \leftarrow (A[6] \ll \text{shift} 39);$
2. $m_4 \leftarrow (A[5] \gg \text{shift} 62); m_5 \leftarrow (A[5] \ll \text{shift} 62); m_6 \leftarrow (A[5] \gg \text{shift} 25); m_7 \leftarrow (A[5] \ll \text{shift} 39);$
3. $m_8 \leftarrow (A[4] \gg \text{shift} 62); m_9 \leftarrow (A[4] \ll \text{shift} 62); m_{10} \leftarrow (A[4] \gg \text{shift} 25); m_{11} \leftarrow (A[4] \ll \text{shift} 39);$
4. $t_0 \leftarrow m_1 \gg m_2;$
5. $Z[3] \leftarrow A[3] \gg t_0; t_1 \leftarrow m_4 \gg m_3; t_2 \leftarrow (m_0, t_1) \gg \text{shift} 8;$
6. $Z[3] \leftarrow Z[3] \gg t_2; t_0 \leftarrow m_5 \gg m_6;$
7. $Z[2] \leftarrow A[2] \gg t_0; m_7 \leftarrow m_7 \gg m_8; t_1 \leftarrow (t_1, m_7) \gg \text{shift} 8;$
8. $Z[2] \leftarrow Z[2] \gg t_1; t_2 \leftarrow m_9 \gg m_{10};$
10. $t_0 \leftarrow \text{shift} (0xFFFFFFFFFFFFF, 0xFFFFFFFFE0000000); t_0 \leftarrow Z[3] \gg t_0; Z[3] \leftarrow Z[3] \gg t_0;$
11. $m_0 \leftarrow (t_0 \gg \text{shift} 2); m_1 \leftarrow (t_0 \ll \text{shift} 62); m_2 \leftarrow (t_0 \gg \text{shift} 25); m_3 \leftarrow (t_0 \ll \text{shift} 39);$
12. $t_0 \leftarrow m_{11} \gg m_0; t_1 \leftarrow (m_7, t_0) \gg \text{shift} 8;$
13. $Z[1] \leftarrow Z[1] \gg t_1; t_2 \leftarrow m_1 \gg m_2;$
14. $Z[0] \leftarrow A[0] \gg t_2; t_0 \leftarrow (t_0, m_3) \gg \text{shift} 8; Z[0] \leftarrow Z[0] \gg t_0;$
15. return $z(x) = (Z[3], Z[2], Z[1], Z[0])$

39
64-Bit Reduction in $GF(2^{239})$

The polynomial $f(x) = x^{239} + x^{158} + 1$ is one of the two polynomials suggested in [56] for this field size and currently integrated in OpenSSL. For the implementation with vector extensions, this polynomial is suboptimal. This is caused by the small distance $239 - 158 = 81 < 128$, which means that a 128 bit word of the remainder affects itself during the reduction, which makes shift and add reduction inefficient for this size. The $GF(2^{239})$ reduction is therefore provided in 64 bit mode only. We found that reduction with the polynomial $f(x) = x^{239} + x^{36} + 1$ (which has also been suggested by SECG) is much faster and therefore obviously the better choice for this field when using vector extensions.

Due to this issues we do not recommend the use of this field, because reduction in $GF(2^{233})$ is much faster and this particular curve is located at almost the same security level (see 2.2) with an only slightly shorter key. Anyhow, we added this curve to our implementation for convenience and to compare the reduction with and without vector extensions.

Pentanomial Reduction

The strategy for pentanomials differs from the trinomial way in terms of recombination. Instead of recombining each component, we gather the results of left and right shifts of a vector $A$ in separated vectors and recombine them by finally adding them to the remainder, which saves three alignments per 128 bit element. Figure 4.4 shows the combination steps for a 128 bit vector $A$.

Unfortunately, this strategy does not work together with the inter-component fast shift trick since the previous results are only available in split representation, however, the amount of saved 128 bit shifts for the re-alignment pay off in comparison.
We suggest Algorithm 18 for $GF(2^{163})$ and 19 for the reduction in $GF(2^{571})$.

**Algorithm 18: Reduction in $GF(2^{163})$**

Input: $a(x) = \sum_{i=0}^{325} a_i x^i = (A[2], A[3], A[0])_{128}$

Output: $z(x) = a(x) \mod x^{163} + x^9 + x^4 + x^1 + 1$

1. $t_0 \leftarrow A[2] \gg 64 \ 35$; $t_1 \leftarrow A[2] \ll 64 \ 29$;
2. $t_2 \leftarrow A[2] \gg 128 \ 4$; $t_3 \leftarrow t_1 \oplus t_3$;
3. $t_2 \leftarrow A[2] \gg 64 \ 29$; $t_3 \leftarrow A[2] \ll 64 \ 35$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
4. $t_2 \leftarrow A[2] \gg 64 \ 28$; $t_3 \leftarrow A[2] \ll 64 \ 36$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
5. $t_2 \leftarrow t_1 \ll 128 \ 8$; $t_1 \leftarrow t_1 \gg 128 \ 8$; $t_0 \leftarrow t_0 \oplus t_1$;
6. $Z[0] \leftarrow A[0] \oplus t_2$;
8. $t_4 \leftarrow 64 \ (0x0FFFFFFF0FFFFF0, 0x0FFFFFFF0FFFFF0)$;
9. $t_4 \leftarrow Z[1] \land t_4$; $Z[1] \leftarrow Z[1] \land t_1$;
10. $t_0 \leftarrow t_4 \gg 64 \ 35$; $t_1 \leftarrow t_4 \ll 64 \ 29$; $t_2 \leftarrow t_4 \gg 128 \ 4$; $t_3 \leftarrow t_0 \oplus t_2$;
11. $t_2 \leftarrow t_4 \gg 64 \ 29$; $t_3 \leftarrow t_4 \ll 64 \ 35$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
12. $t_2 \leftarrow t_4 \gg 64 \ 28$; $t_3 \leftarrow t_4 \ll 64 \ 36$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
13. $t_1 \leftarrow t_1 \gg 128 \ 8$; $t_0 \leftarrow t_0 \oplus t_1$;
14. $Z[0] \leftarrow Z[0] \oplus t_0$;
15. return $z(x) = (Z[1], Z[0])_{128}$

**Algorithm 19: Reduction in $GF(2^{571})$**

Input: $a(x) = \sum_{i=0}^{144} a_i x^i = (A[8], ..., A[0])_{128}$

Output: $z(x) = a(x) \mod x^{571} + x^{18} + x^{14} + x^{13} + 1$

1. $t_4 \leftarrow \text{ZERO}$;
2. for $(i = 8; i > 4; i--)$ do
3. $t_0 \leftarrow A[i] \ll 64 \ 5$; $t_1 \leftarrow A[i] \gg 64 \ 59$;
4. $t_2 \leftarrow A[i] \ll 64 \ 7$; $t_3 \leftarrow A[i] \gg 64 \ 57$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
5. $t_2 \leftarrow A[i] \ll 64 \ 10$; $t_3 \leftarrow A[i] \gg 64 \ 54$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
6. $t_2 \leftarrow A[i] \ll 64 \ 15$; $t_3 \leftarrow A[i] \gg 64 \ 49$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
7. $t_2 \leftarrow (t_4, t_0) \gg 8$;
8. $A[i-4] \leftarrow A[i-4] \oplus t_2 \oplus t_1$;
9. $t_4 \leftarrow t_0$;
end
10. $t_0 \leftarrow t_4 \ll 128 \ 8$;
11. $A[1] \leftarrow t_4 \oplus t_0$;
12. $t_4 \leftarrow (t_4 \ll 64 \ 50, t_1 \ll 64 \ 5)$;
13. $t_2 \leftarrow (t_4 \gg 64 \ 57, t_3 \ll 64 \ 7)$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
14. $t_2 \leftarrow (t_4 \gg 64 \ 54, t_3 \ll 64 \ 10)$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
15. $t_2 \leftarrow (t_4 \gg 64 \ 49, t_3 \ll 64 \ 15)$; $t_0 \leftarrow t_0 \oplus t_2$; $t_1 \leftarrow t_1 \oplus t_3$;
16. $t_1 \leftarrow t_1 \ll 128 \ 8$;
17. $A[0] \leftarrow A[0] \oplus t_0 \oplus t_1$;

41
Reduction in $GF(2^{283})$

Although the reduction polynomial for $GF(2^{283})$ is a pentanomial, the reduction can be sped up with the observation

$$f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$$
$$= x^{283} + (x^7 + 1) \cdot (x^5 + 1).$$

Algorithm 20 shows a fast reduction of elements in $GF(2^{283})$ as suggested in [7], where we also refer to for further information about this special reduction.

Algorithm 20: Reduction in $GF(2^{283})$

```
Input: $a(x) = \sum_{i=0}^{565} a_i x^i = (A[4], ..., A[0])_{128}$
Output: $z(x) = a(x) \mod x^{283} + (x^7 + 1) \cdot (x^5 + 1)$
1 $t_0 \leftarrow (A[3], A[2]) \gg 8; t_1 \leftarrow (A[4], A[3]) \gg 8; t_3 \leftarrow A[2];$
4 $A[2] \leftarrow A[2] \oplus t_2; t_0 \leftarrow (A[4], A[3]) \gg 8 15; t_2 \leftarrow t_0 \gg 64 1;$
5 $A[4] \leftarrow A[4] \oplus t_2; t_1 \leftarrow (A[3], A[2]) \gg 8 8; t_2 \leftarrow A[3] \ll 64 7;$
6 $A[3] \leftarrow A[3] \oplus t_2; t_2 \leftarrow t_1 \gg 64 57; A[3] \leftarrow A[3] \oplus t_2;$
7 $t_0 \leftarrow (A[2], ZERO) \gg 8 8; t_2 \leftarrow A[2] \ll 64 7;$
8 $A[2] \leftarrow A[2] \oplus t_2; t_2 \leftarrow t_1 \gg 64 57;$
9 $A[2] \leftarrow A[2] \oplus t_2; t_0 \leftarrow (A[4], A[3]) \gg 8 15; t_1 \leftarrow t_0 \gg 64 3;$
10 $A[4] \leftarrow A[4] \oplus t_1; t_1 \leftarrow (A[3], A[2]) \gg 8 8; t_2 \leftarrow A[3] \ll 64 7;$
11 $A[3] \leftarrow A[3] \oplus t_2; t_2 \leftarrow t_1 \gg 64 57;$
12 $A[3] \leftarrow A[3] \oplus t_2; t_0 \leftarrow (A[2], ZERO) \gg 8 8; t_2 \leftarrow A[2] \ll 64 5;$
13 $A[2] \leftarrow A[2] \oplus t_2; t_2 \leftarrow t_1 \gg 64 59;$
15 $Z[0] \leftarrow A[0] \oplus A[2];$
17 $Z[2] \leftarrow t_3 \oplus A[4];$
18 $t_0 \leftarrow A[4] \gg 64 27; t_2 \leftarrow t_0 \ll 64 5; t_1 \leftarrow t_0 \oplus t_2;$
19 $t_2 \leftarrow t_1 \gg 64 7; t_0 \leftarrow t_1 \oplus t_2;$
20 $Z[0] \leftarrow Z[0] \oplus t_0; t_2 \leftarrow 64 (0x0000000000000000, 0x0000000007FFFFFF);$
21 $Z[2] \leftarrow Z[2] \land t_2;$
22 return $z(x) = (Z[2], Z[1], Z[0])_{128}$
```

Reduction Scheme Using Random Multiplier

In this subsection we want to discuss a reduction scheme using the pclmulqdq instruction. Here, we want to analyze the practicability of reduction with a carry-less random 64 bit multiplier as available on the target platforms.

Each 64 bit word from the operand $c_H(x)$ must be multiplied with $r(x) = \sum_{i=0}^{n} r_i x^i$ and added to the remainder in $c_L(x)$. Obviously, $n$ must hold the condition $n < 64$ to ensure the lowest number of multiplications, which is exactly $t = \left\lceil \frac{\log_2(c_H(x))}{64} \right\rceil$ then. If $n > 63$, we must multiply each word more than once, particularly $s = \left\lceil \frac{n}{63} \right\rceil$ times per word and we therefore require $t \cdot s$ multiplications overall.

In order to spare the additional reduction step, as described above, the multiplications are executed from the left to the right and the results are immediately added to the remainder. Due to the fact that the bits of $c(x)$ are results from either multiplication or squaring and are stored into a series of words
where the border between $c_L(x)$ and $c_H(x)$ at position $m - 1$ is right in the middle of a register, each result of a remainder multiplication must be aligned in order to be added to the remainder. Therefore, we can either shift the polynomial $r(x)$ in advance to the left by $k = m \mod 64$ bits if $n + k < 64$ still holds, or we can shift the input words $k$ bits to the right, or $64 - k$ bits to the left. Thereby, the operand needs to be saved temporarily as the reduction is performed in-place, requiring more temporary variables.

Figure 4.5.: Optimal operand shifting for multiplication

Shifting the operand especially makes sense if the data in $c_H(x)$ overlaps into an additional register, thus requiring $t + 1$ multiplications as we can see in figure 4.5 and algorithm 21. In this example, the reduction in $GF(2^{163})$, $c_H(x) = \sum_{i=163}^{325} c_i x^i$ is spread over four 64 bit words. Algorithm 22 shows how shifting the operand enables to perform the reduction in just three multiplication steps.

Algorithm 21: Reduction in $GF(2^{163})$ using PCLMULQDQ

| Input: $a(x) = \sum_{i=0}^{325} a_i x^i = (A[2], A[1], A[0])_{128}$ |
| Output: $z(x) = a(x) \mod x^{163} + x^7 + x^6 + x^3 + 1$ |

1. $p \leftarrow 64 \ (0x0000000000000000, 0x00000019200000000)$;
2. $t_0 \leftarrow A[2]_H \times p_L$;
3. $Z[1] \leftarrow A[1] \oplus t_0; \ t_3 \leftarrow A[2]_L \times p_L; \ t_1 \leftarrow t_3 \gg 128$ 8;
4. $Z[1] \leftarrow Z[1] \oplus t_1$;
5. $t_1 \leftarrow 64 \ (0xFFFFFFFFFFFFFFFF, 0xFFFFFFFF8000000000)$;
6. $t_2 \leftarrow Z[1] \wedge t_1; \ Z[1] \leftarrow Z[1] \wedge t_1; \ t_0 \leftarrow t_2,H \times p_L$;
7. $Z[0] \leftarrow A[0] \oplus t_0; \ t_1 \leftarrow t_2,L \times p_L; \ t_1 \leftarrow (t_3, t_1) \gg 8$ 8;
8. $Z[0] \leftarrow Z[0] \oplus t_1$;
9. return $z(x) = (Z[1], Z[0])_{128}$
Consider the case $m < 128$, the straightforward shift and add process requires another correction step, if the shifts are done in 256 bit registers, since the distance between the topmost and the second topmost bit in the reduction polynomial is very large, we are able to fetch 8 multiplications consecutively, which helps us benefit from the smaller throughput of the pclmulqdq.

Obviously, this proposed technique makes more sense the greater the hamming weight of $r(x)$ is, and if the number of multiplications is low, meaning all the bits in $r(x)$ are in the first 64 bit word ($s = 1$). Let the reduction polynomial be a pentanomial with no special properties (no fast shifts or other exploitable properties), then the required amount of operations with shift&add reduction for each 128 bit word is

- 4-2 SHIFT’s ($psllq/psrlq$
- 3-2 XOR’s ($pxor$
- 1 Memory Alignment ($palignr/pslldq/pssldq$).

The multiplier must be able to perform 2 multiplications faster than the time required for the operations above in order to beat the shift&add implementation for pentanomial reduction polynomials. We will discuss this further in chapter 5 and give more detailed numbers. However, this reduction scheme has the further benefit of working together with the shift&add approach. Especially with the forecast to AVX2, this reduction method can be very useful in combination with the generic shift&add approach. Considering the case $m < 256$, the straightforward shift and add process requires another correction step, if the shifts are done in 256 bit registers, since $r(x) \cdot c_H(x) > m$. For this correction step, we can use the pclmulqdq-reduction to avoid repeating the whole shift&add process for the overlapping bits. Please see section 4.4 and Algorithm 24 for an example.

### Barrett Reduction

Another famous reduction method is the Barrett reduction, introduced in 1989 by P.D.Barrett. In [38], Knežević et al. presented a Barrett reduction method not requiring precomputation for $GF(2^m)$. This was, for instance implemented by Intel engineers [39] in a paper about the benefits of the pclmulqdq instruction for the ECC. Compared to the shift&add approach, the Barrett reduction turned out considerably slower in our experiments.
### 4.2.5. Inversion

For the inversion in $GF(2^m)$ we use the Itoh-Tsujii Algorithm (ITA)\[^3\] using the property \[^2\] in $GF(2^n)$. Let $a(x), b(x) \in GF(2^m)$. With equation \[^2\] we follow that
\[
a^{-1}(x) = (a^2)^{m-1} - (x) = b^{m-1} - (x) = b^{1+2+2^2+\ldots+2^{m-2}} \mod f(x)
\] (4.2)
where each “$+$” in $b$’s exponent is a multiplication. Thus, the goal is to find a decomposition of the chain $1+2^1+2^2+\ldots+2^{m-2}$ resulting in a low number of multiplications. The ITA performs this exponentiation with exactly $m-1$ squares and $\lfloor \log_2(m-1) \rfloor + H(m-1) - 1$ multiplications with $H(m-1)$ the Hamming weight of $(m-1)$. The general recursive formula to obtain the chain can be given as
\[
\begin{split}
1+2^n+2^{2n}+\ldots+2^{(k-2)n} = \left\{
\begin{array}{ll}
(1 + 2^n) \times (1 + 2^{2n}) \times \ldots \times (1 + 2^{(k-2)n}), & \text{if } k-1 \equiv 0 \mod 2 \\
1 + (2^n \times (1 + 2^n) \times \ldots \times (1 + 2^{(k-2)n})), & \text{if } k-1 \equiv 1 \mod 2
\end{array}
\right.
\end{split}
\] (4.3)

In this work, we use the following decompositions for the finite fields of characteristic 2 as follows.

<table>
<thead>
<tr>
<th>Field</th>
<th>Decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GF(2^{163})$</td>
<td>$(1 + 2)(1 + 2^2)(1 + 2^3)(1 + 2^4)(1 + 2^{16})(1 + 2^{32})(1 + 2^{64})(1 + 2^{128})$</td>
</tr>
<tr>
<td>$GF(2^{193})$</td>
<td>$(1 + 2)(1 + 2^3)(1 + 2^5)(1 + 2^6)(1 + 2^{12})(1 + 2^{24})(1 + 2^{48})(1 + 2^{96})$</td>
</tr>
<tr>
<td>$GF(2^{233})$</td>
<td>$(1 + 2)(1 + 2^2)(1 + 2^4)(1 + 2^5)(1 + 2^{10})(1 + 2^{20})(1 + 2^{40})(1 + 2^{80})$</td>
</tr>
<tr>
<td>$GF(2^{283})$</td>
<td>$(1 + 2)(1 + 2^2)(1 + 2^3)(1 + 2^5)(1 + 2^{10})(1 + 2^{20})(1 + 2^{40})(1 + 2^{80})(1 + 2^{160})$</td>
</tr>
<tr>
<td>$GF(2^{383})$</td>
<td>$(1 + 2)(1 + 2^3)(1 + 2^5)(1 + 2^6)(1 + 2^{12})(1 + 2^{24})(1 + 2^{48})(1 + 2^{96})$</td>
</tr>
<tr>
<td>$GF(2^{409})$</td>
<td>$(1 + 2)(1 + 2^2)(1 + 2^3)(1 + 2^4)(1 + 2^{10})(1 + 2^{20})(1 + 2^{40})(1 + 2^{80})(1 + 2^{160})$</td>
</tr>
<tr>
<td>$GF(2^{277})$</td>
<td>$(1 + 2)(1 + 2^2)(1 + 2^3)(1 + 2^4)(1 + 2^{10})(1 + 2^{20})(1 + 2^{40})(1 + 2^{80})(1 + 2^{160})$</td>
</tr>
</tbody>
</table>

In the following, we will give an example for inversion in $GF(2^{193})$. The algorithms for inversion in the other fields are very similar and obtained accordingly. Please note that an exponentiation $t^{2^n}$ means the $x$-times repeated squaring of $t$, modulo the field polynomial $f(x)$ and $\times$ the modular multiplication. With the assumption that a multiplication is three times as expensive as a square, we can easily see that the running time boundaries for this algorithm is mainly determined by the squares. Although there are techniques to accelerate the squares (e.g., multi-squaring\[^{13}\]), we do not use these in this work, not only because of the required table lookups, which we want to avoid in our implementation, but mostly because the costly inversion is only needed once per point multiplication. The inversion algorithm uses different amounts of temporary variables, depending on the fields decomposition. We will show timings and memory consumption of each inversion algorithm in the second to last chapter \[^5\] in detail.

**Algorithm 23:** Example inversion in $GF(2^{193})$

**Input:** $a(x) \in GF(2^{193})$

**Output:** $a^{-1}(x) = a^{2^{m-1}}$

1. $t_0 \leftarrow a^2$;
2. // Initial square
   2. $t_0 \leftarrow (t_0^2)^1 \times t_0$;
   3. $t_0 \leftarrow (t_0^2)^2 \times t_0$;
   4. $t_0 \leftarrow (t_0^2)^4 \times t_0$;
   5. $t_0 \leftarrow (t_0^2)^8 \times t_0$;
   6. $t_0 \leftarrow (t_0^2)^16 \times t_0$;
   7. $t_0 \leftarrow (t_0^2)^32 \times t_0$;
   8. $t_0 \leftarrow t_0$;
   9. $t_0 \leftarrow (t_0^2)^64 \times t_0$;
   10. $t_0 \leftarrow (t_0^2)^64 \times t_1$;
11. return $t_0 = a^{-1}(x)$

---

45
4.3. OpenSSL Integration

In order to adapt our implementation to OpenSSL, we needed to add certain additional extensions to the library. For the ECC, OpenSSL implements a mechanism to enable the developers to easily add improved methods. Therefore, it uses a **struct** called `EC_METHOD`, which is used to provide an interface to access all functions and parameter concerning the curve, group, points and expensive field operations like multiplication, division and squaring. This struct can be used for curves over both binary and prime fields and further provides the interface to precomputation methods and tables.

A big issue concerning the constant time processing is the OpenSSL implementation of the `GF2m` arithmetic, which does not allow zero words at the top of a `BIGNUM` number, as most functions do a correction step (`bn_correct_top(BIGNUM *)`) before processing the number. This is of course useful regarding the efficiency but does not work together with the constant element size condition. To circumvent this issue, we included certain functions to process constant size elements. As mentioned above, the `EC_METHOD` structure enables to add fast implementations for some field arithmetics. The current version, OpenSSL-1.0.1e, generally implements the binary curve arithmetic in affine coordinates only and transforms the coordinates into the LD-projective format for the Montgomery point multiplication. However, the usual functions for point doubling and addition are implemented separately and, in order to use the fast binary field implementations, need to be fixed to avoid an application crash when loading elements with leading zero words. Thus, all functions dealing with loading, storing or processing of field elements of any kind require to be rewritten to prevent cutting off leading zero words. Additionally, the creation of field elements has been separated from its processing to avoid data dependent operations.

In the implementation it is necessary to identify critical and non-critical data, as non-critical data does not require to be processed in constant time. An example for non-critical data are public curve parameters such as $a$ and $b$. Whenever the operation involves any critical data, though, it must be processed constantly. However, the upper implementation logic such as the implementation for the `EC_sign`, `EC_verify` and other functions are requiring the data in the original format. Therefore, we need to transform the data out of the constant size context into the usual `BIGNUM` format and cut off the leading zeros as done in the function `ec_GF2m_nist_point_get_affine_coordinates`. At this point, we also want to mention that our implementation does not offer a constant time implementation for the whole EC cryptosystem but for the LD/Montgomery point multiplication only. Fixing the EC cryptosystem to constant time element processing would require more changes on the higher implementation logic, especially to the point addition and doubling methods and additional binary field functions.

4.4. Expected Implementation Benefits with AVX2

In this section, we will give a forecast on how to apply further improvements on the binary field arithmetic with the use of the coming INTEL AVX2 instruction set.

In general, all field arithmetic functions will benefit from the logical and arithmetic functions for the 256 bit vectors introduced with AVX2. However, some functions are expected to gain more than others. The multiplication is expected to gain the least from AVX2, since the carry-less multiplier size will remain the same, although it will be much faster on newer architectures, which is however no performance gain from AVX2 itself but an upgrade for the `pclmulqdq`. The additions for the recombination of the multiplication results will become faster, as the addition itself is expected to perform twice as fast as the SSE4 128 bit `XOR`.

The field element squaring is expected to gain the most from AVX2, particularly through the advanced 256 bit `vpshufb` instruction. The implementation is more or less the same as for 128 bit with only half the number of blocks processed, resulting in a speedup of factor 2 compared to our current implementation.

The field reduction is another promising application case of the AVX2. The extension includes 64 bit shifts for 256 bit vectors and enables the programmer to decrease the amount of shifts in the
shift&add implementations we have seen before. Unfortunately, using this shift instruction introduces new problems. The shift&add approach wins from the fact that the scheme is sequentially adding the upper words from $c_H(x)$ to the lower part $c_L(x)$. If $c_H(x)$ (and thus $c_L(x)$) fits into one 256 bit vector, the problem described in 4.2.4 re-appears and the result $c'(x) = c_L(x) \oplus c_H(x) \cdot r(x)$ must again be reduced for the topmost $|r(x)|$ bits. Additionally, the recombination can become costly for certain fields.

In the following, we will give an example for how reduction could look like in $GF(2^{163})$. We included our thoughts from page 42, where we suggested to use the pclmulqdq instruction to speed up the reduction. Regarding the example below, the pclmulqdq reduction scheme proposed in Algorithm 22 could win depending on the performance of the carry-less multiplier. Expecting a latency of 5 cycles, the version below should win with narrow lead though.

**Algorithm 24: Mixed reduction in $GF(2^{163})$**

**Input:** $a(x) = \sum_{i=0}^{325} a_i x^i = (A[1], A[0])_{256}$

**Output:** $z(x) = a(x) \mod x^{163} + x^9 + x^3 + 1$

1. $p \leftarrow 64 \ (0x0000000000000000, 0x0000000000000000)$;
2. $mask \leftarrow 64 \ (0xFFFFFFFFFFFFFFFF, 0xFFFFFFFF80000000, 0x0000000000000000)$;
3. $t_1 \leftarrow A[0] \wedge mask; A[0] \leftarrow A[0] \wedge mask$;
4. $t_0 \leftarrow t_1 \gg 64 35; t_1 \leftarrow t_4 \ll 64 29$;
5. $t_3 \leftarrow t_4 \gg 256 8; t_2 \leftarrow t_0 \oplus t_4$;
6. $t_3 \leftarrow t_4 \ll 64 36; t_0 \leftarrow t_0 \oplus t_2$;
7. $t_1 \leftarrow t_1 \gg 256 8; t_0 \leftarrow t_0 \oplus t_1$;
8. $Z[0] \leftarrow A[0] \oplus t_0$;
9. $t_1 \leftarrow Z[0] \wedge mask; Z[0] \leftarrow Z[0] \wedge mask$;
10. $t_1 \leftarrow 128 t_1 \gg 256 16$;
11. $t_0 \leftarrow t_1 \times p_L; t_0 \leftarrow t_0 \gg 256 8$;
12. $Z[0] \leftarrow Z[0] \triangleq t_0$;
13. **return** $z(x) = Z[0]_{256}$
5. Measurement and Results

In this chapter we describe the measurement setup and discuss the results of our implementation.

5.1. Measurement Guidelines

In this section our measurement techniques and setup will be described to enable the reader to reproduce the results presented in this chapter.

The test machines were running with a Linux operating system. We collected results mainly for the Ivy Bridge (IVB) but also measured on Sandy Bridge (SNB) and Haswell (HSW) architecture. We used the GNU C Compiler Suite (gcc) in version 4.8.0 and 4.7.0 to build the source code.

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU Info</th>
<th>Freq.</th>
<th>GCC Version</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ivy Bridge</td>
<td>i5-3210M</td>
<td>2.5 GHz</td>
<td>4.8.0</td>
<td>IVB</td>
</tr>
<tr>
<td>Ivy Bridge</td>
<td>i7-3770</td>
<td>3.4 GHz</td>
<td>4.7.0</td>
<td>IVB</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>i7-2600</td>
<td>3.4 GHz</td>
<td>4.7.0</td>
<td>SNB</td>
</tr>
<tr>
<td>Haswell</td>
<td>i7-4770</td>
<td>2.9 GHz</td>
<td>4.7.0</td>
<td>HSW</td>
</tr>
</tbody>
</table>

In the past, several methods have been proposed to reduce the randomness in measurements. In this work, we follow the guidelines from ECRYPT Benchmarking of Cryptographic Systems (eBACS) [61], which suggest to:

1. disable Hyper-Threading,
2. run tests with a fixed frequency (thus disabling e.g., Turbo Boost),
3. run tests in idle mode (e.g., without X server running).

Linux kernel features as CPU hotplugging and frequency scaling help to prepare the computer for this task without requiring the change of BIOS settings. For the measurement itself, we used two different techniques:

- **Cyclecounter**
- **OpenSSL Speed Utility**

The **Cyclecounter** is a C MACRO and uses CPU functionality to count the cycles for a given test function. Our implementation can be found in the Appendix D.1 and works as follows for any given function `func`:

1. It repeats `func r/4` times, in order to warm the cache,
2. it reads the Timestamp Counter at the beginning of the test,
3. it repeats `func r` number of times,
4. it reads the Time Stamp Counter again at the end of the test,
5. it calculates the average number of cycles per one iteration of `func`, by calculating the total number of cycles, and dividing it by `r`.

This measurement technique has a higher precision than time measurement and is therefore especially useful to determine the execution times of low level functions such as the binary field arithmetics. We will always state the repetition factor, which is usually `r = 10^9`. This is important since, if `r` is too small, the measurement will show great deviations caused by the caching structure and other architectural side effects. However, the differences between the various runs are still not insignificant, thus we chose to run the tests several times and built the mean average, although, single results still may lack of accuracy.
The OpenSSL Speed Utility can be used to easily reproduce our results after applying the patch and indicates the patch’s impact to the overall performance of OpenSSL ECDH and ECDSA operations. This tool has been implemented by OpenSSL and averages the number of executions for a certain function for a given time distance.

5.2. Results

This section presents the results for the binary field arithmetic, shows the impact of the improved LD/Montgomery point multiplication flow and the resulting performance gain to the OpenSSL implementation.

5.2.1. Speed Results - Binary Field Arithmetic

In this section we show the results for the binary field arithmetic, mostly measured on the IVB1 as defined in the previous section.

Addition Cost

The costs for the addition in $GF(2^m)$ can be estimated by the costs for one XOR, multiplied by the amount of words. Assuming XOR is performed in one cycle with register size $W$ bits, the amount of cycles for one addition is $t_{add} = \lceil \frac{m}{W} \rceil$ and therefore twice as fast with vector extensions. Additionally, the throughput for XOR is 0.33 on the target platforms, so three instructions can be fetched simultaneously, which should lead to an overall speedup of factor 6 against implementation in 64 bit mode. Also, the AVX vxorpd instruction, as discussed in 4.2.1, can be applied for even multiples of 128 bit words, giving the same results for short numbers due to the load and store overhead (remember, latency 1/throughput 0.33) but paying off for four quadwords or more. However, we find the real numbers for the addition to be much slower than expected.

<table>
<thead>
<tr>
<th>Element Size</th>
<th>xor (64 bit)</th>
<th>pxor (128 bit)</th>
<th>pxor + vxorpd (128/256 bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 × 64</td>
<td>28</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3 × 64</td>
<td>36</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>4 × 64</td>
<td>39</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>5 × 64</td>
<td>52</td>
<td>11</td>
<td>9</td>
</tr>
<tr>
<td>7 × 64</td>
<td>65</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>9 × 64</td>
<td>78</td>
<td>17</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5.1.: Cycles for addition in $GF(2^m)$

Reduction Cost

In the table below, we included the reduction costs for the shift&add approach as well as for the pclmulqdq reduction scheme. The table shows the performance difference for trinomials and pentanomials, as well as the performance loss due to the 64 bit mode in $GF(2^{239})$. 


The results of the shift&add reduction are way better than for the pclmulqdq reduction. The performance mainly depends on the performance of the pclmulqdq instruction, and this reduction scheme starts to pay off when the multiplier performs two multiplications faster than

- $4 \times 2$ SHIFT's (psllq/psrlq)
- $3 \times 2 + 3$ XOR's (pxor)
- 1 Memory Alignment (palignr/pslldq/pslldq).

On the Sandy/Ivy Bridge this can be performed in about 11 cycles, whereas one 64 bit multiplication has a latency of 14 cycles. This explains why this reduction scheme does not pay off on these architectures, however, with a faster carryless multiplier it becomes more important in the future. The following table shows the results for a Haswell architecture.

<table>
<thead>
<tr>
<th>Field</th>
<th>Polynomial</th>
<th>Shift&amp;Add</th>
<th>PCLMULQDQ Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GF(2^{163})$</td>
<td>$f(x) = x^{163} + x^7 + x^6 + x^3 + 1$</td>
<td>19</td>
<td>18</td>
</tr>
<tr>
<td>$GF(2^{193})$</td>
<td>$f(x) = x^{193} + x^{15} + 1$</td>
<td>13</td>
<td>-</td>
</tr>
<tr>
<td>$GF(2^{233})$</td>
<td>$f(x) = x^{233} + x^{74} + 1$</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>$GF(2^{239})$</td>
<td>$f(x) = x^{239} + x^{158} + 1$</td>
<td>41</td>
<td>-</td>
</tr>
<tr>
<td>$GF(2^{283})$</td>
<td>$f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$</td>
<td>25</td>
<td>57</td>
</tr>
<tr>
<td>$GF(2^{293})$</td>
<td>$f(x) = x^{293} + x^{10} + x^5 + x^2 + 1$</td>
<td>48</td>
<td>81</td>
</tr>
</tbody>
</table>

Table 5.3.: Cycles for Reduction in $GF(2^m)$ on Haswell

The pclmulqdq reduction wins against the shift&add reduction, especially in $GF(2^{283})$ and $GF(2^{571})$ with 9 and 12 cycles respectively. However, further research is required to evaluate its benefit against shift&add when implemented with AVX2.

Squaring Cost

The cost for a simple field squaring is ridiculously cheap - it takes almost the same amount of cycles as for addition. The final field reduction makes the squaring much more expensive though. Also, we can observe that the merged squaring and reduction sums up close to the numbers for reduction only. Since the numbers match across the platforms, we do not give specific numbers for the different architectures.
### Multiplication Cost

The multiplication is an essential part of the binary arithmetic and highly influences the running time of the point multiplication. In the table below, we show numbers for the multiplication on all platforms for crossover comparison. As expected, the multiplier on the Haswell architecture is enormously faster, compared to the other architectures.

<table>
<thead>
<tr>
<th>MUL Size</th>
<th>Unreduced MUL</th>
<th>Field</th>
<th>Field MUL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IVB₁</td>
<td>IVB₂</td>
<td>SNB</td>
</tr>
<tr>
<td>3 × 3</td>
<td>58</td>
<td>63</td>
<td>63</td>
</tr>
<tr>
<td>4 × 4</td>
<td>92</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>4 × 4</td>
<td>92</td>
<td>93</td>
<td>93</td>
</tr>
<tr>
<td>5 × 5</td>
<td>120</td>
<td>126</td>
<td>126</td>
</tr>
<tr>
<td>7 × 7</td>
<td>227</td>
<td>245</td>
<td>247</td>
</tr>
<tr>
<td>9 × 9</td>
<td>316</td>
<td>352</td>
<td>354</td>
</tr>
</tbody>
</table>

Table 5.5.: Cycles for Multiplication in $GF(2^m)$

### Inversion/Division Cost

Inversions and divisions are the most costly operations in $GF(2^m)$ in polynomial basis representation. In the following we give the numbers on the IVB₂ for non-accelerated inversion and division in the binary field using the Itoh-Tsujii Inversion Algorithm. Additionally, table 5.6 shows the amount of multiplications and the storage used for a single inversion.

<table>
<thead>
<tr>
<th>Field</th>
<th>Inversion</th>
<th>Multiplications</th>
<th>Storage</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GF(2^{163})$</td>
<td>4,811</td>
<td>9</td>
<td>3 ×163 bits</td>
<td>4,878</td>
</tr>
<tr>
<td>$GF(2^{193})$</td>
<td>4,788</td>
<td>8</td>
<td>2 ×193 bits</td>
<td>4,895</td>
</tr>
<tr>
<td>$GF(2^{233})$</td>
<td>6,956</td>
<td>10</td>
<td>4 ×233 bits</td>
<td>7,062</td>
</tr>
<tr>
<td>$GF(2^{239})$</td>
<td>14,777</td>
<td>12</td>
<td>6 ×239 bits</td>
<td>14,905</td>
</tr>
<tr>
<td>$GF(2^{283})$</td>
<td>9,677</td>
<td>11</td>
<td>4 ×283 bits</td>
<td>9,808</td>
</tr>
<tr>
<td>$GF(2^{409})$</td>
<td>15,648</td>
<td>11</td>
<td>4 ×409 bits</td>
<td>15,873</td>
</tr>
<tr>
<td>$GF(2^{571})$</td>
<td>42,595</td>
<td>13</td>
<td>5 ×571 bits</td>
<td>42,983</td>
</tr>
</tbody>
</table>

Table 5.6.: Cycles for Inversions and Division in $GF(2^m)$
Lazy Reduction Cost

The lazy reduction technique as introduced in 3.2.2 saves the cost for one reduction, minus the costs for one field addition. We present numbers for lazy reductions for two different cases, adding the result of either two multiplications or one multiplication and one square and performing the field reduction afterwards. Please note that applying the lazy reduction to two square results does not make sense, since the addition can be done before the result is squared and reduced, which saves one square and one reduction.

The tables 5.7 and 5.8 show sample numbers to demonstrate the effect of the lazy reduction, measured on the IVB\textsubscript{2}. These results should be seen only as a hint, because the sum for the single operation costs are bigger than the real costs. This effect can as well be observed for the merged squaring and reduction 5.2.1.

<table>
<thead>
<tr>
<th>Field</th>
<th>SQR + MUL + RED</th>
<th>Estimated Costs</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GF(2^{163})$</td>
<td>76</td>
<td>$21 + 70 + 7 = 98$</td>
<td>22</td>
</tr>
<tr>
<td>$GF(2^{193})$</td>
<td>105</td>
<td>$19 + 99 + 9 = 127$</td>
<td>18</td>
</tr>
<tr>
<td>$GF(2^{233})$</td>
<td>103</td>
<td>$21 + 100 + 9 = 130$</td>
<td>27</td>
</tr>
<tr>
<td>$GF(2^{239})$</td>
<td>114</td>
<td>$44 + 108 + 9 = 161$</td>
<td>47</td>
</tr>
<tr>
<td>$GF(2^{283})$</td>
<td>153</td>
<td>$30 + 140 + 11 = 181$</td>
<td>28</td>
</tr>
<tr>
<td>$GF(2^{299})$</td>
<td>267</td>
<td>$31 + 254 + 14 = 299$</td>
<td>32</td>
</tr>
<tr>
<td>$GF(2^{371})$</td>
<td>416</td>
<td>$68 + 390 + 17 = 475$</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 5.7.: Cycles for Lazy Reductions in $GF(2^m)$, SQR + MUL

<table>
<thead>
<tr>
<th>Field</th>
<th>$2 \times$MUL + RED</th>
<th>Estimated Costs</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>$GF(2^{163})$</td>
<td>128</td>
<td>$2 \times 70 + 7 = 147$</td>
<td>19</td>
</tr>
<tr>
<td>$GF(2^{193})$</td>
<td>187</td>
<td>$2 \times 99 + 9 = 207$</td>
<td>20</td>
</tr>
<tr>
<td>$GF(2^{233})$</td>
<td>187</td>
<td>$2 \times 100 + 9 = 209$</td>
<td>22</td>
</tr>
<tr>
<td>$GF(2^{239})$</td>
<td>197</td>
<td>$2 \times 108 + 9 = 225$</td>
<td>28</td>
</tr>
<tr>
<td>$GF(2^{283})$</td>
<td>270</td>
<td>$2 \times 140 + 11 = 291$</td>
<td>21</td>
</tr>
<tr>
<td>$GF(2^{299})$</td>
<td>493</td>
<td>$2 \times 254 + 14 = 522$</td>
<td>29</td>
</tr>
<tr>
<td>$GF(2^{371})$</td>
<td>754</td>
<td>$2 \times 390 + 17 = 797$</td>
<td>43</td>
</tr>
</tbody>
</table>

Table 5.8.: Cycles for Lazy Reductions in $GF(2^m)$, MUL + MUL
5.2.2. Speed Results - Improved 2P Flow

This subsection points out the performance benefits of the improved point multiplication flow as introduced in section 3.2. Therefore, we patch a plain OpenSSL-1.0.1e with the improved Mdouble function, without any further improvements on the binary arithmetic level. In addition, we give numbers for the data veiling countermeasure we suggested in section 3.1.3.

Mdouble Cost

We have shown how to save one squaring (and reduction) in the code flow of Mdouble in section 3.2.1 and an additional multiplication (also plus reduction) for Koblitz curves. Table 5.9 shows the relative and absolute savings for a simple implementation of the point multiplication to demonstrate the effect for the EC sign/verify operations.

<table>
<thead>
<tr>
<th>Binary Curve</th>
<th>OpenSSL-1.0.1e</th>
<th>Patched OpenSSL</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign</td>
<td>Verify</td>
<td>Sign</td>
</tr>
<tr>
<td>sect163k1</td>
<td>957,446</td>
<td>1,747,006</td>
<td>849,716</td>
</tr>
<tr>
<td>sect163r1</td>
<td>999,417</td>
<td>1,839,900</td>
<td>939,033</td>
</tr>
<tr>
<td>sect163r2</td>
<td>998,869</td>
<td>1,810,380</td>
<td>932,775</td>
</tr>
<tr>
<td>sect193r1</td>
<td>1,041,955</td>
<td>1,894,347</td>
<td>976,858</td>
</tr>
<tr>
<td>sect193r2</td>
<td>1,042,466</td>
<td>1,889,547</td>
<td>972,166</td>
</tr>
<tr>
<td>sect233k1</td>
<td>1,206,539</td>
<td>2,215,841</td>
<td>1,053,952</td>
</tr>
<tr>
<td>sect233r1</td>
<td>1,234,760</td>
<td>2,306,159</td>
<td>1,165,043</td>
</tr>
<tr>
<td>sect239k1</td>
<td>1,231,252</td>
<td>2,276,507</td>
<td>1,088,732</td>
</tr>
<tr>
<td>sect283k1</td>
<td>2,172,917</td>
<td>4,133,936</td>
<td>1,925,247</td>
</tr>
<tr>
<td>sect283r1</td>
<td>2,285,954</td>
<td>4,382,237</td>
<td>2,155,617</td>
</tr>
<tr>
<td>sect409k1</td>
<td>3,608,986</td>
<td>6,984,209</td>
<td>3,278,985</td>
</tr>
<tr>
<td>sect409r1</td>
<td>3,893,633</td>
<td>7,482,653</td>
<td>3,716,780</td>
</tr>
<tr>
<td>sect571k1</td>
<td>8,291,314</td>
<td>16,081,911</td>
<td>7,470,456</td>
</tr>
<tr>
<td>sect571r1</td>
<td>8,920,204</td>
<td>17,539,995</td>
<td>8,486,708</td>
</tr>
</tbody>
</table>

Table 5.9.: Cost Saving for Mdouble

Costs for Data Veiling

Table 5.10 shows the costs for obscuring the coordinates in order to assure a fixed memory access pattern per round. A single data veiling requires twelve logical operations at the cost of an XOR. Additionally, one requires four temporary variables in order to hide the data processing for all four coordinates involved in the point multiplication. This has of course a big performance impact on the point multiplication, as this performance overhead applies for each loop execution. However, we think that the consequences in terms of security - better side channel resistance - is worth the cost.

<table>
<thead>
<tr>
<th>Field</th>
<th>128 bit Words</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF(2^{163})</td>
<td>2 × 2</td>
<td>2 × 29</td>
</tr>
<tr>
<td>GF(2^{193})</td>
<td>2 × 3</td>
<td>2 × 35</td>
</tr>
<tr>
<td>GF(2^{233})</td>
<td>4 × 4</td>
<td>2 × 42</td>
</tr>
<tr>
<td>GF(2^{239})</td>
<td>5 × 5</td>
<td>2 × 48</td>
</tr>
</tbody>
</table>

Table 5.10.: Data Veiling costs per round
5.2.3. Speed Results - ECDH/ECDSA

In this section, we want to demonstrate the performance impact of our proposed binary field library, the improved code flow and the impact of the implemented side channel countermeasures to the OpenSSL library. This thesis is targeting the point multiplication, but impacts on the results for the whole elliptic curve cryptosystem. To address this matter, we present our results with the use of different measurement techniques. In this section, we will not give specific numbers for each architecture, since the relative speedups do not significantly differ.

Point Multiplication and ECDH

The importance of the point multiplication has been pointed out several times by now. The next table shows results for the point multiplication \( k \cdot P \) in cycles. We find that the amount of cycles have been substantially diminished up to a factor of almost 10. Also, we observe the impact of the improved code flow for the Koblitz rather than Random curves, which is not tremendous but noticeable in both absolute numbers and the relative speedup.

<table>
<thead>
<tr>
<th>Binary Curve</th>
<th>OpenSSL-1.0.1e</th>
<th>Patched OpenSSL</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>sect163k1</td>
<td>624,949</td>
<td>66,378</td>
<td>9.42</td>
</tr>
<tr>
<td>sect163r1</td>
<td>643,236</td>
<td>76,534</td>
<td>8.40</td>
</tr>
<tr>
<td>sect163r2</td>
<td>646,374</td>
<td>77,238</td>
<td>8.37</td>
</tr>
<tr>
<td>sect193r1</td>
<td>801,932</td>
<td>121,127</td>
<td>6.62</td>
</tr>
<tr>
<td>sect193r2</td>
<td>818,985</td>
<td>121,163</td>
<td>6.76</td>
</tr>
<tr>
<td>sect233k1</td>
<td>993,195</td>
<td>128,474</td>
<td>7.73</td>
</tr>
<tr>
<td>sect233r1</td>
<td>1,030,043</td>
<td>148,230</td>
<td>6.95</td>
</tr>
<tr>
<td>sect239k1</td>
<td>1,027,541</td>
<td>185,629</td>
<td>5.54</td>
</tr>
<tr>
<td>sect283k1</td>
<td>1,892,016</td>
<td>217,192</td>
<td>8.71</td>
</tr>
<tr>
<td>sect283r1</td>
<td>2,019,134</td>
<td>254,163</td>
<td>7.94</td>
</tr>
<tr>
<td>sect409k1</td>
<td>3,315,895</td>
<td>510,787</td>
<td>6.49</td>
</tr>
<tr>
<td>sect409r1</td>
<td>3,655,279</td>
<td>599,002</td>
<td>6.10</td>
</tr>
<tr>
<td>sect571k1</td>
<td>7,659,288</td>
<td>1,094,765</td>
<td>7.00</td>
</tr>
<tr>
<td>sect571r1</td>
<td>8,432,393</td>
<td>1,270,666</td>
<td>6.64</td>
</tr>
</tbody>
</table>

Table 5.11.: OpenSSL Point Multiplication in \( GF(2^m) \)

Hereby, we have to mention that the numbers for the unpatched OpenSSL-1.0.1e are not constant, since it does not offer a constant time implementation, especially the bit length of \( k \) is not fixed. Therefore, the **speedup factor** is not fixed and will differ from run to run. The following graphic shows the speedup for the different curves. At minimum, we have a speedup factor of around five (six without the suboptimal \( GF(2^{239}) \)), whereas the highest factor is almost ten. This means a significant improvement of the point multiplication over binary curves.

Another obvious point is the great impact of the binary field reduction on the results. As we have seen in the implementation part (see page 40), the reduction in \( GF(2^{239}) \) was implemented in 64 bit mode due to the special form of the remainder polynomial.
The next table 5.12 shows the speed results for 'ECDH operations' per second, which is basically a point multiplication (see 2.3.2).

<table>
<thead>
<tr>
<th>Binary Curve</th>
<th>OpenSSL-1.0.1e (OP/s)</th>
<th>Patched OpenSSL (OP/s)</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>nistk163</td>
<td>4,129.0</td>
<td>34,739.0</td>
<td>8.41</td>
</tr>
<tr>
<td>nistk233</td>
<td>2,628.4</td>
<td>17,013.4</td>
<td>6.47</td>
</tr>
<tr>
<td>nistk283</td>
<td>1,338.5</td>
<td>9,017.6</td>
<td>6.74</td>
</tr>
<tr>
<td>nistk409</td>
<td>836.3</td>
<td>3,884.0</td>
<td>4.64</td>
</tr>
<tr>
<td>nistk571</td>
<td>406.5</td>
<td>1,824.1</td>
<td>4.49</td>
</tr>
<tr>
<td>nistb163</td>
<td>3,217.6</td>
<td>24,237.2</td>
<td>7.53</td>
</tr>
<tr>
<td>nistb233</td>
<td>2,443.5</td>
<td>13,114.3</td>
<td>5.37</td>
</tr>
<tr>
<td>nistb283</td>
<td>1,244.3</td>
<td>7,711.6</td>
<td>6.20</td>
</tr>
<tr>
<td>nistb409</td>
<td>723.4</td>
<td>3,382.3</td>
<td>4.68</td>
</tr>
<tr>
<td>nistb571</td>
<td>368.3</td>
<td>1,943.5</td>
<td>5.28</td>
</tr>
</tbody>
</table>

Table 5.12: OpenSSL ECDH Operations over NIST curves
Table 5.13 demonstrates the impact of the improved point multiplication on another actual cryptographic operation - the signature creation and verification process. Therefore, the functions ECDSA\_sign and ECDSA\_verify have been measured. We perceive an average speedup of factor 4.9 and 6.2, respectively.

<table>
<thead>
<tr>
<th>Binary Curve</th>
<th>OpenSSL-1.0.1e</th>
<th>Patched OpenSSL</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign</td>
<td>Verify</td>
<td>Sign</td>
</tr>
<tr>
<td>sect163k1</td>
<td>929,081</td>
<td>1,667,026</td>
<td>181,832</td>
</tr>
<tr>
<td>sect163r1</td>
<td>955,187</td>
<td>1,743,232</td>
<td>190,778</td>
</tr>
<tr>
<td>sect163r2</td>
<td>959,119</td>
<td>1,734,078</td>
<td>193,299</td>
</tr>
<tr>
<td>sect193r1</td>
<td>1,004,362</td>
<td>1,806,773</td>
<td>248,191</td>
</tr>
<tr>
<td>sect193r2</td>
<td>1,007,317</td>
<td>1,816,476</td>
<td>248,683</td>
</tr>
<tr>
<td>sect233k1</td>
<td>1,161,852</td>
<td>2,125,398</td>
<td>262,615</td>
</tr>
<tr>
<td>sect233r1</td>
<td>1,197,683</td>
<td>2,226,927</td>
<td>289,039</td>
</tr>
<tr>
<td>sect239k1</td>
<td>1,194,324</td>
<td>2,210,181</td>
<td>321,347</td>
</tr>
<tr>
<td>sect239r1</td>
<td>2,097,257</td>
<td>4,006,091</td>
<td>366,470</td>
</tr>
<tr>
<td>sect283k1</td>
<td>2,227,001</td>
<td>4,241,617</td>
<td>404,091</td>
</tr>
<tr>
<td>sect283r1</td>
<td>3,505,242</td>
<td>6,801,755</td>
<td>702,767</td>
</tr>
<tr>
<td>sect409k1</td>
<td>3,776,632</td>
<td>7,322,093</td>
<td>793,073</td>
</tr>
<tr>
<td>sect409r1</td>
<td>8,018,930</td>
<td>15,635,228</td>
<td>1,365,881</td>
</tr>
<tr>
<td>sect571k1</td>
<td>8,745,813</td>
<td>17,133,137</td>
<td>1,542,300</td>
</tr>
<tr>
<td>sect571r1</td>
<td>181,832</td>
<td>215,389</td>
<td>5.11</td>
</tr>
<tr>
<td></td>
<td>190,778</td>
<td>238,232</td>
<td>5.01</td>
</tr>
<tr>
<td></td>
<td>193,299</td>
<td>236,150</td>
<td>4.96</td>
</tr>
<tr>
<td></td>
<td>248,191</td>
<td>335,012</td>
<td>4.05</td>
</tr>
<tr>
<td></td>
<td>248,683</td>
<td>334,215</td>
<td>4.05</td>
</tr>
<tr>
<td></td>
<td>262,615</td>
<td>362,315</td>
<td>4.42</td>
</tr>
<tr>
<td></td>
<td>289,039</td>
<td>405,330</td>
<td>4.14</td>
</tr>
<tr>
<td></td>
<td>321,347</td>
<td>482,566</td>
<td>3.72</td>
</tr>
<tr>
<td></td>
<td>366,470</td>
<td>557,568</td>
<td>5.72</td>
</tr>
<tr>
<td></td>
<td>404,091</td>
<td>628,606</td>
<td>5.51</td>
</tr>
<tr>
<td></td>
<td>702,767</td>
<td>1,181,176</td>
<td>4.99</td>
</tr>
<tr>
<td></td>
<td>793,073</td>
<td>1,359,816</td>
<td>4.76</td>
</tr>
<tr>
<td></td>
<td>1,365,881</td>
<td>2,461,485</td>
<td>5.87</td>
</tr>
<tr>
<td></td>
<td>1,542,300</td>
<td>2,816,402</td>
<td>5.67</td>
</tr>
</tbody>
</table>

Table 5.13.: OpenSSL ECDSA Sign/Verify in $GF(2^m)$

Contrary to the results in the previous table, the OpenSSL speed tool has been used to create the numbers in the table below. Since the bit length for the ECDSA has been fixed before the point multiplication as a patch for the timing attack [17], the resulting scalar $k$ is not in the interval $[1,n]$ anymore and is either one or two bits longer (but constant) than the original scalar (depending on the form of the corresponding order $n$), which slightly increases the amount of cycles for the patched version. Nevertheless, the results show an even higher average speedup factor of 5.3 and 5.7, mainly caused by the positive effect of the absence of the curve over $GF(2^{239})$ with the slow reduction.

<table>
<thead>
<tr>
<th>Binary Curve</th>
<th>OpenSSL-1.0.1e</th>
<th>Patched OpenSSL</th>
<th>Speedup Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sign/s</td>
<td>Verify/s</td>
<td>Sign/s</td>
</tr>
<tr>
<td>nistk163</td>
<td>3,821.3</td>
<td>1,574.7</td>
<td>21,566.9</td>
</tr>
<tr>
<td>nistk233</td>
<td>1,904.0</td>
<td>1,213.9</td>
<td>10,390.5</td>
</tr>
<tr>
<td>nistk283</td>
<td>1,257.1</td>
<td>637.8</td>
<td>6,725.1</td>
</tr>
<tr>
<td>nistk409</td>
<td>537.8</td>
<td>370.5</td>
<td>3,147.4</td>
</tr>
<tr>
<td>nistk571</td>
<td>246.1</td>
<td>158.9</td>
<td>1,566.9</td>
</tr>
<tr>
<td>nistb163</td>
<td>3,837.6</td>
<td>1,514.0</td>
<td>16,236.7</td>
</tr>
<tr>
<td>nistb233</td>
<td>1,921.1</td>
<td>1,160.2</td>
<td>9,420.1</td>
</tr>
<tr>
<td>nistb283</td>
<td>1,250.4</td>
<td>599.6</td>
<td>5,957.3</td>
</tr>
<tr>
<td>nistb409</td>
<td>534.9</td>
<td>341.6</td>
<td>2,761.3</td>
</tr>
<tr>
<td>nistb571</td>
<td>246.1</td>
<td>145.0</td>
<td>1,371.3</td>
</tr>
</tbody>
</table>

Table 5.14.: OpenSSL ECDSA Operations in $GF(2^m)$
5.3. Comparison

We finally want to compare our results with other works. As most works about the implementation of binary elliptic curves with the use of vector instructions are aiming to break speed records and thus often exploit algebraic properties of certain fields, we split our comparison into two parts. The first part deals with the fast implementation of the binary field arithmetic, the second part compares point multiplication techniques.

5.3.1. Comparison - Binary Field Arithmetic

For the multiplication in $GF(2^{283})$, [9] reports with 142 cycles almost the same results for a Sandy Bridge architecture, [60] reports 100 cycles in $GF(2^{233})$ and 270 cycles in $GF(2^{409})$ on a Sandy Bridge. In this work we achieved similar results for field multiplication in $GF(2^{233})$ and beat the reported number for $GF(2^{409})$ by 16 cycles on the same architecture.

Compared to [59], our results for the unreduced multiplication are better for all fields greater than 233 bits. We also beat the reported numbers in $GF(2^{233})$ (115) and $GF(2^{409})$ (288) for the field multiplication by 14% and 17%. Unfortunately, we found no numbers to compare with for the addition, but since addition is very cheap and straightforward we do not expect any great differences anyway.

The numbers for the squaring follow that tendency. Our implementation is as fast as [60] for $GF(2^{233})$ and 35% faster in $GF(2^{409})$, 7% slower than reported in [9] for $GF(2^{283})$ (faster reduction) and up to 72% faster ($GF(2^{571})$) than claimed in [8]. Since we are not able to compare the implementations, our guess is that these differences are caused by the field reduction impact. For the field reduction itself we do not possess any numbers except the 20 cycles for $GF(2^{283})$, reported in [9]. This result is faster than the 25 cycles we report for this field using the same technique.

The same work uses multi-squaring and performs a field inversion in 3,286 cycles for $GF(2^{283})$, compared to 9,808 cycles in our implementation. On the other hand, [8] reports 13,608 cycles for the inversion with the Extended Euclidean Algorithm, but they state that their implementation was not optimized and uses a slow software multiplication (without hardware multiplier).

5.3.2. Comparison - Point Multiplication

Due to our preliminaries, this work aims for provision of a generic point multiplication for the well known NIST/SECG curves over the binary field. We want to briefly sum up the limitations of our implementation in comparison to other fast implementations. This work does not use

- precomputation schemes (such as wNAF, τNAF),
- point halving,
- special fields with beneficial properties (such as $F_{q^2}$),
- frobenius automorphism.

The use of precomputation schemes often require expensive operations to add resistance against side channel attacks (or relaxing assumptions), but are much faster in general. However, with our tight requirements concerning the side channel resistance such as completely data independent implementation, we decided to leave precomputation out, which also has the benefit of low memory consumption. Generally, point halving is faster than point doubling, however, our target fields are not optimal for that. Finally, the use of other algebraic improvements such as the Frobenius Morphism are not applicable since our implementation aims to work for both Koblitz and Random curves.

In the following table we show results for implementations of random point multiplication for binary elliptic curves in NIST/SEC fields. All numbers are given in $10^3$ cycles on a single core.

This work yields very good results for random curves and wins against current state-of-the-art implementations over the NIST fields, even with the costly side channel countermeasures. For the random
Aranha et al. [8] tested Koblitz curves using the Core i7-860 processor and achieved a result of 1656 cycles for the NIST field $GF(2^{233})$. Taverne et al. [50] also used a Koblitz curve with the Core i7 (SNB) and reported a $\tau$-NAF, $\tau$ & add method, achieving 264 cycles for $GF(2^{283})$ and 738 cycles for $GF(2^{409})$. Aranha et al. [9] used the Core i7 (SNB) for Koblitz curves and reported 099 cycles for $GF(2^{409})$.

This work uses a Koblitz curve over the NIST field $GF(2^{233})$ with a Core i5 (IVB) processor and reports 1095 cycles for LD-Montgomery. Aranha et al. [8] used a random curve over the Core i7-860 with 793 cycles for $GF(2^{233})$ and 4440 cycles for $GF(2^{409})$.

Table 5.15.: Unknown Point Multiplication over NIST Curves

<table>
<thead>
<tr>
<th>Work</th>
<th>Curve</th>
<th>SCP</th>
<th>CPU</th>
<th>Method</th>
<th>$GF(2^{233})$</th>
<th>$GF(2^{283})$</th>
<th>$GF(2^{409})$</th>
<th>$GF(2^{571})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aranha et al. [8]</td>
<td>Koblitz</td>
<td>no</td>
<td>Core i7-860</td>
<td>4-TNAF</td>
<td>-</td>
<td>386</td>
<td>-</td>
<td>1656</td>
</tr>
<tr>
<td>Taverne et al. [50]</td>
<td>Koblitz</td>
<td>no</td>
<td>Core i7 (SNB)</td>
<td>5-τNAF, τ &amp; add</td>
<td>068</td>
<td>-</td>
<td>264</td>
<td>-</td>
</tr>
<tr>
<td>Aranha et al. [9]</td>
<td>Koblitz</td>
<td>no</td>
<td>Core i7 (SNB)</td>
<td>5-τNAF, $\lfloor 2^{283}/2 \rfloor$</td>
<td>-</td>
<td>099</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>This work</td>
<td>Koblitz</td>
<td>yes</td>
<td>Core i5 (IVB)</td>
<td>LD-Montgomery</td>
<td>128</td>
<td>217</td>
<td>511</td>
<td>1095</td>
</tr>
<tr>
<td>Aranha et al. [8]</td>
<td>Random</td>
<td>no</td>
<td>Core i7-860</td>
<td>LD-Montgomery</td>
<td>-</td>
<td>793</td>
<td>-</td>
<td>4440</td>
</tr>
<tr>
<td>Taverne et al. [50]</td>
<td>Random</td>
<td>no</td>
<td>Core i7 (SNB)</td>
<td>4-ωNAF, double &amp; add</td>
<td>180</td>
<td>-</td>
<td>738</td>
<td>-</td>
</tr>
<tr>
<td>This work</td>
<td>Random</td>
<td>yes</td>
<td>Core i5 (IVB)</td>
<td>LD-Montgomery</td>
<td>148</td>
<td>254</td>
<td>599</td>
<td>1271</td>
</tr>
</tbody>
</table>

Anyhow, the techniques for Koblitz curves we briefly summed up above lead to much faster, but in this case unprotected implementations. Although our implementation beats the numbers presented in [8] by 77% and 51%, [60] achieve results which are 53% faster, whilst [9] is even as twice as fast. Anyhow, to the present day, we are not aware of any implementation for Koblitz curves over NIST fields, protecting the point multiplication in a similar way as presented in this work but also exploiting the algebraic properties.

For further comparison, we also want to show the latest numbers of implementations at the 128 bit security level, using curves with special algebraic properties.

<table>
<thead>
<tr>
<th>Work</th>
<th>Curve</th>
<th>SCP</th>
<th>CPU</th>
<th>Method</th>
<th>kP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamburg [31]</td>
<td>Montgomery</td>
<td>yes</td>
<td>Core i7 (SB)</td>
<td>Montgomery Ladder</td>
<td>153</td>
</tr>
<tr>
<td>Longa/Sica [42]</td>
<td>GLS</td>
<td>yes</td>
<td>Core i7 (SB)</td>
<td>4-GLV, double &amp; add</td>
<td>137</td>
</tr>
<tr>
<td>Bos et al. [14]</td>
<td>Kummer</td>
<td>yes</td>
<td>Core i7 (IB)</td>
<td>Montgomery Ladder</td>
<td>117</td>
</tr>
<tr>
<td>Oliviera et al. [52]</td>
<td>GLS</td>
<td>yes</td>
<td>Xeon/Core i5 (IB)</td>
<td>2-GLV, double &amp; add((\lambda))</td>
<td>115</td>
</tr>
</tbody>
</table>

Table 5.16.: Unknown Point Multiplication over Special Curves

Side channel protection was claimed by the authors and the understanding of side channel protection may differ from case to case. Please also note that the numbers for scalar multiplication are for unknown points only, fixed point multiplication usually gains even more from precomputation schemes. Besides that, all measurements have been reported for single core, running on multiple cores may also tremendously decrease the amount of cycles, whereas some methods gain from that more than others.

5.3.3. Further Work

Further, the binary field multiplication can be improved by applying more sophisticated non-recursive forms of the Karatsuba algorithm. In addition, for small amounts of 64 bit word multiplication the schoolbook method could win against the Karatsuba on future architectures, caused by the greater influence of the instructions for recombination in contrary to the faster \texttt{pclmulqdq}. The new AVX2 instruction set, introduced with the Haswell architecture in June 2013, is extending AVX with 256 bit vector instructions for integers and thus promises even better numbers for binary field operations. In this context, the use of the proposed reduction using random multipliers needs to be further evaluated as well as the usage of mixed approaches with the shift & add scheme.

To address the matter of not having a fast and protected implementation for Koblitz curves over NIST fields, exploiting the algebraic properties of this fields, we suggest further research on how to apply the well known techniques together with the side channel countermeasures we proposed earlier.

---

3Side Channel Protection
6. Conclusion

In this work, we proposed a fast, constant time implementation for binary field arithmetic for binary elliptic curves at a security level for more than 80 bits, standardized by NIST and SECG.

We analyzed major implementation threats for server environments and secured the implementation by applying countermeasures against common and dangerous side channel threats such as cache and remote timing attacks. Over the last years, several publications have pointed out that these side channel attacks are more than a theoretical but a serious menace. To address this issue, we provide a constant time implementation of the point multiplication without any data dependent branches and an oblivious memory access pattern.

Additionally, we introduced an improved version for the well known Montgomery multiplication with enhancements on the arithmetic level and demonstrated how to integrate this improvements into the OpenSSL library and showed its significant performance impact. These improvements are available on all 64 bit architectures implementing SSE4 and/or AVX and the pclmulqdq instruction, introduced together with the AES instruction set on the Intel Westmere architecture in March 2010. The implementation has been contributed to OpenSSL as patch for version 1.0.1e and is available for download.

We further showed that this impact is not limited to the point multiplication itself but improves the servers performance regarding ECDH operations and signature based cryptography. This further leads to a major performance improvement for servers and clients conducting TLS handshakes. This is an essential enhancement especially for server environments, which are often required to conduct a big number of handshakes simultaneously.
A. Bibliography


B. OpenSSL Patch

diff --urN openssl-orig/crypto/bn/bn_gf2m.c openssl-work/crypto/bn/bn_gf2m.c
--- openssl-orig/crypto/bn/bn_gf2m.c 2013-02-11 16:26:04.000000000 +0100
+++ openssl-work/crypto/bn/bn_gf2m.c 2013-05-12 05:29:49.875830999 +0200
@@ -1110,4 +1110,223 @@

  return 1;
 }
+#ifdef OPENSSL_FAST_EC2M

+/*
+ * Functions for constant time BIGNUM implementations, kindly ignoring zero padding.
+ */
+
+/* Expand BIGNUM `a` to an element of constant size and set it zero.
+ * NOT constant time.
+ */
+int BN_GF2m_const_init(BIGNUM *a, int word_size)
+{
+  int i;
+  BN_ULONG *ap;
+  /* Expand a */
+  if(bn_wexpand(a, word_size) == NULL) return 0;
+  a->top = word_size;
+  /* Set a to zero */
+  ap = a->d;
+  for(i=0; i < word_size; i++) ap[i] = 0;
+  return 1;
+}
+
+/* Copy the value of `b` to `a` without destroying the length of both operands.
+ * `a` must have at least the same number of words as `b` and initialized with
+ * BN_GF2m_const_init to assert leading zero padding. Only constant time if
+ * length of `a` equals length of `b`.
+ */
+int BN_GF2m_copy(BIGNUM *a, const BIGNUM *b)
+{
+  int i, ret=0;
+  BN_ULONG *ap,*bp;
+  if (b->top > a->top) goto err;
+  ap = a->d;
+  bp = b->d;
+  for(i=0; i < b->top; i++) ap[i] = bp[i];
+  ret = 1;
+err:
+  return ret;
+}
+
+/* Copy the value of `b` to `a` for numbers with the same word length. */
+int BN_GF2m_const_copy(BIGNUM *a, const BIGNUM *b)
+{
+  int i, ret=0;
+  BN_ULONG *ap,*bp;
+  if (a->top != b->top) goto err;
+  ap = a->d;
+  bp = b->d;
+  for(i=0; i < b->top; i++) ap[i] = bp[i];
+  ret = 1;
+err:
+  return ret;
+}
/* Computes \( z = a + b \) for integers with constant number of words.
  * Does not correct zero padding. 'a' can be 'a' and 'b' can be 'b'.
  * Returns error if carry bit occurs at the topmost word.
  */
int BN_GF2m_const_int_add (BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
  int ret = 0;
  BN_ULONG *ap, *bp, *zp;

  /* Assert constant size of elements */
  if (a->top < 1) goto err;
  if (a->top != b->top) goto err;
  if (z->top != b->top) goto err;

  ap = a->d;
  bp = b->d;
  zp = z->d;

  /* Addition */
  if (bn_add_words (zp, bp, ap, a->top)) goto err;

  ret = 1;

  err :
  return ret;
}

/* Does constant addition in GF2m without correcting zero padding.
  * Computes \( z = a \land b \). */
int BN_GF2m_const_add (BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
  int i, ret = 0;
  BN_ULONG *ap, *bp, *zp;

  /* Assert constant size of elements */
  if (a->top != b->top) goto err;
  if (z->top != b->top) goto err;

  ap = a->d;
  bp = b->d;
  zp = z->d;

  for (i = 0; i < a->top; i++)
  {
    zp[i] = ap[i] ^ bp[i];
  }

  ret = 1;

  err :
  return ret;
}

/* Compare value of 'a' to zero: Returns 1 if a equals zero. */
int BN_GF2m_const_cmp_zero (const BIGNUM *a)
{
  int i, ret = 1;
  BN_ULONG *ap;

  ap = a->d;
  for (i = 0; i < a->top; i++)
  {
    if (ap[i] != 0) ret = 0;
  }

  return ret;
}

/* Compare value of 'a' to one: Returns 0 if a equals one. */
int BN_GF2m_const_cmp_one (const BIGNUM *a)
int i, ret=0;
BN_ULONG *ap;

ap = a->d;

if (ap[0] != 1) ret = 1;
for (i=1; i < a->top; i++)
{
    if (ap[i] != 0) ret = 1;
}
return ret;
}

/* Compare two field elements 'a' and 'b'. */
/* Returns */
/* 0 if a equals b */
/* 1 if a not equals b. */
int BN_GF2m_const_cmp_eq (const BIGNUM *a, const BIGNUM *b)
{
    int i, ret=0;
    BN_ULONG *ap,*bp;

    if (a->top != b->top) goto err ;
    ap=a->d;
    bp=b->d;
    for (i=a->top-1; i>=0; i--)
    {
        if (ap[i] != bp[i]) ret = 1;
    }
err :
return ret;
}

/* Set value of BIGNUM 'a' to one without destroying its length. */
/* 'a' must have at least one word. */
int BN_GF2m_const_setone (BIGNUM *a)
{
    int i, ret=0;
    BN_ULONG *ap;

    if (a->top < 1) goto err ;
    ap = a->d;
    ap[0] = 1;
    for (i=1; i < a->top; i++) ap[i] = 0;
    ret = 1;
err :
return ret;
}

/* Set all words of BIGNUM 'a' to 'b' without destroying its length. */
/* 'a' must have at least one word. */
int BN_GF2m_const_setword (BIGNUM *a, BN_ULONG b)
{
    int i, ret=0;
    BN_ULONG *ap;

    if (a->top < 1) goto err ;
    ap = a->d;
    for (i=0; i < a->top; i++) ap[i] = b;
    ret = 1;
err :
return ret;
}
#endif
/* crypto/bn/bn_gf2m_xmm.c */

/* Written by Manuel Bluhm for the OpenSSL project. */

/* Copyright (c) 1998 - 2011 The OpenSSL Project. All rights reserved. */

/* Redistributions of source code must retain the above copyright */
/* notice, this list of conditions and the following disclaimer. */

/* Redistributions in binary form must reproduce the above copyright */
/* notice, this list of conditions and the following disclaimer in */
/* the documentation and/or other materials provided with the */
/* distribution. */

/* All advertising materials mentioning features or use of this */
/* software must display the following acknowledgment: */
/* "This product includes software developed by the OpenSSL Project */
/* for use in the OpenSSL Toolkit. (http://www.openssl.org/)" */

/* The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to */
/* endorse or promote products derived from this software without */
/* prior written permission. For written permission, please contact */
/* openssl-core@openssl.org. */

/* Products derived from this software may not be called "OpenSSL" */
/* nor may "OpenSSL" appear in their names without prior written */
/* permission of the OpenSSL Project. */

/* Redistributions of any form whatsoever must retain the following */
/* acknowledgment: */
/* "This product includes software developed by the OpenSSL Project */
/* for use in the OpenSSL Toolkit (http://www.openssl.org/)" */

/* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT "AS IS" AND ANY */
/* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE */
/* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR */
/* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR */
/* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, */
/* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT */
/* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; */
/* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) */
/* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, */
/* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) */
/* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED */
/* OF THE POSSIBILITY OF SUCH DAMAGE. */

/* This product includes cryptographic software written by Eric Young */
/* (eay@cryptsoft.com). This product includes software written by Tim */
/* Hudson (tjh@cryptsoft.com). */

# ifndef OPENSSL_NO_EC2M

# define LOOP_MACRO for squarings in inversion */
#define LOOP(function, loops) for(i=0; i < loops; i++) {function;}

# endif

#endif

// no newline at end of file

/* Macros for SSE and PCLMUL compiler intrinsics */
/* Load, store and extraction */
#define LOAD_64 _mm_loadl_epi64
#define LOAD128 _mm_load_si128
#define STORE_64 _mm_storel_epi64
#define STORE128 _mm_store_si128
#define SET64 _mm_set_epi64x
#define GET64 _mm_extract_epi64
+
/* Arithmetic */
#define PCLMUL _mm_clmulepi64_si128
#define SHUFFLE _mm_shuffle_epi8
#define XOR _mm_xor_si128
#define AND _mm_and_si128
#define NAND _mm_andnot_si128
#define OR _mm_or_si128
#define NAND _mm_ornot_si128
#define SHR _mm_srli_epi64
#define SHL _mm_slli_epi64
#define SHL128 _mm_slli_si128
#define SHR128 _mm_srli_si128
+
/* Memory alignment */
#define ALIGNR _mm_alignr_epi8
#define MOVE64 _mm_move_epi64
#define UNPACKLO8 _mm_unpacklo_epi8
#define UNPACKHI8 _mm_unpackhi_epi8
#define UNPACKLO64 _mm_unpacklo_epi64
#define UNPACKHI64 _mm_unpackhi_epi64
+
#define ZERO _mm_setzero_si128()
+
+ /* BN <-> XMM CONVERSATIONS */
+ /* Functions to convert between XMM and BN representation. */
+ */
+
static inline void BN_to_XMM_1term ( __m128i z[1], BN_ULONG *a)
{  
z[0] = LOAD_64 (( __m128i *) (a));
}
+
static inline void BN_to_XMM_2term ( __m128i z[2], BN_ULONG *a)
{  
z[0] = LOAD128 (( __m128i *) (a));
    z[1] = LOAD_64 (( __m128i *) (a + 2));
}
+
static inline void BN_to_XMM_3term ( __m128i z[3], BN_ULONG *a)
{  
z[0] = LOAD128 (( __m128i *) (a));
    z[1] = LOAD128 (( __m128i *) (a + 2));
    z[2] = LOAD_64 (( __m128i *) (a + 4));
}
+
static inline void BN_to_XMM_4term ( __m128i z[3], BN_ULONG *a)
{  
z[0] = LOAD128 (( __m128i *) (a));
    z[1] = LOAD128 (( __m128i *) (a + 2));
    z[2] = LOAD128 (( __m128i *) (a + 4));
    z[3] = LOAD128 (( __m128i *) (a + 6));
}
+
static inline void BN_to_XMM_5term ( __m128i z[4], BN_ULONG *a)
{  
z[0] = LOAD128 (( __m128i *) (a));
    z[1] = LOAD128 (( __m128i *) (a + 2));
    z[2] = LOAD128 (( __m128i *) (a + 4));
    z[3] = LOAD128 (( __m128i *) (a + 6));
    z[4] = LOAD128 (( __m128i *) (a + 8));
}
+
static inline void BN_to_XMM_6term ( __m128i z[5], BN_ULONG *a)
{  
z[0] = LOAD128 (( __m128i *) (a));
    z[1] = LOAD128 (( __m128i *) (a + 2));
    z[2] = LOAD128 (( __m128i *) (a + 4));
    z[3] = LOAD128 (( __m128i *) (a + 6));
    z[4] = LOAD128 (( __m128i *) (a + 8));
    z[5] = LOAD128 (( __m128i *) (a + 10));
}
static inline void BN_to_XMM_7term(__m128i z[4], BN_ULONG *a)
{
  z[0] = LOAD128((__m128i *) (a));
  z[1] = LOAD128((__m128i *) (a + 2));
  z[2] = LOAD128((__m128i *) (a + 4));
  z[3] = LOAD_64((__m128i *) (a + 6));
}

static inline void BN_to_XMM_8term(__m128i z[4], BN_ULONG *a)
{
  z[0] = LOAD128((__m128i *) (a));
  z[1] = LOAD128((__m128i *) (a + 2));
  z[2] = LOAD128((__m128i *) (a + 4));
  z[3] = LOAD128((__m128i *) (a + 6));
}

static inline void BN_to_XMM_9term(__m128i z[5], BN_ULONG *a)
{
  z[0] = LOAD128((__m128i *) (a));
  z[1] = LOAD128((__m128i *) (a + 2));
  z[2] = LOAD128((__m128i *) (a + 4));
  z[3] = LOAD128((__m128i *) (a + 6));
  z[4] = LOAD_64((__m128i *) (a + 8));
}

static inline void XMM_to_BN_1term(BN_ULONG *z, __m128i a[1])
{
  STORE_64((__m128i *) (z), a[0]);
}

static inline void XMM_to_BN_2term(BN_ULONG *z, __m128i a[1])
{
  STORE128((__m128i *) (z), a[0]);
}

static inline void XMM_to_BN_3term(BN_ULONG *z, __m128i a[2])
{
  STORE128((__m128i *) (z), a[0]);
  STORE_64((__m128i *) (z + 2), a[1]);
}

static inline void XMM_to_BN_4term(BN_ULONG *z, __m128i a[2])
{
  STORE128((__m128i *) (z), a[0]);
  STORE128((__m128i *) (z + 2), a[1]);
}

static inline void XMM_to_BN_5term(BN_ULONG *z, __m128i a[3])
{
  STORE128((__m128i *) (z), a[0]);
  STORE128((__m128i *) (z + 2), a[1]);
  STORE_64((__m128i *) (z + 4), a[2]);
}

static inline void XMM_to_BN_6term(BN_ULONG *z, __m128i a[3])
{
  STORE128((__m128i *) (z), a[0]);
  STORE128((__m128i *) (z + 2), a[1]);
  STORE128((__m128i *) (z + 4), a[2]);
}

static inline void XMM_to_BN_7term(BN_ULONG *z, __m128i a[4])
{
  STORE128((__m128i *) (z), a[0]);
  STORE128((__m128i *) (z + 2), a[1]);
  STORE128((__m128i *) (z + 4), a[2]);
  STORE_64((__m128i *) (z + 6), a[3]);
}

static inline void XMM_to_BN_8term(BN_ULONG *z, __m128i a[4])
{
  STORE128((__m128i *) (z), a[0]);
  STORE128((__m128i *) (z + 2), a[1]);
  STORE128((__m128i *) (z + 4), a[2]);
  STORE128((__m128i *) (z + 6), a[3]);
}

static inline void XMM_to_BN_9term(BN_ULONG *z, __m128i a[5])
{

+ STORE128((__m128i *)(z), a[0]);
+ STORE128((__m128i *)(z + 2), a[1]);
+ STORE128((__m128i *)(z + 4), a[2]);
+ STORE128((__m128i *)(z + 6), a[3]);
+ STORE_64((__m128i *)(z + 8), a[4]);
+
+ /*

+ ****************************************************************************
+ * XMM COPY
+ *
+ * Functions to copy numbers in XMM representation.
+ *
+ ****************************************************************************
+ */
+
+ static inline void XMM_GF2m_copy_2term(__m128i z[2], const __m128i a[2])
+ {
+   z[0] = a[0];
+   z[1] = a[1];
+ }
+
+ static inline void XMM_GF2m_copy_3term(__m128i z[3], const __m128i a[3])
+ {
+   z[0] = a[0];
+   z[1] = a[1];
+   z[2] = a[2];
+ }
+
+ static inline void XMM_GF2m_copy_4term(__m128i z[4], const __m128i a[4])
+ {
+   z[0] = a[0];
+   z[1] = a[1];
+   z[2] = a[2];
+   z[3] = a[3];
+ }
+
+ static inline void XMM_GF2m_copy_5term(__m128i z[5], const __m128i a[5])
+ {
+   z[0] = a[0];
+   z[1] = a[1];
+   z[2] = a[2];
+   z[3] = a[3];
+   z[4] = a[4];
+ }
+
+ /*

+ ****************************************************************************
+ * XMM ADDITION
+ *
+ * This section implements addition in GF(2^m) in XMM registers.
+ *
+ ****************************************************************************
+ */
+
+ static inline void XMM_GF2m_add_2term(__m128i z[2], __m128i a[2], __m128i b[2])
+ {
+   z[0] = XOR(a[0], b[0]);
+   z[1] = XOR(a[1], b[1]);
+ }
+
+ static inline void XMM_GF2m_add_3term(__m128i z[3], __m128i a[3], __m128i b[3])
+ {
+   z[0] = XOR(a[0], b[0]);
+   z[1] = XOR(a[1], b[1]);
+   z[2] = XOR(a[2], b[2]);
+ }
+
+ static inline void XMM_GF2m_add_4term(__m128i z[4], __m128i a[4], __m128i b[4])
+ {
+   z[0] = XOR(a[0], b[0]);
+   z[1] = XOR(a[1], b[1]);
+   z[2] = XOR(a[2], b[2]);
+   z[3] = XOR(a[3], b[3]);
+ }
+ /*
+static inline void XMM_GF2m_add_5term ( __m128i z[5], __m128i a[5], __m128i b[5])
+ {
  + z[0] = XOR(a[0], b[0]);
  + z[1] = XOR(a[1], b[1]);
  + z[2] = XOR(a[2], b[2]);
  + z[3] = XOR(a[3], b[3]);
  + z[4] = XOR(a[4], b[4]);
+ }

+static inline void XMM_GF2m_add_7term ( __m128i z[7], __m128i a[7], __m128i b[7])
+ {
  + z[0] = XOR(a[0], b[0]);
  + z[1] = XOR(a[1], b[1]);
  + z[2] = XOR(a[2], b[2]);
  + z[3] = XOR(a[3], b[3]);
  + z[4] = XOR(a[4], b[4]);
  + z[5] = XOR(a[5], b[5]);
  + z[6] = XOR(a[6], b[6]);
+ }

+static inline void XMM_GF2m_add_9term ( __m128i z[9], __m128i a[9], __m128i b[9])
+ {
  + z[0] = XOR(a[0], b[0]);
  + z[1] = XOR(a[1], b[1]);
  + z[2] = XOR(a[2], b[2]);
  + z[3] = XOR(a[3], b[3]);
  + z[4] = XOR(a[4], b[4]);
  + z[5] = XOR(a[5], b[5]);
  + z[6] = XOR(a[6], b[6]);
  + z[7] = XOR(a[7], b[7]);
  + z[8] = XOR(a[8], b[8]);
+ }

+/

="/*********************************************************************************************/
+ /* XMM VEILING */
+ /* */
+ /* This section implements the data veiling for the Montgomery point multiplication. */
+ + /*******************************************************************************************/
+ +static inline void XMM_GF2m_mask_2term ( __m128i z[2], __m128i y[2],
+ __m128i a[2], __m128i b[2], __m128i mask)
+ {
  + z[0] = AND( mask, a[0]);
  + z[1] = AND( mask, a[1]);
  + y[0] = NAND( mask, b[0]);
  + y[1] = NAND( mask, b[1]);
+ }

+static inline void XMM_GF2m_mask_3term ( __m128i z[3], __m128i y[3],
+ __m128i a[3], __m128i b[3], __m128i mask)
+ {
  + z[0] = AND( mask, a[0]);
  + z[1] = AND( mask, a[1]);
  + z[2] = AND( mask, a[2]);
  + y[0] = NAND( mask, b[0]);
  + y[1] = NAND( mask, b[1]);
  + y[2] = NAND( mask, b[2]);
+ }

+static inline void XMM_GF2m_mask_4term ( __m128i z[4], __m128i y[4],
+ __m128i a[4], __m128i b[4], __m128i mask)
+ {
  + z[0] = AND( mask, a[0]);
  + z[1] = AND( mask, a[1]);
  + z[2] = AND( mask, a[2]);
  + z[3] = AND( mask, a[3]);
  + y[0] = NAND( mask, b[0]);
  + y[1] = NAND( mask, b[1]);
  + y[2] = NAND( mask, b[2]);
  + y[3] = NAND( mask, b[3]);
+ }

+static inline void XMM_GF2m_mask_5term ( __m128i z[4], __m128i y[4],
+ __m128i a[4], __m128i b[4], __m128i mask)
+ {
  + z[0] = AND( mask, a[0]);
+ z[0] = AND( mask, a[0]);
+ z[1] = AND( mask, a[1]);
+ z[2] = AND( mask, a[2]);
+ z[3] = AND( mask, a[3]);
+ z[4] = AND( mask, a[4]);
+ y[0] = NAND( mask, b[0]);
+ y[1] = NAND( mask, b[1]);
+ y[2] = NAND( mask, b[2]);
+ y[3] = NAND( mask, b[3]);
+ y[4] = NAND( mask, b[4]);
+
+ static inline void XMM_GF2m_veil_2term ( __m128i x1[2], __m128i z1[2], __m128i x2[2], __m128i z2[2],
+                                           __m128i tx1[2], __m128i tz1[2], __m128i tx2[2], __m128i tz2[2], BN_ULONG k)
+ {
+   __m128i mask, t1[2], t2[2];
+   BN_ULONG mk;
+   mk = (0 - k);
+   mask = SET64( mk, mk);
+   XMM_GF2m_mask_2term(t1, t2, tx1, tx2, mask);
+   XMM_GF2m_add_2term(x1, t1, t2);
+   XMM_GF2m_mask_2term(t1, t2, tx2, tx1, mask);
+   XMM_GF2m_add_2term(x2, t1, t2);
+   XMM_GF2m_mask_2term(t1, t2, tz1, tz2, mask);
+   XMM_GF2m_add_2term(z1, t1, t2);
+   XMM_GF2m_mask_2term(t1, t2, tz2, tz1, mask);
+   XMM_GF2m_add_2term(z2, t1, t2);
+ }
+
+ static inline void XMM_GF2m_veil_3term ( __m128i x1[3], __m128i z1[3], __m128i x2[3], __m128i z2[3],
+                                           __m128i tx1[3], __m128i tz1[3], __m128i tx2[3], __m128i tz2[3], BN_ULONG k)
+ {
+   __m128i mask, t1[3], t2[3];
+   BN_ULONG mk;
+   mk = (0 - k);
+   mask = SET64( mk, mk);
+   XMM_GF2m_mask_3term(t1, t2, tx1, tx2, mask);
+   XMM_GF2m_add_3term(x1, t1, t2);
+   XMM_GF2m_mask_3term(t1, t2, tx2, tx1, mask);
+   XMM_GF2m_add_3term(x2, t1, t2);
+   XMM_GF2m_mask_3term(t1, t2, tz1, tz2, mask);
+   XMM_GF2m_add_3term(z1, t1, t2);
+   XMM_GF2m_mask_3term(t1, t2, tz2, tz1, mask);
+   XMM_GF2m_add_3term(z2, t1, t2);
+ }
+
+ static inline void XMM_GF2m_veil_4term ( __m128i x1[4], __m128i z1[4], __m128i x2[4], __m128i z2[4],
+                                          __m128i tx1[4], __m128i tz1[4], __m128i tx2[4], __m128i tz2[4], BN_ULONG k)
+ {
+   __m128i mask, t1[4], t2[4];
+   BN_ULONG mk;
+   mk = (0 - k);
+   mask = SET64( mk, mk);
+   XMM_GF2m_mask_4term(t1, t2, tx1, tx2, mask);
+   XMM_GF2m_add_4term(x1, t1, t2);
+   XMM_GF2m_mask_4term(t1, t2, tx2, tx1, mask);
+   XMM_GF2m_add_4term(x2, t1, t2);
+   XMM_GF2m_mask_4term(t1, t2, tz1, tz2, mask);
+   XMM_GF2m_add_4term(z1, t1, t2);
+   XMM_GF2m_mask_4term(t1, t2, tz2, tz1, mask);
+   XMM_GF2m_add_4term(z2, t1, t2);
+   XMM_GF2m_mask_4term(t1, t2, tz2, tz1, mask);
+   XMM_GF2m_add_4term(z2, t1, t2);
static inline void XMM_GF2m_veil_5term(__m128i x1[5], __m128i z1[5], __m128i x2[5], __m128i z2[5],
  __m128i tx1[5], __m128i tz1[5], __m128i tx2[5], __m128i tz2[5], BN_ULONGLONG k)
{
  __m128i mask, t1[5], t2[5];
  BN_ULONGLONG mk;
  mk = (0 - k);
  mask = SET64(mk, mk);
  XMM_GF2m_mask_5term(t1, t2, tx1, tx2, mask);
  XMM_GF2m_add_5term(x1, t1, t2);
  XMM_GF2m_mask_5term(t1, t2, tx2, tx1, mask);
  XMM_GF2m_add_5term(x2, t1, t2);
  XMM_GF2m_mask_5term(t1, t2, tz1, tz2, mask);
  XMM_GF2m_add_5term(z1, t1, t2);
  XMM_GF2m_mask_5term(t1, t2, tz2, tz1, mask);
  XMM_GF2m_add_5term(z2, t1, t2);
}

static inline void XMM_GF2m_mod_nist163(__m128i z[2], __m128i r[3])
{
  /* Init */
  __m128i x[5];
  x[0] = SHR(r[2], 35);
  x[1] = SHL(r[2], 29);
  x[3] = SHL128(r[2], 4);
  x[1] = XOR(x[1], x[3]);
  x[2] = SHR(r[2], 28);
  x[3] = SHL(r[2], 35);
  x[0] = XOR(x[0], x[2]);
  x[1] = XOR(x[1], x[3]);
  x[2] = SHL128(x[1], 8);
  x[1] = SHR128(x[1], 8);
  x[0] = XOR(x[0], x[2]);
  z[0] = XOR(r[0], x[2]);
  z[1] = XOR(r[1], x[0]);
  /* Clear top */
  x[1] = SET64(0xFFFFFFFFFFFFFFFF, 0xFFFFFFFF80000000);
  x[4] = AND(x[1], x[1]);
  z[1] = NAND(x[1], x[1]);
  x[0] = SHR(x[4], 36);
  x[1] = SHL(x[4], 29);
  x[2] = SHR128(x[4], 4);
  x[0] = XOR(x[0], x[2]);
  x[2] = SHR(x[4], 29);
static inline void XMM_GF2m_mod_sect193(__m128i z[2], __m128i r[4])
{
    /* Init */
    __m128i x[5];
    
    x[0] = SHL(r[3], 14);
    x[1] = SHR(r[3], 1);
    x[2] = SHL(r[3], 63);
    z[1] = XOR(r[1], x[2]);
    x[4] = XOR(x[0], x[1]);
    
    x[0] = SHR(r[2], 50);
    x[1] = SHL(r[2], 14);
    x[2] = SHR(r[2], 1);
    x[3] = SHL(r[2], 63);
    z[1] = XOR(z[1], x[0]);
    z[0] = XOR(r[0], x[3]);
    x[0] = XOR(x[1], x[2]);
    x[4] = ALIGNR(x[4], x[0], 8);
    z[1] = XOR(z[1], x[4]);
    
    /* Clear top */
    z[3] = SET64(0x0000000000000001, 0xFFFFFFFFFFFFFFFF);
    x[4] = NAND(x[3], z[1]);
    z[1] = AND(z[1], x[4]);
    z[1] = ALIGNR(z[4], x[0], 8);
    z[0] = XOR(z[0], x[1]);
    x[1] = SHR(x[4], 1);
    x[0] = ALIGNR(x[0], x[1], 8);
}

static inline void XMM_GF2m_mod_nist233(__m128i z[2], __m128i r[4])
{
    /* Init */
    __m128i x[6];
    
    x[0] = SHL(r[3], 33);
    x[1] = SHR(r[3], 31);
    x[2] = SHL(r[3], 23);
    x[3] = SHR(r[3], 41);
    
    x[4] = XOR(x[0], x[3]);
    x[3] = SHR128(x[4], 8);
    z[1] = XOR(r[1], x[2]);
    r[2] = XOR(r[2], x[1]);
    r[2] = XOR(r[2], x[3]);
    
    z[0] = SHR(r[2], 33);
    x[1] = SHR(r[2], 31);
    x[2] = SHL(r[2], 23);
    x[3] = SHR(r[2], 41);
    
    x[5] = XOR(x[0], x[3]);
    x[3] = ALIGNR(x[4], x[5], 8);
    z[0] = XOR(r[0], x[2]);
    z[1] = XOR(z[1], x[1]);
    z[1] = XOR(z[1], x[3]);
    
    /* Clear top */
    x[2] = SET64(0x0000000000000001, 0xFFFFFFFFFFFFFFFF);
    x[0] = NAND(x[2], z[1]);
```c
static inline void XMM_GF2m_mod_sect239(__m128i z[2], __m128i r[4]) {
    /* Init */
    BN_ULONG zz, w[8];
    STORE128((__m128i *) (w), r[0]);
    STORE128((__m128i *) (w + 2), r[1]);
    STORE128((__m128i *) (w + 4), r[2]);
    STORE128((__m128i *) (w + 6), r[3]);
    zz = w[7];
    w[6] ^= (zz >> 17);
    w[5] ^= (zz >> 47);
    w[4] ^= (zz >> 47);
    w[3] ^= (zz >> 17);
    zz = w[6];
    w[5] ^= (zz >> 17);
    w[4] ^= (zz >> 47);
    w[3] ^= (zz >> 47);
    w[2] ^= (zz >> 17);
    zz = w[5];
    w[4] ^= (zz >> 17);
    w[3] ^= (zz >> 47);
    w[2] ^= (zz >> 47);
    w[1] ^= (zz >> 17);
    zz = w[4];
    w[3] ^= (zz >> 17);
    w[2] ^= (zz >> 47);
    w[1] ^= (zz >> 47);
    w[0] ^= (zz >> 17);
    /* Clear top */
    zz = (w[3] >> 47);
    w[3] &= 0x00007FFFFFFFFF;
    w[0] ^= zz;
    w[2] ^= (zz << 30);
    z[0] = LOAD128((__m128i *) (w));
    z[1] = LOAD128((__m128i *) (w + 2));
}
```

```c
static inline void XMM_GF2m_mod_sect239(__m128i z[2], __m128i r[4]) {
    /* Init */
    BN_ULONG zz, w[8];
    STORE128((__m128i *) (w), r[0]);
    STORE128((__m128i *) (w + 2), r[1]);
    STORE128((__m128i *) (w + 4), r[2]);
    STORE128((__m128i *) (w + 6), r[3]);
    zz = w[7];
    w[6] ^= (zz >> 17);
    w[5] ^= (zz >> 47);
    w[4] ^= (zz >> 47);
    w[3] ^= (zz >> 17);
    zz = w[6];
    w[5] ^= (zz >> 17);
    w[4] ^= (zz >> 47);
    w[3] ^= (zz >> 47);
    w[2] ^= (zz >> 17);
    zz = w[5];
    w[4] ^= (zz >> 17);
    w[3] ^= (zz >> 47);
    w[2] ^= (zz >> 47);
    w[1] ^= (zz >> 17);
    zz = w[4];
    w[3] ^= (zz >> 17);
    w[2] ^= (zz >> 47);
    w[1] ^= (zz >> 47);
    w[0] ^= (zz >> 17);
    /* Clear top */
    zz = (w[3] >> 47);
    w[3] &= 0x00007FFFFFFFFF;
    w[0] ^= zz;
    w[2] ^= (zz << 30);
    z[0] = LOAD128((__m128i *) (w));
    z[1] = LOAD128((__m128i *) (w + 2));
}
```
r[3] = XOR(r[3], x[2]);
+ x[0] = ALIGNR(r[2], ZERO, 8);
+ x[2] = SML(r[2], 7);
+ r[2] = XOR(r[2], x[2]);
+ x[0] = ALIGNR(r[4], r[3], 16);
+ x[1] = SHR(x[0], 57);
+ r[4] = XOR(r[4], x[1]);
+ x[1] = ALIGNR(r[3], r[2], 8);
+ x[2] = SML(r[3], 6);
+ r[3] = XOR(r[3], x[2]);
+ x[2] = SHR(x[0], 59);
+ r[2] = XOR(r[2], x[2]);
+ z[0] = XOR(r[0], r[2]);
+ z[1] = XOR(r[1], r[3]);
+ z[2] = XOR(x[3], r[4]);
+ 
+ /* Clear top */
+ x[0] = SHR(r[4], 27);
+ x[2] = SML(x[0], 5);
+ x[1] = XOR(x[0], x[2]);
+ x[0] = XOR(z[0], x[0]);
+ x[2] = SET64(0x0000000000000000, 0x0000000007FFFFFF);
+ z[2] = AND(z[2], x[2]);
+ }
+
+ static inline void XMM_GF2m_mod_nist409(__m128i z[4], __m128i r[7])
+ {
+ /* Init */
+ __m128i x[3], m[12];
+ 
+ m[0] = SHR(r[6], 2);
+ m[1] = SHL(r[6], 62);
+ m[2] = SHR(r[6], 25);
+ m[3] = SHL(r[6], 39);
+ m[4] = SHR(r[5], 2);
+ m[5] = SHL(r[5], 62);
+ m[6] = SHR(r[5], 25);
+ m[7] = SHL(r[5], 39);
+ m[8] = SHR(r[4], 2);
+ m[9] = SHL(r[4], 62);
+ m[10] = SHR(r[4], 25);
+ m[11] = SHL(r[4], 39);
+ 
+ x[0] = XOR(m[1], m[2]);
+ z[3] = XOR(r[3], x[0]);
+ + x[1] = XOR(m[4], m[3]);
+ + x[2] = ALIGNR(m[0], x[1], 8);
+ + x[3] = XOR(x[3], x[2]);
+ + x[0] = XOR(m[5], m[6]);
+ + z[2] = XOR(r[2], x[0]);
+ + m[7] = XOR(m[7], m[8]);
+ + x[1] = ALIGNR(x[1], n[7], 8);
+ + z[2] = XOR(x[2], x[1]);
+ + x[2] = XOR(m[9], n[10]);
+ + x[1] = XOR(r[1], x[2]);
+ + 
+ /* Clear top */
+ x[0] = SET64(0x0000000000000000, 0x0000000000000000);
+ x[0] = AND(x[0], x[0]);
+ + x[3] = XOR(x[3], x[0]);
+ + m[0] = SHR(x[0], 2);
+ + m[1] = SHL(x[0], 62);
+ + m[2] = SHR(x[0], 25);
static inline void XMM_GF2m_mod_nist571(__m128i z[5], __m128i r[9])
{
    /* Init */
    const int n = 4;
    int i = 8;
    __m128i x[5];
    /* Reduce */
    x[4] = ZERO;
    for (; i > n; i--)
    {
        // Component x
        x[0] = SHL(r[i], 5);
        x[1] = SHR(r[i], 59);
        // Component x
        x[2] = SHL(r[i], 7);
        x[3] = SHR(r[i], 57);
        x[0] = XOR(x[0], x[2]);
        x[1] = XOR(x[1], x[3]);
        // Component x
        x[2] = SHL(r[i], 10);
        x[3] = SHR(r[i], 54);
        x[0] = XOR(x[0], x[2]);
        x[1] = XOR(x[1], x[3]);
        // Component x
        x[2] = SHL(r[i], 15);
        x[3] = SHR(r[i], 49);
        x[0] = XOR(x[0], x[2]);
        x[1] = XOR(x[1], x[3]);
        // Add component z0 -> (r[i-n], r[i-n-1])
        x[2] = ALIGNR(x[4], x[0], 8);
        r[i-n] = XOR(r[i-n], x[2]);
        x[4] = x[0];
        // Add component z1 -> r[i-n]
        r[i-n] = XOR(r[i-n], x[1]);
    }
    x[0] = SHL128(x[4], 8);
    r[i-n] = XOR(r[i-n], x[0]);
    /* Clear top */
    x[4] = SET64(0xFFFFFFFFFFFFFFFF, 0xF800000000000000);
    x[4] = AND(r[4], x[4]);
    r[4] = XOR(r[4], x[4]);
    // Component x
    x[0] = SHR(x[4], 59);
    x[1] = SHL(x[4], 5);
    // Component x
    x[2] = SHR(x[4], 57);
    x[3] = SHL(x[4], 7);
    x[0] = XOR(x[0], x[2]);
    x[1] = XOR(x[1], x[3]);
    // Component x
    x[2] = SHR(x[4], 64);
    x[3] = SHL(x[4], 10);
    x[0] = XOR(x[0], x[2]);
    x[1] = XOR(x[1], x[3]);
}
// Component x
  + x[2] = SHR(x[4], 49);
  + x[3] = SHL(x[4], 15);
  + x[0] = XOR(x[0], x[2]);
  + x[1] = XOR(x[1], x[3]);
  + // Add component x1 -> (r[0], ### )
  + x[1] = SHR128(x[1], 8);
  + r[0] = XOR(r[0], x[1]);
  + // Add component x0 -> r[0]
  + r[0] = XOR(r[0], x[0]);
  + XMM_GF2m_copy_5term(z, r);
  + }
  +}
  +}  

 + XMM_GF2m_sqr_3term(__m128i z[3], const __m128i a[2])
 + {
 +  /* Init */
 +    __m128i x[2], sqrT, mask;
 +    sqrT = SET64(0x5554515045444140, 0x1514111005040100);
 +    mask = SET64(0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F);
 +    x[0] = AND(a[0], mask);
 +    x[1] = SHR(a[0], 4);
 +    x[1] = AND(x[1], mask);
 +    x[0] = SHUFFLE(sqrT, x[0]);
 +    x[1] = SHUFFLE(sqrT, x[1]);
 +    z[0] = UNPACKLO8(x[0], x[1]);
 +    z[1] = UNPACKHI8(x[0], x[1]);
 +    +
 +    x[0] = AND(a[1], mask);
 +    x[1] = SHR(a[1], 4);
 +    x[1] = AND(x[1], mask);
 +    x[0] = SHUFFLE(sqrT, x[0]);
 +    x[1] = SHUFFLE(sqrT, x[1]);
 +    z[2] = UNPACKLO8(x[0], x[1]);
 + }

 + static inline void XMM_GF2m_sqr_4term(__m128i z[4], const __m128i a[2])
 + {
 +  /* Init */
 +    __m128i x[2], sqrT, mask;
 +    sqrT = SET64(0x5554515045444140, 0x1514111005040100);
 +    mask = SET64(0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F);
 +    x[0] = AND(a[0], mask);
 +    x[1] = SHR(a[0], 4);
 +    x[1] = AND(x[1], mask);
 +    x[0] = SHUFFLE(sqrT, x[0]);
 +    x[1] = SHUFFLE(sqrT, x[1]);
 +    z[0] = UNPACKLO8(x[0], x[1]);
 +    z[1] = UNPACKHI8(x[0], x[1]);
 +    x[0] = AND(a[1], mask);


```c
+ x[1] = SHR( a[1], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ }
+
+ static inline void XMM_GF2m_sqr_5term( __m128i z[5], const __m128i a[3])
+ {
+ /* Init */
+ __m128i x[2], sqrT, mask;
+ 
+ sqrT = SET64( 0x5554515045444140, 0x1514111005040100);
+ mask = SET64( 0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F );
+ 
+ x[0] = AND( a[0], mask);
+ x[1] = SHR( a[0], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ 
+ x[0] = AND( a[1], mask);
+ x[1] = SHR( a[1], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ 
+ x[0] = AND( a[2], mask);
+ x[1] = SHR( a[2], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ 
+ static inline void XMM_GF2m_sqr_7term( __m128i z[7], const __m128i a[4])
+ {
+ /* Init */
+ __m128i x[2], sqrT, mask;
+ 
+ sqrT = SET64( 0x5554515045444140, 0x1514111005040100);
+ mask = SET64( 0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F );
+ 
+ x[0] = AND( a[0], mask);
+ x[1] = SHR( a[0], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ 
+ x[0] = AND( a[1], mask);
+ x[1] = SHR( a[1], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ 
+ x[0] = AND( a[2], mask);
+ x[1] = SHR( a[2], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ z[3] = UNPACKHIB8( x[0], x[1] );
+ 
+ x[0] = AND( a[3], mask);
+ x[1] = SHR( a[3], 4);
+ x[1] = AND( x[1], mask);
+ x[0] = SHUFFLE( sqrT, x[0] );
+ x[1] = SHUFFLE( sqrT, x[1] );
+ z[2] = UNPACKLO8( x[0], x[1] );
+ 
+ 79
```
static inline void XMM_GF2m_sqr_9term(__m128i z[9], const __m128i a[5])
{
    /* Init */
    __m128i x[2], sqrT, mask;
    sqrT = SET64(0x5554515045444140, 0x1514111005040100);
    mask = SET64(0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F);
    x[0] = AND(a[0], mask);
    x[1] = SHR(a[0], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE2(sqrT, x[0]);
    z[0] = UNPACKLO8(x[0], x[1]);
    z[1] = UNPACKHI8(x[0], x[1]);

    x[0] = AND(a[1], mask);
    x[1] = SHR(a[1], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE2(sqrT, x[0]);
    z[2] = UNPACKLO8(x[0], x[1]);
    z[3] = UNPACKHI8(x[0], x[1]);

    x[0] = AND(a[2], mask);
    x[1] = SHR(a[2], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE2(sqrT, x[0]);
    z[4] = UNPACKLO8(x[0], x[1]);
    z[5] = UNPACKHI8(x[0], x[1]);

    x[0] = AND(a[3], mask);
    x[1] = SHR(a[3], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE2(sqrT, x[0]);
    z[6] = UNPACKLO8(x[0], x[1]);
    z[7] = UNPACKHI8(x[0], x[1]);

    x[0] = AND(a[4], mask);
    x[1] = SHR(a[4], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE2(sqrT, x[0]);
    z[8] = UNPACKLO8(x[0], x[1]);
}

static inline void XMM_GF2m_mod_sqr_nist163(__m128i z[2], const __m128i a[2])
{
    /* Init */
    __m128i t[3];
    /* Square */
    XMM_GF2m_sqr_3term(t, a);
    /* Reduce */
    XMM_GF2m_mod_nist163(z, t);
}

static inline void XMM_GF2m_mod_sqr_sect193(__m128i z[2], const __m128i a[2])
{
    /* Init */
    __m128i t[4];
    /* Square */
    XMM_GF2m_sqr_4term(t, a);
    /* Reduce */
    XMM_GF2m_mod_sect193(z, t);
}

static inline void XMM_GF2m_mod_sqr_nist233(__m128i z[2], const __m128i a[2])
{
    /* Init */
    __m128i t[4];
    /* Square */
    XMM_GF2m_sqr_4term(t, a);
static inline void XMM_GF2m_mod_sqr_sect239(__m128i z[2], const __m128i a[2]) {
    /* Init */
    __m128i t[4];
    /* Square */
    XMM_GF2m_sqr_4term(t, a);
    /* Reduce */
    XMM_GF2m_mod_sect239(z, t);
}

static inline void XMM_GF2m_mod_sqr_nist283(__m128i z[3], const __m128i a[3]) {
    /* Init */
    __m128i t[5];
    /* Square */
    XMM_GF2m_sqr_5term(t, a);
    /* Reduce */
    XMM_GF2m_mod_nist283(z, t);
}

static inline void XMM_GF2m_mod_sqr_nist409(__m128i z[4], const __m128i a[4]) {
    /* Init */
    __m128i t[7];
    /* Square */
    XMM_GF2m_sqr_7term(t, a);
    /* Reduce */
    XMM_GF2m_mod_nist409(z, t);
}

static inline void XMM_GF2m_mod_sqr_nist571(__m128i z[5], const __m128i a[5]) {
    /* Init */
    __m128i t[9];
    /* Square */
    XMM_GF2m_sqr_9term(t, a);
    /* Reduce */
    XMM_GF2m_mod_nist571(z, t);
}

/* Simple 2-term Karatsuba multiplication. */  
static inline void XMM_GF2m_2x2_mul(__m128i z[2], const __m128i a, const __m128i b) {  
    /* Reduce */
    XMM_GF2m_mod_nist233(z, t);
}
{ /* Init */
    __m128i x[4];
    x[0] = SHl128(a, 8);
    x[1] = XOR(a, x[0]);
    x[2] = SHl128(b, 8);
    x[3] = XOR(b, x[2]);
    /* Do multiplications */
    z[0] = PCLMUL(a, b, 0x00);
    z[1] = PCLMUL(a, b, 0x11);
    x[0] = XOR(z[0], x[1]);
    x[0] = XOR(x[0], x[1]);
    x[1] = SHl128(x[0], 8);
    x[2] = SHl128(x[0], 8);
    z[0] = XOR(z[0], x[1]);
    z[1] = XOR(z[1], x[2]);
}
/* Simple 3-term Karatsuba multiplication. */
static inline void XMM_GF2m_3x3_mul(__m128i z[3], const __m128i a[2],
    const __m128i b[2]) {
    /* Init */
    __m128i m[3], t[4];
    t[0] = ALIGNR(a[1], a[0], 8);
    t[1] = XOR(a[0], t[0]);
    t[2] = ALIGNR(b[1], b[0], 8);
    t[3] = XOR(b[0], t[2]);
    t[0] = XOR(a[0], a[1]);
    t[2] = XOR(b[0], b[1]);
    /* Do multiplications */
    z[0] = PCLMUL(a[0], b[0], 0x00);
    z[1] = PCLMUL(a[0], b[0], 0x11);
    m[0] = PCLMUL(t[1], t[3], 0x00);
    m[1] = PCLMUL(t[2], t[0], 0x00);
    m[2] = PCLMUL(t[1], t[3], 0x11);
    m[0] = XOR(m[0], z[0]);
    m[0] = XOR(m[0], z[1]);
    m[1] = XOR(m[1], z[0]);
    m[1] = XOR(m[1], z[2]);
    m[2] = XOR(m[2], z[1]);
    m[2] = XOR(m[2], z[2]);
    t[0] = SHl128(m[0], 8);
    z[0] = XOR(z[0], t[0]);
    t[1] = ALIGNR(m[1], m[0], 8);
    z[1] = XOR(z[1], t[1]);
    t[3] = SHl128(m[2], 8);
    z[2] = XOR(z[2], t[3]);
}
/* Recursive 4-term Karatsuba multiplication. */
static inline void XMM_GF2m_4x4_mul(__m128i z[4], const __m128i a[2],
    const __m128i b[2]) {
    /* Init */
    __m128i t[4];
    XMM_GF2m_2x2_mul(z, a[0], b[0]);
    XMM_GF2m_2x2_mul(z + 2, a[1], b[1]);
    t[2] = XOR(a[0], a[1]);
    t[3] = XOR(b[0], b[1]);
    XMM_GF2m_2x2_mul(t, t[2], t[3]);
    t[0] = XOR(t[0], z[0]);
    t[0] = XOR(t[0], z[2]);
\[ t[1] = \text{XOR}(t[1], z[1]); \]
\[ t[1] = \text{XOR}(t[1], z[3]); \]
\[ z[1] = \text{XOR}(z[1], t[0]); \]
\[ z[2] = \text{XOR}(z[2], t[1]); \]
\[
\]
\[
\]
\[
\]
\[
\]

/* Advanced 5-term Karatsuba multiplication as suggested in “Five, Six, and Seven-Term Karatsuba-Like Formulae” by Peter L. Montgomery, requiring 13 multiplications. */

```c
static inline void XMM_GF2m_5x5_mul(__m128i z[5], const __m128i a[3],
  const __m128i b[3])
{
  // Init
  __m128i m[13], t[13];

  // Prepare temporary operands
  t[0] = UNPACKLO64(a[0], b[0]);
  t[1] = UNPACKHI64(a[0], b[0]);
  t[2] = UNPACKLO64(a[1], b[1]);
  t[3] = UNPACKHI64(a[1], b[1]);
  t[4] = UNPACKLO64(a[2], b[2]);
  t[5] = XOR(t[0], t[1]);
  t[6] = XOR(t[0], t[2]);
  t[7] = XOR(t[2], t[4]);
  t[8] = XOR(t[3], t[4]);
  t[9] = XOR(t[3], t[6]);
  t[10] = XOR(t[1], t[7]);
  t[11] = XOR(t[5], t[8]);
  t[12] = XOR(t[2], t[11]);

  // Do multiplications
  m[0] = PCLMUL(t[0], t[0], 0x01);
  m[1] = PCLMUL(t[1], t[1], 0x01);
  m[2] = PCLMUL(t[2], t[2], 0x01);
  m[3] = PCLMUL(t[3], t[3], 0x01);
  m[4] = PCLMUL(t[4], t[4], 0x01);
  m[5] = PCLMUL(t[5], t[5], 0x01);
  m[6] = PCLMUL(t[6], t[6], 0x01);
  m[7] = PCLMUL(t[7], t[7], 0x01);
  m[8] = PCLMUL(t[8], t[8], 0x01);
  m[9] = PCLMUL(t[9], t[9], 0x01);
  m[10] = PCLMUL(t[10], t[10], 0x01);
  m[11] = PCLMUL(t[11], t[11], 0x01);
  m[12] = PCLMUL(t[12], t[12], 0x01);

  // Combine results
  t[0] = m[0];
  t[1] = XOR(t[0], m[1]);
  t[2] = XOR(t[1], m[6]);
  t[3] = XOR(t[1], m[6]);
  t[2] = XOR(t[2], m[2]);
  t[7] = XOR(t[8], m[3]);
  t[6] = XOR(t[7], m[7]);
  t[7] = XOR(t[7], m[8]);
  t[6] = XOR(t[6], m[3]);
  t[5] = XOR(t[5], m[9]);
  t[3] = XOR(t[3], t[0]);
  t[3] = XOR(t[3], t[6]);
  t[4] = XOR(t[1], t[7]);
  t[4] = XOR(t[4], m[9]);
  t[4] = XOR(t[4], m[10]);
  t[4] = XOR(t[4], m[12]);
  t[5] = XOR(t[5], t[2]);
  t[5] = XOR(t[5], t[4]);
  t[5] = XOR(t[6], m[10]);
  t[9] = SHRL28(t[7], 8);
  t[7] = ALIGNR(t[7], t[6], 8);
  t[5] = ALIGNR(t[6], t[3], 8);
  t[3] = ALIGNR(t[3], t[1], 8);
  t[1] = SHL128(t[1], 8);
  z[0] = XOR(t[0], t[1]);
```
```
```c
+ z[1] = XOR(t[2], t[3]);
+ z[2] = XOR(t[4], t[5]);
+ z[3] = XOR(t[6], t[7]);
+ z[4] = XOR(t[8], t[9]);
+ }
+ */ 7-term Karatsuba multiplication with 6-4-3 strategy. */
+static inline void XMM_GF2m_7x7_mul(__m128i z[7], const __m128i a[4],
+ const __m128i b[4])
+ {
+ /* Init */
+ __m128i t[4], e[4];
+ /* Multiply lower part */
+ XMM_GF2m_4x4_mul(z, a, b);
+ /* Multiply upper part */
+ XMM_GF2m_3x3_mul(z + 4, a + 2, b + 2);
+ t[0] = XOR(a[0], a[2]);
+ t[1] = XOR(a[1], a[3]);
+ t[2] = XOR(b[0], b[2]);
+ t[3] = XOR(b[1], b[3]);
+ /* Multiply middle part */
+ XMM_GF2m_4x4_mul(e, t + 2, t);
+ /* Combine results */
+ t[0] = XOR(e[0], z[4]);
+ t[1] = XOR(e[1], z[5]);
+ t[2] = XOR(e[2], z[3]);
+ e[0] = XOR(t[0], z[0]);
+ e[1] = XOR(t[1], z[1]);
+ e[2] = XOR(t[2], z[2]);
+ z[2] = XOR(z[2], e[0]);
+ z[3] = XOR(z[3], e[1]);
+ z[4] = XOR(z[4], e[2]);
+ z[5] = XOR(z[5], t[3]);
+ }
+ */ 9-term Karatsuba multiplication with 8-5-4 strategy. */
+static inline void XMM_GF2m_9x9_mul(__m128i z[9], const __m128i a[5],
+ const __m128i b[5])
+ {
+ /* Init */
+ __m128i t[5], e[4], f[6], at[5], bt[5];
+ /* Multiply lower part */
+ XMM_GF2m_5x5_mul(z, a, b);
+ /* Make local copy of a,b to not destroy them */
+ at[4] = ALIGNR(a[4], a[3], 8);
+ at[3] = ALIGNR(a[3], a[2], 8);
+ at[2] = MOV64(a[2]);
+ XMM_GF2m_copy_2term(at, a);
+ bt[4] = ALIGNR(b[4], b[3], 8);
+ bt[3] = ALIGNR(b[3], b[2], 8);
+ bt[2] = MOV64(b[2]);
+ XMM_GF2m_copy_2term(bt, b);
+ /* Prepare operands */
+ t[0] = XOR(at[0], at[3]); // t0 = [ (a6+a1);(a5+a0) ]
+ t[1] = XOR(at[1], at[4]); // t1 = [ (a8+a3);(a7+a2) ]
+ t[2] = at[2]; // t2 = [ 0;a4 ]
+ e[0] = XOR(bt[0], bt[3]); // e0 = [ (b6+b1);(b5+b0) ]
+ e[1] = XOR(bt[1], bt[4]); // e1 = [ (b8+b3);(b7+b2) ]
+ e[2] = bt[2]; // e2 = [ 0;b4 ]
+ /* Multiply middle part */
+ XMM_GF2m_5x5_mul(f, t, e);
+ t[0] = XOR(f[0], z[0]);
+ t[1] = XOR(f[1], z[1]);
+ ```
t[2] = XOR(f[2], z[2]);
+ t[3] = XOR(f[3], z[3]);
+ t[4] = XOR(f[4], z[4]);
+ /* Multiply upper part */
+ XMM_GF2m_4x4_mul(z + 5, at + 3, bt + 3);
+ /* Combine results */
+ e[0] = XOR(t[0], z[6]);
+ e[1] = XOR(t[1], z[6]);
+ e[2] = XOR(t[2], z[7]);
+ e[3] = XOR(t[3], z[8]);
+ f[0] = SHL128(e[0], 8);
+ z[2] = XOR(z[2], f[0]);
+ f[1] = ALIGNR(e[1], e[0], 8);
+ z[3] = XOR(e[3], e[1], 8);
+ z[4] = XOR(z[4], f[2]);
+ f[3] = ALIGNR(e[3], e[2], 8);
+ z[5] = XOR(z[5], f[3]);
+ f[4] = ALIGNR(t[4], e[3], 8);
+ z[6] = XOR(z[6], f[4]);
+ f[0] = SHR128(t[4], 8);
+ z[7] = XOR(z[7], f[0]);
static inline void XMM_GF2m_mod_mul_nist283(__m128i z[3], const __m128i a[3], const __m128i b[3]) {
    /* Init */
    __m128i t[5];
    /* Do multiplication */
    XMM_GF2m_5x5_mul(t, a, b);
    /* Reduce */
    XMM_GF2m_mod_nist283(z, t);
}

static inline void XMM_GF2m_mod_mul_nist409(__m128i z[4], const __m128i a[4], const __m128i b[4]) {
    /* Init */
    __m128i t[7];
    /* Do multiplication */
    XMM_GF2m_7x7_mul(t, a, b);
    /* Reduce */
    XMM_GF2m_mod_nist409(z, t);
}

static inline void XMM_GF2m_mod_mul_nist571(__m128i z[5], const __m128i a[5], const __m128i b[5]) {
    /* Init */
    __m128i t[9];
    /* Do multiplication */
    XMM_GF2m_9x9_mul(t, a, b);
    /* Reduce */
    XMM_GF2m_mod_nist571(z, t);
}

static inline void XMM_GF2m_mod_inv_nist163(__m128i z[2], __m128i a[2]) {
    /* Init */
    int i;
    __m128i t0[2], t1[2], t2[2];
    /* Exponent chain */
    const int chain[] = { /*1, 2,*/ 4, 8, 16, 32, 64, 32, 2 };  
    /* Initial Square z = a */
    XMM_GF2m_mod_sqr_nist163(z, a);
    /* Square t0 = z^2 */
    XMM_GF2m_mod_sqr_nist163(t0, z);
```c
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+ /* Save component t1 = z */
+ XMM_GF2m_copy_2term(t1, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist163(t0, t0);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist163(t0, t0), chain[0] -1);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+ /* Square t0 = z^4 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist163(t0, t0), chain[1] -1);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+ /* Square t0 = z^8 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist163(t0, t0), chain[2] -1);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+ /* Save component t2 = z */
+ XMM_GF2m_copy_2term(t2, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist163(t0, t0), chain[3] -1);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+ /* Square t0 = z^64 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist163(t0, t0), chain[4] -1);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, z, t0);
+ /* Square t0 = z^32 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist163(t0, t0), chain[5] -1);
+ /* Multiply component z = z * t0 */
+ XMM_GF2m_mod_mul_nist163(z, t2, t0);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist163(t0, z);
+ XMM_GF2m_mod_sqr_nist163(t0, t0);
+ /* Multiply component z = t1 * t0 */
+ XMM_GF2m_mod_mul_nist163(z, t1, t0);
+ }
+ */
+ /* Calculates z = a for any a GF(2^193) with exponent decomposition:
+ (1 + 2^3)(1 + 2^2)(1 + 2^3)(1 + 2^8)(1 + 2^16)(1 + 2^32)(1 + 2^64)(1 + 2^64)
+ */
+ static inline void XMM_GF2m_mod_inv_sect193(__m128i z[2], __m128i a[2])
+ {
+   /* Init */
+   int i;
+   __m128i t0[2], t1[2];
+   /* Exponent chain */
+   const int chain[] = { /*1, 2,*/ 4, 8, 16, 32, 64, 64 };
+   /*
+   */
+   /*
+   */
+   /*
+   */
+ }
Calculates $z = a$ for any $a \in GF(2^{233})$ with exponent decomposition:

$1 + 2 \cdot 16 + 1 + 2 \cdot 8 + 1 + 2 \cdot 32 + 1 + 2 \cdot 64 (1 + 2 \cdot 4) + 1 + 2 \cdot 16 + 1 + 2 \cdot 32 (1 + 2 \cdot 32) (1 + 2 \cdot 64 (1 + 2 \cdot 64) )$

static inline void XMM_GF2m_mod_inv_nist233(__m128i z[2], __m128i a[2])
{
    /* Init */
    int i;
    __m128i t0[2], t1[2], t2[2], t3[2];
    /* Exponent chain */
    const int chain[] = { 1, 2, 4, 8, 16, 32, 64, 128, 32, 8 };
    /* Initial Square $z = a$ */
    XMM_GF2m_mod_sqr_nist233(z, a);
}
/* Square t0 = $z^2$ */
XMM_GF2m_mod_sqr_nis233(t0, z);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Square t0 = $z^3$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
XMM_GF2m_mod_sqr_nis233(t0, t0);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Square t0 = $z^4$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[0] -1);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Save component $t1 = z$ */
XMM_GF2m_copy_2term(t1, z);

/* Square t0 = $z^8$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[1] -1);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Square t0 = $z^{16}$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[2] -1);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Save component $t2 = z$ */
XMM_GF2m_copy_2term(t2, z);

/* Square t0 = $z^{32}$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[3] -1);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Save component $t3 = z$ */
XMM_GF2m_copy_2term(t3, z);

/* Square t0 = $z^{64}$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[4] -1);

/* Multiply component $z = z \cdot t0$ */
XMM_GF2m_mod_mul_nis233(z, z, t0);

/* Square t0 = $z^{128}$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[5] -1);

/* Multiply component $z = t0 \cdot t0$ */
XMM_GF2m_mod_mul_nist233(z, z, t0);

/* Square t0 = $z^{256}$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nis233(t0, t0), chain[6] -1);

/* Multiply component $z = t2 \cdot t0$ */
XMM_GF2m_mod_mul_nist233(z, z, t0);

/* Square t0 = $z^{512}$ */
XMM_GF2m_mod_sqr_nis233(t0, z);
LOOP(XMM_GF2m_mod_sqr_nist233(t0, t0), chain[7] -1);

/* Multiply component $z = t1 \cdot t0$ */
XMM_GF2m_mod_mul_nist233(z, z, t0);

}
/* Calculates $z = a$ for any $a \in GF(2^{239})$ with exponent decomposition:
\begin{align*}
(1 + 2)(1 + 2^2)(1 + 2^4)(1 + 2^8)(1 + 2^{16})(1 + 2^{32})(1 + 2^{64})(1 + 2^{128})(1 + 2^{256})
\end{align*}
*/

static inline void XMM_GF2m_mod_inv_sect239(__m128i z[2], __m128i a[2]) {
  {
    /* Init */
    int i;
    __m128i t0[2], t1[2], t2[2], t3[2], t4[2], t5[2];
    /* Exponent chain */
    const int chain[] = { /*1, 2, */ 4, 8, 16, 32, 64, 64, 32, 32, 8, 4, 2 };
    /* Initial Square $z = a$ */
    XMM_GF2m_mod_sqr_sect239(z, a);
    /* Square $t0 = z^{2^1}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    /* Multiply component $z = z \times t0$ */
    XMM_GF2m_mod_mul_sect239(z, z, t0);
    /* Save component $t1 = z$ */
    XMM_GF2m_copy_2term(t1, z);
    /* Square $t0 = z^{2^2}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    XMM_GF2m_mod_sqr_sect239(t0, t0);
    /* Multiply component $z = z \times t0$ */
    XMM_GF2m_mod_mul_sect239(z, z, t0);
    /* Save component $t2 = z$ */
    XMM_GF2m_copy_2term(t2, z);
    /* Square $t0 = z^{2^4}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    LOOP (XMM_GF2m_mod_sqr_sect239(t0, t0), chain[0]-1);
    /* Multiply component $z = z \times t0$ */
    XMM_GF2m_mod_mul_sect239(z, z, t0);
    /* Save component $t3 = z$ */
    XMM_GF2m_copy_2term(t3, z);
    /* Square $t0 = z^{2^8}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    LOOP (XMM_GF2m_mod_sqr_sect239(t0, t0), chain[1]-1);
    /* Multiply component $z = z \times t0$ */
    XMM_GF2m_mod_mul_sect239(z, z, t0);
    /* Square $t0 = z^{2^{16}}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    LOOP (XMM_GF2m_mod_sqr_sect239(t0, t0), chain[2]-1);
    /* Multiply component $z = z \times t0$ */
    XMM_GF2m_mod_mul_sect239(z, z, t0);
    /* Save component $t4 = z$ */
    XMM_GF2m_copy_2term(t4, z);
    /* Square $t0 = z^{2^{32}}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    LOOP (XMM_GF2m_mod_sqr_sect239(t0, t0), chain[3]-1);
    /* Multiply component $z = z \times t0$ */
    XMM_GF2m_mod_mul_sect239(z, z, t0);
    /* Save component $t5 = z$ */
    XMM_GF2m_copy_2term(t5, z);
    /* Square $t0 = z^{2^{64}}$ */
    XMM_GF2m_mod_sqr_sect239(t0, z);
    LOOP (XMM_GF2m_mod_sqr_sect239(t0, t0), chain[4]-1);
    /* Multiply component $z = z \times t0$ */
  }
}
XMM_GF2m_mod_mul_sec t 239(z, z, t0);
/* Square t0 = z\(^2\) \^32 */
XMM_GF2m_mod_sqr_sec t 239(t0, z);
LOOP(XMM_GF2m_mod_sqr_sec t 239(t0, t0), chain[5] -1);
/* Multiply component z = t0\times t0 */
XMM_GF2m_mod_mul_sec t 239(z, t6, t0);
/* Square t0 = z\(^2\) \^8 */
XMM_GF2m_mod_sqr_sec t 239(t0, z);
LOOP(XMM_GF2m_mod_sqr_sec t 239(t0, t0), chain[6] -1);
/* Multiply component z = t1\times t0 */
XMM_GF2m_mod_mul_sec t 239(z, t4, t0);
/* Square t0 = z\(^2\) \^4 */
XMM_GF2m_mod_sqr_sec t 239(t0, z);
LOOP(XMM_GF2m_mod_sqr_sec t 239(t0, t0), chain[7] -1);
/* Multiply component z = t3\times t0 */
XMM_GF2m_mod_mul_sec t 239(z, t2, t0);
/* Square t0 = z\(^2\) \^2 */
XMM_GF2m_mod_sqr_sec t 239(t0, z);
LOOP(XMM_GF2m_mod_sqr_sec t 239(t0, t0), chain[8] -1);
/* Multiply component z = t1\times t0 */
XMM_GF2m_mod_mul_sec t 239(z, t1, t0);
}

/* Calculates z = a for any a GF(2\(^283\)) with exponent decomposition:
 * (1 + 2) (1 + 2\(^2\)) (1 + 2\(^4\)) (1 + 2\(^8\)) (1 + 2\(^16\)) (1 + 2\(^32\)) (1 + 2\(^64\)) (1 + 2\(^128\)) */
static inline void XMM_GF2m_mod_inv_nist283(__m128i z[3], __m128i a[3])
{
    __m128i t0[3], t1[3], t2[3], t3[3];
    __m128i t1[3], t2[3], t3[3];
    const int chain[] = { /*1, 2,*/ 4, 8, 16, 32, 64, 128, 16, 8, 2 };
    __m128i t1 = a[3];
    XMM_GF2m_mod_sqr_nist283(z, a);
    XMM_GF2m_mod_sqr_nist283(t0, z);
    XMM_GF2m_mod_mul_nist283(z, z, t0);
    /* Save component t1 = z */
    XMM_GF2m_copy_term(t1, z);
    XMM_GF2m_mod_sqr_nist283(t0, z);
    XMM_GF2m_mod_sqr_nist283(t0, t0);
    XMM_GF2m_mod_mul_nist283(z, z, t0);
    XMM_GF2m_mod_sqr_nist283(t0, z);
    LOOP(XMM_GF2m_mod_sqr_nist283(t0, t0), chain[0] -1);
    XMM_GF2m_mod_mul_nist283(z, z, t0);
    /* Save component t2 = z */
}
/* Calculates \(z = a\) for any \(a\) GF(\(2^{409}\)) with exponent decomposition:
\[
(1 + 2)(1 + 2^2)(1 + 2^4)(1 + 2^8)(1 + 2^16)(1 + 2^32)(1 + 2^64)(1 + 2^{128})(1 + 2^{128})
\]
*/

static inline void XMM_GF2m_mod_inv_nist409(__m128i z[4], __m128i a[4]) {
    /* Init */
    int i;
    __m128i t0[4], t1[4], t2[4], t3[4];

    /* Exponent chain */
    const int chain[] = { /x1, 2,*/ 4, 8, 16, 32, 64, 128, 128, 16, 8 }

    /* Initial Square \(z = a\) */
    XMM_GF2m_mod_sqr_nist409(z, a);
/* Square $t_0 = z^{2^1}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Square $t_0 = z^{2^2}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
XMM_GF2m_mod_sqr_nist409(t0, t0);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Square $t_0 = z^{2^4}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_1 = z$ */
XMM_GF2m_copy_4term(t1, z);
/* Square $t_0 = z^{2^8}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_2 = z$ */
XMM_GF2m_copy_4term(t2, z);
/* Square $t_0 = z^{2^16}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_3 = z$ */
XMM_GF2m_copy_4term(t3, z);
/* Square $t_0 = z^{2^32}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_4 = z$ */
XMM_GF2m_copy_4term(t4, z);
/* Square $t_0 = z^{2^64}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_5 = z$ */
XMM_GF2m_copy_4term(t5, z);
/* Square $t_0 = z^{2^128}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_6 = z$ */
XMM_GF2m_copy_4term(t6, z);
/* Square $t_0 = z^{2^256}$ */
XMM_GF2m_mod_sqr_nist409(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist409( t0 , t0), chain[0], -1);
/* Multiply component $z = z \cdot t_0$ */
XMM_GF2m_mod_mul_nist409(z, z, t0);
/* Save component $t_7 = z$ */
XMM_GF2m_copy_4term(t7, z);
+ XMM_GF2m_mod_sqr_nist409(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist409(t0, t0), chain[8]-1);
+ /* Multiply component z = t1*t0 */
+ XMM_GF2m_mod_mul_nist409(z, t1, t0);
+ }
+
+ /* Calculates z = a for any GF(2^571) with exponent decomposition: */
+ /* (1 + 2^2) (1 + 2^2) (1 + 2^4) (1 + 2^8) (1 + 2^8) (1 + 2^16) (1 + 2^16) (1 + 2^32) (1 + 2^32) (1 + 2^64) (1 + 2^128) (1 + 2^256)) */
+ static inline void XMM_GF2m_mod_inv_nist571( __m128i z[5], __m128i a[5])
+ {
+ /* Init */
+ int i;
+ __m128i t0[5], t1[5], t2[5], t3[5], t4[5];
+ /* Exponent chain */
+ const int chain[] = { /*1, 2,*/ 4, 8, 16, 32, 64, 128, 256, 32, 16, 8, 2 };
+ /* Initial Square z = a */
+ XMM_GF2m_mod_sqr_nist571(z, a);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Save component t1 = z */
+ XMM_GF2m_copy_5term(t1, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ XMM_GF2m_mod_sqr_nist571(t0, t0);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Save component t2 = z */
+ XMM_GF2m_copy_5term(t2, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[0]-1);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Save component t3 = z */
+ XMM_GF2m_copy_5term(t3, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[1]-1);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Save component t4 = z */
+ XMM_GF2m_copy_5term(t4, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[2]-1);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Save component t5 = z */
+ XMM_GF2m_copy_5term(t5, z);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[3]-1);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[4]-1);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[5]-1);
+ /* Multiply component z = z*t0 */
+ XMM_GF2m_mod_mul_nist571(z, z, t0);
+ /* Square t0 = z^2 */
+ XMM_GF2m_mod_sqr_nist571(t0, z);
+ LOOP(XMM_GF2m_mod_sqr_nist571(t0, t0), chain[6]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_nist571(z, z, t0);
/* Square t0 = z^2^128 */
XMM_GF2m_mod_sqr_nist571(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist571(t0, t0), chain[4] -1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_nist571(z, z, t0);
/* Square t0 = z^2^256 */
XMM_GF2m_mod_sqr_nist571(t0, z);
LOOP (XMM_GF2m_mod_sqr_nist571(t0, t0), chain[6] -1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_nist571(z, z, t0);
/* Square t0 = z^2^32 */
XMM_GF2m_mod_sqr_nist571(t0, z);
/* Multiply component z = t4*t0 */
XMM_GF2m_mod_mul_nist571(z, t4, t0);
/* Square t0 = z^2^16 */
XMM_GF2m_mod_sqr_nist571(t0, z);
/* Multiply component z = t3*t0 */
XMM_GF2m_mod_mul_nist571(z, t3, t0);
/* Square t0 = z^2^8 */
XMM_GF2m_mod_sqr_nist571(t0, z);
/* Multiply component z = t2*t0 */
XMM_GF2m_mod_mul_nist571(z, t2, t0);
/* Square t0 = z^2^2 */
XMM_GF2m_mod_sqr_nist571(t0, z);
/* Multiply component z = t1*t0 */
XMM_GF2m_mod_mul_nist571(z, t1, t0);
} /* Calculates z = a/b = a * b for a,b,z GF(2^163) */
static inline void XMM_GF2m_div_nist163(__m128i z[2], __m128i a[2],
+ __m128i b[2])
+ { + /* Init */
+ __m128i c[2];
+ /* Invert b */
+ XMM_GF2m_mod_inv_nist163(c, b);
+ /* Multiply a * b */
+ XMM_GF2m_mod_mul_nist163(z, a, c);
+ }
/* Calculates z = a/b = a * b for a,b,z GF(2^193) */
static inline void XMM_GF2m_div_sect193(__m128i z[2], __m128i a[2],
+ __m128i b[2])
+ { + /* Init */
+ __m128i c[2];
+ /* Invert b */
+ XMM_GF2m_mod_inv_sect193(c, b);
+ /* Multiply a * b */
+ XMM_GF2m_mod_mul_sect193(z, a, c);
+ }
/* Calculates \( z = \frac{a}{b} = a \times b \) for \( a, b, z \) GF \((2^{233})\) */

static inline void XMM_GF2m_div_nist233(__m128i z[2], __m128i a[2],
+ __m128i b[2])
{
    /* Init */
    __m128i c[2];

    /* Invert b */
    XMM_GF2m_mod_inv_nist233(c, b);

    /* Multiply a * b */
    XMM_GF2m_mod_mul_nist233(z, a, c);
}

/* Calculates \( z = \frac{a}{b} = a \times b \) for \( a, b, z \) GF \((2^{239})\) */

static inline void XMM_GF2m_div_sect239(__m128i z[2], __m128i a[2],
+ __m128i b[2])
{
    /* Init */
    __m128i c[2];

    /* Invert b */
    XMM_GF2m_mod_inv_sect239(c, b);

    /* Multiply a * b */
    XMM_GF2m_mod_mul_sect239(z, a, c);
}

/* Calculates \( z = \frac{a}{b} = a \times b \) for \( a, b, z \) GF \((2^{283})\) */

static inline void XMM_GF2m_div_nist283(__m128i z[3], __m128i a[3],
+ __m128i b[3])
{
    /* Init */
    __m128i c[3];

    /* Invert b */
    XMM_GF2m_mod_inv_nist283(c, b);

    /* Multiply a * b */
    XMM_GF2m_mod_mul_nist283(z, a, c);
}

/* Calculates \( z = \frac{a}{b} = a \times b \) for \( a, b, z \) GF \((2^{409})\) */

static inline void XMM_GF2m_div_nist409(__m128i z[4], __m128i a[4],
+ __m128i b[4])
{
    /* Init */
    __m128i c[4];

    /* Invert b */
    XMM_GF2m_mod_inv_nist409(c, b);

    /* Multiply a * b */
    XMM_GF2m_mod_mul_nist409(z, a, c);
}

/* Calculates \( z = \frac{a}{b} = a \times b \) for \( a, b, z \) GF \((2^{571})\) */

static inline void XMM_GF2m_div_nist571(__m128i z[5], __m128i a[5],
+ __m128i b[5])
{
    /* Init */
    __m128i c[5];

    /* Invert b */
    XMM_GF2m_mod_inv_nist571(c, b);

    /* Multiply a * b */
    XMM_GF2m_mod_mul_nist571(z, a, c);
}

/*******************************************************************************/

/* BIGNUM WRAPPER FUNCTIONS */

/* This section provides non-static wrapper functions for selected SECT/NIST curves */
/* for the use with the OpenSSL BN/EC library. */
/* All BIGNUMs are expected to have the correct amount of words according to the
* field. This means that a field element in GF(2^m) [m prime] needs to have
* exactly (m/64)+1 words.
*/

/*================================================================***************************/

/* Calculates z = a for all a GF(2^163) */
int BN_GF2m_sqr_xmm_nist163 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t [2], _a [2];
    /* Load */
    BN_to_XMM_3term (_a , a->d);
    /* Square */
    XMM_GF2m_mod_sqr_nist163 (_t , _a);
    /* Store */
    XMM_to_BN_3term (z->d, _t);
    ret = 1;
    return ret;
}

/*================================================================***************************/

/* Calculates z = a for all a GF(2^193) */
int BN_GF2m_sqr_xmm_sect193 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t [2], _a [2];
    /* Load */
    BN_to_XMM_4term (_a , a->d);
    /* Square */
    XMM_GF2m_mod_sqr_sect193 (_t , _a);
    /* Store */
    XMM_to_BN_4term (z->d, _t);
    ret = 1;
    return ret;
}

/*================================================================***************************/

/* Calculates z = a for all a GF(2^233) */
int BN_GF2m_sqr_xmm_nist233 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t [2], _a [2];
    /* Load */
    BN_to_XMM_4term (_a , a->d);
    /* Square */
    XMM_GF2m_mod_sqr_nist233 (_t , _a);
    /* Store */
    XMM_to_BN_4term (z->d, _t);
    ret = 1;
    return ret;
}

/*================================================================***************************/

/* Calculates z = a for all a GF(2^239) */
int BN_GF2m_sqr_xmm_sect239 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t [2], _a [2];
    /* Load */
    BN_to_XMM_4term (_a , a->d);
    /* Square */
    XMM_GF2m_mod_sqr_sect239 (_t , _a);
    /* Store */
    XMM_to_BN_4term (z->d, _t);
    ret = 1;
    return ret;
}
int BN_GF2m_sqr_xmm_nist283 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t[3], _a[3];
    /* Load */
    BN_to_XMM_5term(_a, a->d);
    /* Square */
    XMM_GF2m_mod_sqr_nist283(_t, _a);
    /* Store */
    XMM_to_BN_5term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_sqr_xmm_nist409 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t[4], _a[4];
    /* Load */
    BN_to_XMM_7term(_a, a->d);
    /* Square */
    XMM_GF2m_mod_sqr_nist409(_t, _a);
    /* Store */
    XMM_to_BN_7term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_sqr_xmm_nist571 (BIGNUM *z, const BIGNUM *a) {
    /* Init */
    int ret = 0;
    __m128i _t[5], _a[5];
    /* Load */
    BN_to_XMM_9term(_a, a->d);
    /* Square */
    XMM_GF2m_mod_sqr_nist571(_t, _a);
    /* Store */
    XMM_to_BN_9term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_mul_xmm_nist163 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    BN_to_XMM_3term(_a, a->d);
    BN_to_XMM_3term(_b, b->d);
}
int BN_GF2m_mul_xmm_nist163(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    BN_to_XMM_4term(_a, a->d);
    BN_to_XMM_4term(_b, b->d);
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul_nist163(_t, _a, _b);
    /* Store */
    XMM_to_BN_4term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_mul_xmm_sect193(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    BN_to_XMM_4term(_a, a->d);
    BN_to_XMM_4term(_b, b->d);
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul_sect193(_t, _a, _b);
    /* Store */
    XMM_to_BN_4term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_mul_xmm_nist233(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    BN_to_XMM_4term(_a, a->d);
    BN_to_XMM_4term(_b, b->d);
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul_nist233(_t, _a, _b);
    /* Store */
    XMM_to_BN_4term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_mul_xmm_sect239(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    BN_to_XMM_4term(_a, a->d);
    BN_to_XMM_4term(_b, b->d);
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul_sect239(_t, _a, _b);
    /* Store */
    XMM_to_BN_4term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_mul_xmm_nist283(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    int ret = 0;
    __m128i _t[3], _a[3], _b[3];
BN_to_XMM_5term (_a, a->d);
BN_to_XMM_5term (_b, b->d);

/* Multiply & Reduce*/
XMM_GF2m_mod_mul_nist283 (_t, _a, _b);

/* Store */
XMM_to_BN_5term (z->d, _t);
ret = 1;
return ret;
}

/* Calculates z = a * b for all a, b GF(2^409) */
int BN_GF2m_mul_xmm_nist409 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
  /* Init */
  int ret = 0;
  __m128i _t[4], _a[4], _b[4];
  /* Load */
  BN_to_XMM_7term (_a, a->d);
  BN_to_XMM_7term (_b, b->d);
  /* Multiply & Reduce*/
  XMM_GF2m_mod_mul_nist409 (_t, _a, _b);
  /* Store */
  XMM_to_BN_7term (z->d, _t);
  ret = 1;
  return ret;
}

/* Calculates z = a * b for all a, b GF(2^571) */
int BN_GF2m_mul_xmm_nist571 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
  /* Init */
  int ret = 0;
  __m128i _t[5], _a[5], _b[5];
  /* Load */
  BN_to_XMM_9term (_a, a->d);
  BN_to_XMM_9term (_b, b->d);
  /* Multiply & Reduce*/
  XMM_GF2m_mod_mul_nist571 (_t, _a, _b);
  /* Store */
  XMM_to_BN_9term (z->d, _t);
  ret = 1;
  return ret;
}

/* Calculates z = a/b = a * b for all a, b GF(2^163) */
int BN_GF2m_div_xmm_nist163 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
  /* Init */
  int ret = 0;
  __m128i _t[2], _a[2], _b[2];
  /* Load */
  BN_to_XMM_3term (_a, a->d);
  BN_to_XMM_3term (_b, b->d);
  /* Divide */
  XMM_GF2m_div_nist163 (_t, _a, _b);
  /* Store */
  XMM_to_BN_3term (z->d, _t);
  ret = 1;
  return ret;
}

/* Calculates z = a/b = a * b for all a, b GF(2^193) */
int BN_GF2m_div_xmm_nist193 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
  /* Init */
  int ret = 0;
  __m128i _t[2], _a[2], _b[2];
  /* Load */
  BN_to_XMM_3term (_a, a->d);
  BN_to_XMM_3term (_b, b->d);
  /* Divide */
  XMM_GF2m_div_nist193 (_t, _a, _b);
  /* Store */
  XMM_to_BN_3term (z->d, _t);
  ret = 1;
  return ret;
}
```c
/* Calculates z = a/b = a * b for all a, b GF(2^193) */
int BN_GF2m_div_xmm_sect193(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
+
+ ( *
+ /* Init */
+ int ret = 0;
+ __m128i _t[2], _a[2], _b[2];
+ /* Load */
+ BN_to_XMM_4term(_a, a->d);
+ BN_to_XMM_4term(_b, b->d);
+ /* Divide */
+ IMM_GF2m_div_sect193(_t, _a, _b);
+ /* Store */
+ IMM_to_BN_4term(z->d, _t);
+ ret = 1;
+ return ret;
+ )
+
+ /* Calculates z = a/b = a * b for all a, b GF(2^233) */
+int BN_GF2m_div_xmm_nist233(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
+
+ ( *
+ /* Init */
+ int ret = 0;
+ __m128i _t[2], _a[2], _b[2];
+ /* Load */
+ BN_to_XMM_4term(_a, a->d);
+ BN_to_XMM_4term(_b, b->d);
+ /* Divide */
+ IMM_GF2m_div_nist233(_t, _a, _b);
+ /* Store */
+ IMM_to_BN_4term(z->d, _t);
+ ret = 1;
+ return ret;
+ )
+
+ /* Calculates z = a/b = a * b for all a, b GF(2^239) */
+int BN_GF2m_div_xmm_sect239(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
+
+ ( *
+ /* Init */
+ int ret = 0;
+ __m128i _t[2], _a[2], _b[2];
+ /* Load */
+ BN_to_XMM_4term(_a, a->d);
+ BN_to_XMM_4term(_b, b->d);
+ /* Divide */
+ IMM_GF2m_div_sect239(_t, _a, _b);
+ /* Store */
+ IMM_to_BN_4term(z->d, _t);
+ ret = 1;
+ return ret;
+ )
+
+ /* Calculates z = a/b = a * b for all a, b GF(2^283) */
+int BN_GF2m_div_xmm_nist283(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
+
+ ( *
+ /* Init */
+ int ret = 0;
+ __m128i _t[3], _a[3], _b[3];
+ /* Load */
+ BN_to_XMM_5term(_a, a->d);
+ BN_to_XMM_5term(_b, b->d);
+ /* Divide */
+ IMM_GF2m_div_nist283(_t, _a, _b);
+ /* Store */
+ IMM_to_BN_5term(z->d, _t);
+ ret = 1;
+ return ret;
+ )
```
int BN_GF2m_div_xmm_nist409(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    /* Init */
    int ret = 0;
    __m128i _t[4], _a[4], _b[4];
    /* Load */
    BN_to_XMM_7term(_a, a->d);
    BN_to_XMM_7term(_b, b->d);
    /* Divide */
    XMM_GF2m_div_nist409(_t, _a, _b);
    /* Store */
    XMM_to_BN_7term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_div_xmm_nist571(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    /* Init */
    int ret = 0;
    __m128i _t[5], _a[5], _b[5];
    /* Load */
    BN_to_XMM_9term(_a, a->d);
    BN_to_XMM_9term(_b, b->d);
    /* Divide */
    XMM_GF2m_div_nist571(_t, _a, _b);
    /* Store */
    XMM_to_BN_9term(z->d, _t);
    ret = 1;
    return ret;
}

int BN_GF2m_Maddle_xmm_nist163k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
    /* Init */
    int ret = 0;
    __m128i _t1[2], _t2[3], _t3[3], _x1[2], _z1[2], _x2[2], _z2[2];
    /* Load */
    BN_to_XMM_3term(_t1, x1->d);
    BN_to_XMM_3term(_t2, x2->d);
    BN_to_XMM_3term(_t3, z1->d);
    BN_to_XMM_3term(_t4, z2->d);
    /* Data veiling */
+ XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
+ /* MADD */
+ XMM_GF2m_mod_mul_nist163(_x1, _x1, _z2);
+ XMM_GF2m_mod_mul_nist163(_z1, _z1, _x2);
+ /* Multiply w/o reduction */
+ XMM_GF2m_3x3_mul(_t2, _x1, _z1);
+ XMM_GF2m_add_2term(_x1, _x1, _z1);
+ XMM_GF2m_mod_sqr_nist163(_z1, _z1);
+ /* Multiply w/o reduction */
+ BN_to_XMM_3term(_t1, x->d);
+ XMM_GF2m_3x3_mul(_t3, _t1, _t1);
+ /* Add the two double-sized numbers and reduce */
+ XMM_GF2m_add_3term(_t3, _t3, _t2);
+ XMM_GF2m_mod_nist163(_x1, _t3);
+ /* MDOUBLE */
+ XMM_GF2m_mod_sqr_nist163(_x2, _x2);
+ XMM_GF2m_mod_sqr_nist163(_z2, _z2);
+ XMM_GF2m_add_2term(_t1, _z2, _x2);
+ XMM_GF2m_mod_mul_nist163(_z2, _z2, _x2);
+ XMM_GF2m_mod_sqr_nist163(_x2, _t1);
+ /* Unveil data */
+ XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);
+ /* Store results */
+ XMM_to_BN_3term(_x1->d, _tx1);
+ XMM_to_BN_3term(_z1->d, _tz1);
+ XMM_to_BN_3term(_x2->d, _tx2);
+ XMM_to_BN_3term(_z2->d, _tz2);
+ ret = 1;
+ return ret;
+
+ int BN_GF2m_Maddle_xmm_nist163r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c) {
+ /* Init */
+ int ret = 0;
+ __m128i _t1[2], _t2[3], _t3[3], _x1[2], _z1[2], _x2[2], _z2[2];
+ __m128i _tx1[2], _tz1[2], _tx2[2], _tz2[2];
+ /* Load */
+ BN_to_XMM_3term(_tx1, x->d);
+ BN_to_XMM_3term(_tz1, x->d);
+ BN_to_XMM_3term(_tx2, z2->d);
+ BN_to_XMM_3term(_tz2, z2->d);
+ /* Data veiling */
+ XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
+ /* MADD */
+ XMM_GF2m_mod_mul_nist163(_x1, _x1, _z2);
+ XMM_GF2m_mod_mul_nist163(_z1, _z1, _x2);
+ /* Multiply w/o reduction */
+ XMM_GF2m_3x3_mul(_t2, _x1, _z1);
+ XMM_GF2m_add_2term(_x1, _x1, _z1);
+ XMM_GF2m_mod_sqr_nist163(_z1, _z1);
+ /* Multiply w/o reduction */
+ BN_to_XMM_3term(_t1, x->d);
+ XMM_GF2m_3x3_mul(_t3, _t1, _t1);
+ /* Add the two double-sized numbers and reduce */
+ XMM_GF2m_add_3term(_t3, _t3, _t2);
+ XMM_GF2m_mod_nist163(_x1, _t3);
/* MDOUBLE */

XMM_GF2m_mod_sqr_nist163(_x2, _x2);
XMM_GF2m_mod_sqr_nist163(_z2, _z2);

BN_to_XMM_3term(_t1, c->d);
XMM_GF2m_mod_nist163(_t1, _z2, _t1);

XMM_GF2m_mod_nist163(_s2, _s2, _x2);
XMM_GF2m_add_2term(_x2, _x2, _t1);
XMM_GF2m_mod_sqr_nist163(_x2, _x2);

/* Unveil data */

XMM_GF2m_veil_2term(_tx1, _txz1, _tx2, _txz2, _x1, _z1, _x2, _z2, k);

/* Store results */

XMM_to_BN_3term(x1->d, _tx1);
XMM_to_BN_3term(x2->d, _tx2);

ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_sect193r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c)
{

/* Init */

int ret = 0;
__m128i _t1[2], _t2[4], _t3[4], _x1[2], _z1[2], _x2[2], _z2[2];

/* Load */

BN_to_XMM_4term(_tx1, x1->d);
BN_to_XMM_4term(_tz1, z1->d);
BN_to_XMM_4term(_tz2, z2->d);
BN_to_XMM_4term(_tx2, x2->d);

/* Data veiling */

XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

/* ADD */

XMM_GF2m_mod_nulsect193(_x1, _x1, _x2);
XMM_GF2m_mod_nulsect193(_z1, _z1, _x2);

/* Multiply w/o reduction */

XMM_GF2m_4x4_mul(_t2, _x1, _z1);
XMM_GF2m_4x4_mul(_t1, _z1, _x1);
XMM_GF2m_mod_sqrsect193(_z1, _z1);

/* Multiply w/o reduction */

BN_to_XMM_4term(_t_z1, x->d);
XMM_GF2m_4x4_mul(_t3, _z1, _t1);

/* Add the two double-sized numbers and reduce */

XMM_GF2m_add_4term(_t3, _t3, _t2);
XMM_GF2m_mod_sect193(_x1, _t3);

/* MDOUBLE */

XMM_GF2m_mod_sqrsect193(_x2, _x2);
XMM_GF2m_mod_sqrsect193(_z2, _z2);

BN_to_XMM_4term(_t1, c->d);
XMM_GF2m_mod_nulsect193(_t1, _z2, _t1);

XMM_GF2m_mod_nulsect193(_s2, _s2, _x2);
XMM_GF2m_add_2term(_x2, _x2, _t1);
XMM_GF2m_mod_sqrsect193(_s2, _s2);

/* Unveil data */

XMM_GF2m_veil_2term(_tx1, _txz1, _tx2, _txz2, _x1, _z1, _x2, _z2, k);

/* Store results */

XMM_TO_BN_4TERM(x1->d, _tx1);
XMM_TO_BN_4TERM(x2->d, _tx2);
```c
+ XMM_to_BN_4term(z2->d, _tz2);
+ ret = 1;
+ return ret;
+
+ int BN_GF2m_Haddl_xmm_nist233k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
+ {
+   /* Init */
+   int ret = 0;
+   __m128i _t1[2], _t2[4], _t3[4], _x1[2], _z1[2], _x2[2], _z2[2];
+   __m128i _tx1[2], _tx2[2], _tz1[2], _tz2[2];
+   /* Load */
+   BN_to_XMM_4term(_tx1, x1->d);
+   BN_to_XMM_4term(_tx2, x2->d);
+   /* Data veiling */
+   XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
+   /* MADD */
+   XMM_GF2m_mod_mul_nist233(_x1, _z1, _x2);
+   XMM_GF2m_mod_mul_nist233(_z1, _x1, _x2);
+   /* Add the two double-sized numbers and reduce */
+   XMM_GF2m_add_4term(_t3, _x2, _t1);
+   XMM_GF2m_add_4term(_t3, _x1, _z1);
+   /* MDOUBLE */
+   XMM_GF2m_mod_sqr_nist233(_z1, _x1, _t3);
+   XMM_GF2m_mod_sqr_nist233(_x2, _z1, _t3);
+   /* Unveil data */
+   XMM_GF2m_veil_2term(_tx1, _tx2, _tz1, _tz2, _x1, _x2, _z1, _z2, k);
+   /* Store results */
+   XMM_to_BN_4term(x1->d, _tx1);
+   XMM_to_BN_4term(x2->d, _tx2);
+   XMM_to_BN_4term(z1->d, _tz1);
+   XMM_to_BN_4term(z2->d, _tz2);
+   ret = 1;
+   return ret;
+ }
+
+ int BN_GF2m_Haddr_xmm_nist233r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c)
+ {
+   /* Init */
+   int ret = 0;
+   __m128i _t1[2], _t2[4], _t3[4], _x1[2], _z1[2], _x2[2], _z2[2];
+   __m128i _tx1[2], _tx2[2], _tz1[2], _tz2[2];
+   /* Load */
+   BN_to_XMM_4term(_tx1, x1->d);
+   BN_to_XMM_4term(_tx2, x2->d);
```
/* Data veiling */
XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

/* MADD */
XMM_GF2m_mod_mul_nist233(_x1, _x1, _z2);
XMM_GF2m_mod_mul_nist233(_z1, _z1, _x2);

/* Multiply w/o reduction */
XMM_GF2m_4x4_mul(_t2, _x1, _z1);
XMM_GF2m_add_2term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist233(_z1, _z1);

/* Multiply w/o reduction */
BN_to_XMM_4term(_t1, x->d);
XMM_GF2m_4x4_mul(_t3, _z1, _t1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_4term(_t3, _t3, _t2);
XMM_GF2m_mod_nist233(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_nist233(_x2, _x2);
XMM_GF2m_mod_sqr_nist233(_z2, _z2);

BN_to_XMM_4term(_t1, c->d);
XMM_GF2m_4x4_mul(_t3, _z1, _t1);

/* Multiply w/o reduction */
XMM_GF2m_add_4term(_z2, _z2, _x2);
XMM_GF2m_mod_mul_nist233(_t1, _z2, _t1);
XMM_GF2m_mod_mul_nist233(_z2, _z2, _x2);

/* Unveil data */
XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);

/* Store results */
XMM_to_BN_4term(x1->d, _tx1);
XMM_to_BN_4term(z1->d, _tz1);
XMM_to_BN_4term(x2->d, _tz2);

ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_sect239k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
    int ret = 0;
    __m128i _t1[2], _t2[4], _t3[4], _t4[4], _x1[2], _z1[2], _x2[2], _z2[2];
    __m128i _tx1[2], _tz1[2], _tx2[2], _tz2[2];
    int ret = 0;
    BN_to_XMM_4term(_tx1, _x1->d);
    BN_to_XMM_4term(_tz1, _z1->d);
    BN_to_XMM_4term(_tx2, _z2->d);
    BN_to_XMM_4term(_tz2, _x2->d);

    /* Data veiling */
    XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
    /* MADD */
    XMM_GF2m_mod_mul_nist239(_x1, _x1, _z2);
    XMM_GF2m_mod_mul_nist239(_z1, _z1, _x2);

    /* Multiply w/o reduction */
    XMM_GF2m_6x4_mul(_t2, _x1, _z1);
    XMM_GF2m_add_2term(_z1, _z1, _x1);
    XMM_GF2m_mod_sqr_nist239(_z1, _z1);

    /* Multiply w/o reduction */
    BN_to_XMM_4term(_x1, x->d);
    XMM_GF2m_4x4_mul(_t3, _z1, _t1);
/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_4term(_t3, _t3, _t2);
XMM_GF2m_mod_sector239(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_sector239(_x2, _x2);
XMM_GF2m_mod_sqr_sector239(_z2, _z2);
XMM_GF2m_add_2term(_t1, _x2, _z2);
XMM_GF2m_mod_sector239(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_sector239(_x2, _t1);

/* Unveil data */
XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);

/* Store results */
XMM_to_BN_4term(x1, _tx1);
XMM_to_BN_4term(z1, _tz1);
XMM_to_BN_4term(x2, _tx2);
XMM_to_BN_4term(z2, _tz2);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist283k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k) {
/* Init */
int ret = 0;
__m128i _t1[3], _t2[5], _t3[5], _x1[3], _z1[3], _x2[3], _z2[3];
__m128i _tx1[3], _tz1[3], _tx2[3], _tz2[3];
/* Load */
BN_to_XMM_5term(_tx1, x1, x1, x2, _tx1);
BN_to_XMM_5term(_tz1, z1, z1, z2, _tz1);
BN_to_XMM_5term(_tz2, z2, z2, z2, _tz2);
/* Data veiling */
XMM_GF2m_veil_3term(_x1, _z1, _x2, _z2, _tx1, _tx2, _tx1, _tz2, _tz2, k);
/* MADD */
XMM_GF2m_mod_mul_nist283(_x1, _x1, _z2);
XMM_GF2m_mod_mul_nist283(_z1, _z1, _x2);
/* Multiply w/o reduction */
XMM_GF2m_5x5_mul(_t2, _x1, _z1);
XMM_GF2m_add_3term(_z1, _x1, _z1);
XMM_GF2m_mod_sqr_nist283(_z1, _z1);
/* Multiply w/o reduction */
BN_to_XMM_5term(_t1, x, x, z1, _t1);
XMM_GF2m_5x5_mul(_t3, _z1, _z1, _t1);
/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_5term(_t3, _t3, _t2);
XMM_GF2m_mod_nist283(_x1, _t3);
/* MDOUBLE */
XMM_GF2m_mod_sqr_nist283(_x2, _x2);
XMM_GF2m_mod_sqr_nist283(_z2, _z2);
XMM_GF2m_add_3term(_t1, _z2, _x2);
XMM_GF2m_mod_nist283(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_nist283(_x2, _t1);
/* Unveil data */
XMM_GF2m_veil_3term(_tx1, _tx2, _tx1, _tz2, _x1, _z1, _x2, _z2, k);
/* Store results */
XMM_to_BN_5term(x1, _tx1);
XMM_to_BN_5term(z1, _tz1);
XMM_to_BN_5term(x2, _tx2);
```c
+ int BN_GF2m_Maddle_xmm_nist283r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c)
+ {
+    /* Init */
+    int ret = 0;
+    __m128i _t1[3], _t2[6], _t3[6], _x1[3], _z1[3], _x2[3], _z2[3];
+    __m128i _tx1[3], _tx2[3], _tz1[3];
+    /* Load */
+    BN_to_XMM_5term(_tx1, x1->d);
+    BN_to_XMM_5term(_tz1, z1->d);
+    BN_to_XMM_5term(_tx2, x2->d);
+    /* Data veiling */
+    XMM_GF2m_veil_3term(_x1, _z1, _x2, _z2, _tx1, _tx2, _tz1, _tz2, k);
+    /* MADD */
+    XMM_GF2m_mod_mul_nist283(_x1, _x1, _z2);
+    XMM_GF2m_mod_mul_nist283(_z1, _z1, _x2);
+    /* Multiply w/o reduction */
+    XMM_GF2m_5x5_mul(_t1, _t2, _x1, _z1);
+    XMM_GF2m_add_3term(_z1, _z1, _x1);
+    /* Multiply w/o reduction */
+    BN_to_XMM_5term(_t1, x->d);
+    XMM_GF2m_5x5_mul(_t3, _z1, _t1);
+    /* Add the two double-sized numbers and reduce */
+    XMM_GF2m_add_5term(_t3, _t3, _t2);
+    XMM_GF2m_mod_nist283(_x1, _t3);
+    /* MDOUBLE */
+    XMM_GF2m_mod_sqr_nist283(_x2, _x2);
+    XMM_GF2m_mod_sqr_nist283(_z2, _z2);
+    BN_to_XMM_5term(_t1, c->d);
+    XMM_GF2m_mod_nist283(_t1, _z2, _t1);
+    XMM_GF2m_mod_nist283(_t2, _z2, _t2);
+    XMM_GF2m_add_3term(_x2, _z2, _t1);
+    XMM_GF2m_mod_sqr_nist283(_x2, _z2);
+    /* Unveil data */
+    XMM_GF2m_veil_3term(_tx1, _tx2, _tz1, _tz2, _x1, _z1, _x2, _z2, k);
+    /* Store results */
+    BN_to_XMM_5term(x1->d, _tx1);
+    BN_to_XMM_5term(x2->d, _tx2);
+    ret = 1;
+    return ret;
+ }
```

```c
+ int BN_GF2m_Maddle_xmm_nist409k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
+ {
+    /* Init */
+    int ret = 0;
+    __m128i _t1[4], _t2[7], _t3[7], _x1[4], _z1[4], _x2[4], _z2[4];
+    __m128i _tx1[4], _tx2[4], _tz1[4], _tz2[4];
+    /* Load */
+    BN_to_XMM_7term(_tx1, x1->d);
+    BN_to_XMM_7term(_tx2, x2->d);
```

108
/* Data veiling */
XMM_GF2m_veil_4term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

/* ADD */
XMM_GF2m_mod_mul_nist409(_x1, _x1, _x2);
XMM_GF2m_mod_mul_nist409(_z1, _z1, _x2);

/* Multiply w/o reduction */
XMM_GF2m Tx7_mul(_t2, _x1, _z1);
XMM_GF2m_add_4term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist409(_z1, _z1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_7term(_t3, _t3, _t2);
XMM_GF2m_mod_nist409(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_nist409(_x2, _x2);
XMM_GF2m_mod_sqr_nist409(_z2, _z2);
XMM_GF2m_add_4term(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_nist409(_x2, _x2);

/* Unveil data */
XMM_GF2m_veil_4term(_tx1, _tx1, _tx2, _tz1, _tx2, _tz2, _tx1, _tz2, _tx2, _tz2, k);

/* Store results */
XMM_to_BN_7term(_t1->d, _tx1);
XMM_to_BN_7term(_t2->d, _tx1);
XMM_to_BN_7term(_t3->d, _tx2);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist409r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c) {
/* Init */
int ret = 0;
++a128i, _t1[4], _t2[7], _t3[7], _x1[4], _z1[4], _x2[4], _z2[4];
++a128i, _tx1[4], _tx2[4], _tz1[4], _tx2[4];

/* Load */
BN_to_XMM_7term(_tx1, x->d);
BN_to_XMM_7term(_tx2, x->d);
BN_to_XMM_7term(_t1, x->d);
BN_to_XMM_7term(_t2, x->d);

/* Data veiling */
XMM_GF2m_veil_4term(_x1, _z1, _x2, _z2, _tx1, _tx1, _tx2, _tx2, _tx2, _tx2, k);

/* ADD */
XMM_GF2m_mod_mul_nist409(_x1, _x1, _x2);
XMM_GF2m_mod_mul_nist409(_z1, _z1, _x2);

/* Multiply w/o reduction */
XMM_GF2m Tx7_mul(_t2, _x1, _z1);
XMM_GF2m_add_4term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist409(_z1, _z1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_7term(_t3, _t3, _t2);
```c
/* MDOUBLE */
XMM_GF2m_mod_sqr_nist409(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_nist409(_x2, _x2);
XMM_GF2m_mod_sqr_nist409(_z2, _z2);
BX_to_XMM_7term(_t1, c->d);
XMM_GF2m_mod_nist409(_t1, _x2, _t1);

XMM_GF2m_mod_nist409(_z2, _z2, _x2);
XMM_GF2m_add_4term(_x2, _x2, _t1);
XMM_GF2m_mod_sqr_nist409(_x2, _x2);
/* Unveil data */
XMM_GF2m_veil_4term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);
/* Store results */
XMM_to_BN_7term(_tx1, x1->d);
XMM_to_BN_7term(_tx2, x2->d);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist571k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
/* Init */
int ret = 0;
__m128i _t1[5], _t2[9], _t3[9], _x1[5], _z1[5], _x2[5], _z2[5];
__m128i _tx1[5], _tz1[5], _tx2[5], _tz2[5];
/* Load */
BX_to_XMM_9term(_tx1, x1->d);
BX_to_XMM_9term(_tx2, x2->d);
BN_to_XMM_9term(_tz1, z1->d);
BN_to_XMM_9term(_tz2, z2->d);
/* Data veiling */
XMM_GF2m_veil_8term(_tx1, _tx2, _t1, _tz1, _tz2, _tx1, _tx2, _tz2, k);
/* MADD */
XMM_GF2m_mod_mul_nist571(_x1, _x1, _z2);
XMM_GF2m_mod_mul_nist571(_z1, _z1, _x2);
XMM_GF2m_9x9_mul(_t2, _x1, _z1);
XMM_GF2m_add_5term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist571(_z1, _z1);
/* Multiply w/o reduction */
XMM_GF2m_9x9_mul(_t1, _x1, _z1);
XMM_GF2m_add_5term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist571(_z1, _z1);
/* Multiply w/o reduction */
BX_to_XMM_9term(_tx1, x1->d);
XMM_GF2m_9x9_mul(_t3, _x1, _t1);
/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_8term(_t3, _t3, _t2);
XMM_GF2m_mod_nist571(_x1, _t3);
/* MDOUBLE */
XMM_GF2m_mod_sqr_nist571(_x2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _z2);
XMM_GF2m_add_5term(_t1, _x2, _z2);
XMM_GF2m_mod_nist571(_x2, _z2, _x2);
XMM_GF2m_mod_nist571(_x2, _z2, _t1);
/* Unveil data */
XMM_GF2m_veil_8term(_tx1, _tx2, _tz1, _tx2, _tx1, _tz1, _tx2, _tz2, k);
/* Store results */
BX_to_BN_9term(x1->d, _tx1);
BX_to_BN_9term(x2->d, _tx2);
```

+ XMM_to_BN_9term(z2->d, _tz2);
+ ret = 1;
+ return ret;
+ }

int BN_GF2m_Maddle_xmm_nist571r(const BIGNUM *x, BIGNUM *z1, BIGNUM *z2,
+ const BIGNUM *x1, const BIGNUM *x2, BNULONG k, const BIGNUM *c)
+ {
+ /* Init */
+ int ret = 0;
+ __m128i _t1[5], _t2[9], _t3[9], _x1[5], _z1[5], _x2[5], _z2[5];
+ __m128i _tx1[5], _tx2[9], _tx3[9],
+ /* Load */
+ BN_to_XMM_9term(_tx1, x1->d);
+ BN_to_XMM_9term(_tx2, x2->d);
+ BN_to_XMM_9term(_tx3, x3->d);
+ /* Data veiling */
+ XMM_GF2m_veil_5term(_x1, _z1, _x2, _z2, _x1, _x2, _z1, _z2, k);
+ /* MADD */
+ XMM_GF2m_mod_mul_nist571(_x1, _x1, _z2);
+ XMM_GF2m_mod_mul_nist571(_z1, _z1, _x2);
+ /* Multiply w/o reduction */
+ XMM_GF2m_9x9_mul(_t2, _x1, _z1);
+ XMM_GF2m_add_5term(_x1, _z1, _x1);
+ XMM_GF2m_mod_sqr_nist571(_x1, _x1);
+ /* Multiply w/o reduction */
+ BN_to_XMM_9term(_t1, x->d);
+ XMM_GF2m_9x9_mul(_t3, _x1, _z1);
+ /* Add the two double-sized numbers and reduce */
+ XMM_GF2m_add_9term(_t3, _t3, _t2);
+ XMM_GF2m_mod_nist571(x1, _t3);
+ /* MDDOUBLE */
+ XMM_GF2m_mod_sqr_nist571(_x2, _z2);
+ XMM_GF2m_mod_sqr_nist571(_z2, _x2);
+ BN_to_XMM_9term(_t1, c->d);
+ XMM_GF2m_mod_nist571(_z2, _z2, _x2);
+ XMM_GF2m_mod_nist571(_z2, _z2, _t1);
+ XMM_GF2m_add_5term(_x2, _z2, _x2);
+ XMM_GF2m_mod_sqr_nist571(_x2, _x2);
+ /* Unveil data */
+ XMM_GF2m_veil_5term(_tx1, _tx2, _tx3, _tx1, _tx2, _tx3, _tx3, k);
+ /* Store results */
+ XMM_to_BN_9term(x1->d, _tx1);
+ XMM_to_BN_9term(z1->d, _tx1);
+ XMM_to_BN_9term(x2->d, _tx2);
+ XMM_to_BN_9term(z2->d, _tx2);
+ ret = 1;
+ return ret;
+ }
#endif

*/
/* using Intel SSE (Compiler Intrinsics) and PCLMUL */

#define OPENSSL_FAST_ECM

/* Functions for binary field arithmetic to allow constant sizes of BIGNUM. */

int BN_GF2m_const_init (BIGNUM *a, int word_size);
int BN_GF2m_copy (BIGNUM *a, const BIGNUM *b);
int BN_GF2m_const_add (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_const_add (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_const_and (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_const_cnp_zero (const BIGNUM *a);
int BN_GF2m_const_cnp_one (const BIGNUM *a);
int BN_GF2m_const_cnp_eq (const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_const_copy (BIGNUM *a, const BIGNUM *b);
int BN_GF2m_const_setone (BIGNUM *a);
int BN_GF2m_const_setword (BIGNUM *a, BN_ULONG b);
int BN_GF2m_const_setmask (BIGNUM *a, BIGNUM *b, int k);

/* Fast implementation of field arithmetic's */

int BN_GF2m_sqr_xmm_nist163 (BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect193 (BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_nist233 (BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect239 (BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_nist283 (BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_nist409 (BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_nist571 (BIGNUM *z, const BIGNUM *a);

int BN_GF2m_mul_xmm_nist163 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect193 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_nist233 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect239 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_nist283 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_nist409 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_nist571 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);

int BN_GF2m_div_xmm_nist163 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect193 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_nist233 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect239 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_nist283 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_nist409 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_nist571 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b);

/* Fast implementation of Maddle & Mdouble for EC Montgomery multiplication */

int BN_GF2m_Maddle_xmm_nist163k (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist163r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_sect193r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist233k (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist233r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_sect239k (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_sect239r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist283k (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist283r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist409k (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist409r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist571k (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist571r (const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);

#endif
/* faster mod functions for the 'NIST primes'

diff -urN openssl-orig/crypto/bn/Makefile openssl-work/crypto/bn/Makefile
--- openssl-orig/crypto/bn/Makefile 2013-02-11 16:26:04.000000000 +0100
+++ openssl-work/crypto/bn/Makefile 2013-05-12 05:30:29.415829953 +0200
@@ -26,13 +26,13 @@
   bn_print.c bn_rand.c bn_shift.c bn_word.c bn_blind.c \
   bn_kron.c bn_sqrt.c bn_gcd.c bn_prime.c bn_err.c bn_sqrt.c bn_asm.c \
   bn_recip.c bn_mov.c bn_mpi.c bn_exp2.c bn_gf2m.c bn_nist.c \
-  bn_depr.c bn_const.c bn_x931p.c 
+  bn_depr.c bn_const.c bn_x931p.c bn_gf2m_xmm.c 

LIBOBJ= 
   bn_add.o bn_div.o bn_exp.o bn_lib.o bn_ctx.o bn_mul.o bn_mod.o \
   bn_print.o bn_rand.o bn_shift.o bn_word.o bn_blind.o \
   bn_kron.o bn_sqrt.o bn_gcd.o bn_prime.o bn_err.o bn_sqrt.o $(BN_ASM) \
   bn_recip.o bn_mov.o bn_mpi.o bn_exp2.o bn_gf2m.o bn_nist.o \
-  bn_depr.o bn_const.o bn_x931p.o 
+  bn_depr.o bn_const.o bn_x931p.o bn_gf2m_xmm.o 

SRC= $(LIBSRC)

diff -urN openssl-orig/crypto/ec/ec2_nist.c openssl-work/crypto/ec/ec2_nist.c
--- openssl-orig/crypto/ec/ec2_nist.c 1970-01-01 01:00:00.000000000 +0100
+++ openssl-work/crypto/ec/ec2_nist.c 2013-05-12 05:01:59.269208000 +0200
@@ -0,0 +1,1060 @@
+
+/* crypto/ec/ec2_nist.c */
+
+/* Written by Manuel Bluhm for the OpenSSL project.
+* */
+/* This product includes software developed by the OpenSSL Project
+* for use in the OpenSSL Toolkit. (http://www.openssl.org/)*/ 
+/* The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
+* endorse or promote products derived from this software without
+* prior written permission. For written permission, please contact
+* openssl-core@openssl.org.
+*/ 
+/* Products derived from this software may not be called "OpenSSL"
+* nor may "OpenSSL" appear in their names without prior written
+* permission of the OpenSSL Project.
+*/ 
+/* Redistributions of any form whatsoever must retain the following
+* acknowledgment:
+* "This product includes software developed by the OpenSSL Project
+* for use in the OpenSSL Toolkit (http://www.openssl.org/)">*/ 
+/* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ‘AS IS’ AND ANY
+* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
+* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
+* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL The OpenSSL Project OR
+* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
+* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
+* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
+* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
+* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
+* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
+* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
+* OF THE POSSIBILITY OF SUCH DAMAGE.
+*/ 
+/* This product includes cryptographic software written by Eric Young 
+* (eay@cryptsoft.com). This product includes software written by Tim
+* Hudson (tjh@cryptsoft.com). 
+*/
+*/
+*/
```c
#include "ec_lcl.h"
#include <openssl/err.h>

#ifdef OPENSSL_NO_EC2M
    #ifdef OPENSSL_FIPS
        #ifdef OPENSSL_FAST_EC2M
            #undef OPENSSL_FAST_EC2M
        #endif
    #endif
    #endif
#endif

const EC_METHOD *EC_GF2m_nist163_method(void)
{
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT,
        NID_X9_62_characteristic_two_field,
        ec_GF2m_simple_group_init,
        ec_GF2m_simple_group_finish,
        ec_GF2m_simple_group_copy,
        ec_GF2m_nist_group_set_curve,
        ec_GF2m_simple_group_get_curve,
        ec_GF2m_simple_group_get_degree,
        ec_GF2m_simple_group_check_discriminant,
        ec_GF2m_simple_point_init,
        ec_GF2m_simple_point_finish,
        ec_GF2m_simple_point_clear_finish,
        ec_GF2m_nist_point_copy,
        ec_GF2m_nist_point_set_to_infinity,
        0 /* set_Jprojective_coordinates_GFp */,
        0 /* get_Jprojective_coordinates_GFp */,
        ec_GF2m_nist_point_set_affine_coordinates,
        ec_GF2m_nist_point_get_affine_coordinates,
        0, 0,
        ec_GF2m_nist_add,
        ec_GF2m_nist_dbl,
        ec_GF2m_nist_invert,
        ec_GF2m_nist_is_at_infinity,
        ec_GF2m_nist_is_on_curve,
        ec_GF2m_nist_cmp,
        ec_GF2m_nist_make_affine,
        ec_GF2m_nist_points_make_affine,
        /* the following method function is defined in ec3_nist_mult.c */
        ec_GF2m_nist_mul,
        ec_GF2m_precompute_mult,
        ec_GF2m_have_precompute_mult,
        ec_GF2m_nist163_field_mult,
        ec_GF2m_nist163_field_sqr,
        ec_GF2m_nist163_field_div,
        0 /* field_encode */,
        0 /* field_decode */,
        0 /* field_set_to_one */
    };
    return &ret;
}

const EC_METHOD *EC_GF2m_sect193_method(void)
{
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT,
        NID_X9_62_characteristic_two_field,
        ec_GF2m_simple_group_init,
        ec_GF2m_simple_group_finish,
        ec_GF2m_simple_group_copy,
        ec_GF2m_nist_group_set_curve,
        ec_GF2m_nist_group_set_curve,
        ec_GF2m_nist_group_get_curve,
        ec_GF2m_nist_group_get_degree,
        ec_GF2m_nist_group_check_discriminant,
        ec_GF2m_nist_point_init,
        ec_GF2m_nist_point_finish,
        ec_GF2m_nist_point_clear_finish,
        ec_GF2m_nist163_field_mult,
        ec_GF2m_nist163_field_sqr,
        ec_GF2m_nist163_field_div,
        0 /* field_encode */,
        0 /* field_decode */,
        0 /* field_set_to_one */
    };
    return &ret;
}
```

const EC_METHOD *EC_GF2m_nist233_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT, NID_X9_62_characteristic_two_field, ec_GF2m_simple_group_init, ec_GF2m_simple_group_finish, ec_GF2m_nist_group_copy, ec_GF2m_nist_group_set_curve, ec_GF2m_nist_simple_group_get_curve, ec_GF2m_nist_simple_group_get_degree, ec_GF2m_nist_simple_group_check_discriminant, ec_GF2m_nist_simple_point_init, ec_GF2m_nist_simple_point_finish, ec_GF2m_nist_simple_point_clear_finish, ec_GF2m_nist_point_copy, ec_GF2m_nist_point_set_to_infinity, 0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */, ec_GF2m_nist_point_set_affine_coordinates, ec_GF2m_nist_point_get_affine_coordinates, 0,0,0, ec_GF2m_nist_mul, ec_GF2m_have_precompute_mult, ec_GF2m_precompute_mult, ec_GF2m_nist_233_field_mul, ec_GF2m_nist_233_field_sqr, ec_GF2m_nist_233_field_div, 0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */};
    return &ret;
}

const EC_METHOD *EC_GF2m_nist235_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT, NID_X9_62_characteristic_two_field, ec_GF2m_simple_group_init, ec_GF2m_simple_group_finish, ec_GF2m_nist_group_copy, ec_GF2m_nist_group_set_curve, ec_GF2m_nist_simple_group_get_curve, ec_GF2m_nist_simple_group_get_degree, ec_GF2m_nist_simple_group_check_discriminant, ec_GF2m_nist_simple_point_init, ec_GF2m_nist_simple_point_finish, ec_GF2m_nist_simple_point_clear_finish, ec_GF2m_nist_point_copy, ec_GF2m_nist_point_set_to_infinity, 0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */, ec_GF2m_nist_point_set_affine_coordinates, ec_GF2m_nist_point_get_affine_coordinates, 0,0,0, ec_GF2m_nist_mul, ec_GF2m_have_precompute_mult, ec_GF2m_precompute_mult, ec_GF2m_nist_235_field_mul, ec_GF2m_nist_235_field_sqr, ec_GF2m_nist_235_field_div, 0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */};
    return &ret;
}

const EC_METHOD *EC_GF2m_nist239_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT, NID_X9_62_characteristic_two_field, ec_GF2m_simple_group_init, ec_GF2m_simple_group_finish, ec_GF2m_nist_group_copy, ec_GF2m_nist_group_set_curve, ec_GF2m_nist_simple_group_get_curve, ec_GF2m_nist_simple_group_get_degree, ec_GF2m_nist_simple_group_check_discriminant, ec_GF2m_nist_simple_point_init, ec_GF2m_nist_simple_point_finish, ec_GF2m_nist_simple_point_clear_finish, ec_GF2m_nist_point_copy, ec_GF2m_nist_point_set_to_infinity, 0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */, ec_GF2m_nist_point_set_affine_coordinates, ec_GF2m_nist_point_get_affine_coordinates, 0,0,0, ec_GF2m_nist_mul, ec_GF2m_have_precompute_mult, ec_GF2m_precompute_mult, ec_GF2m_nist_239_field_mul, ec_GF2m_nist_239_field_sqr, ec_GF2m_nist_239_field_div, 0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */};
    return &ret;
}

const EC_METHOD *EC_GF2m_nist251_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT, NID_X9_62_characteristic_two_field, ec_GF2m_simple_group_init, ec_GF2m_simple_group_finish, ec_GF2m_nist_group_copy, ec_GF2m_nist_group_set_curve, ec_GF2m_nist_simple_group_get_curve, ec_GF2m_nist_simple_group_get_degree, ec_GF2m_nist_simple_group_check_discriminant, ec_GF2m_nist_simple_point_init, ec_GF2m_nist_simple_point_finish, ec_GF2m_nist_simple_point_clear_finish, ec_GF2m_nist_point_copy, ec_GF2m_nist_point_set_to_infinity, 0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */, ec_GF2m_nist_point_set_affine_coordinates, ec_GF2m_nist_point_get_affine_coordinates, 0,0,0, ec_GF2m_nist_mul, ec_GF2m_have_precompute_mult, ec_GF2m_precompute_mult, ec_GF2m_nist_251_field_mul, ec_GF2m_nist_251_field_sqr, ec_GF2m_nist_251_field_div, 0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */};
    return &ret;
}

const EC_METHOD *EC_GF2m_nist263_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT, NID_X9_62_characteristic_two_field, ec_GF2m_simple_group_init, ec_GF2m_simple_group_finish, ec_GF2m_nist_group_copy, ec_GF2m_nist_group_set_curve, ec_GF2m_nist_simple_group_get_curve, ec_GF2m_nist_simple_group_get_degree, ec_GF2m_nist_simple_group_check_discriminant, ec_GF2m_nist_simple_point_init, ec_GF2m_nist_simple_point_finish, ec_GF2m_nist_simple_point_clear_finish, ec_GF2m_nist_point_copy, ec_GF2m_nist_point_set_to_infinity, 0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */, ec_GF2m_nist_point_set_affine_coordinates, ec_GF2m_nist_point_get_affine_coordinates, 0,0,0, ec_GF2m_nist_mul, ec_GF2m_have_precompute_mult, ec_GF2m_precompute_mult, ec_GF2m_nist_263_field_mul, ec_GF2m_nist_263_field_sqr, ec_GF2m_nist_263_field_div, 0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */};
    return &ret;
}
const EC_METHOD *EC_GF2m_sect239_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT,
        NID_X9_62_characteristic_two_field,
        ec_GF2m_simple_group_init,
        ec_GF2m_simple_group_finish,
        ec_GF2m_simple_group_clear_finish,
        ec_GF2m_nist_group_copy,
        ec_GF2m_nist_group_set_curve,
        ec_GF2m_nist_group_get_curve,
        ec_GF2m_nist_group_check_discriminant,
        ec_GF2m_nist_point_init,
        ec_GF2m_nist_point_finish,
        ec_GF2m_nist_point_clear_finish,
        ec_GF2m_nist_point_copy,
        ec_GF2m_nist_point_set_to_infinity,
        0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */,
        ec_GF2m_nist_point_set_affine_coordinates,
        ec_GF2m_nist_point_get_affine_coordinates,
        0, 0, 0,
        ec_GF2m_nist_add,
        ec_GF2m_nist_dbl,
        ec_GF2m_nist_invert,
        ec_GF2m_nist_is_at_infinity,
        ec_GF2m_nist_is_on_curve,
        ec_GF2m_nist_cmp,
        ec_GF2m_nist_make_affine,
        ec_GF2m_nist_points_make_affine,
        ec_GF2m_nist_field_mul,
        ec_GF2m_nist_field_sqr,
        ec_GF2m_nist_field_div,
        0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */);
    return &ret;
}

const EC_METHOD *EC_GF2m_nist283_method(void) {
    static const EC_METHOD ret = {
        EC_FLAGS_DEFAULT_OCT,
        NID_X9_62_characteristic_two_field,
        ec_GF2m_simple_group_init,
        ec_GF2m_simple_group_finish,
        ec_GF2m_simple_group_clear_finish,
        ec GF2m_nist_group_copy,
        ec_GF2m_nist_group_set_curve,
        ec_GF2m_nist_group_get_curve,
        ec_GF2m_nist_group_check_discriminant,
        ec_GF2m_nist_point_init,
        ec_GF2m_nist_point_finish,
        ec_GF2m_nist_point_clear_finish,
        ec_GF2m_nist_point_copy,
        ec_GF2m_nist_point_set_to_infinity,
        0 /* set_Jprojective_coordinates_GFp */, 0 /* get_Jprojective_coordinates_GFp */,
        ec_GF2m_nist_point_set_affine_coordinates,
        ec_GF2m_nist_point_get_affine_coordinates,
        0, 0, 0,
        ec_GF2m_nist_add,
        ec_GF2m_nist_dbl,
        ec_GF2m_nist_invert,
        ec_GF2m_nist_is_at_infinity,
        ec_GF2m_nist_is_on_curve,
        ec_GF2m_nist_cmp,
        ec_GF2m_nist_make_affine,
        ec_GF2m_nist_points_make_affine,
        ec_GF2m_nist_field_mul,
        ec_GF2m_nist_field_sqr,
        ec_GF2m_nist_field_div,
        0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */};
    return &ret;
}
/* the following method function is defined in ecGF2m_nist_mult.c */
ec_GF2m_nist_mult,
ec_GF2m_precompute_mult,
ec_GF2m_have_precompute_mult,
ec_GF2m_nist283_field_mult,
ec_GF2m_nist283_field_sqr,
ec_GF2m_nist283_field_div,
0 /* field_encode */,
0 /* field_decode */,
0 /* field_set_to_one */);
return &ret;
}

const EC_METHOD *EC_GF2m_nist409_method(void) {
static const EC_METHOD ret = {
EC_FLAGS_DEFAULT_DCT,
NID_X9_62_characteristic_two_field,
ec_GF2m_simple_group_init,
ec_GF2m_simple_group_finish,
ec_GF2m_simple_group_clear_finish,
ec_GF2m_nist_group_copy,
ec_GF2m_nist_group_set_curve,
ec_GF2m_nist_group_get_curve,
ec_GF2m_nist_group_get_degree,
ec_GF2m_nist_group_check_discriminant,
ec_GF2m_simple_point_init,
ec_GF2m_simple_point_finish,
ec_GF2m_simple_point_clear_finish,
ec_GF2m_nist_point_init,
ec_GF2m_nist_point_set_to_infinity,
0 /* set_Jprojective_coordinates_GFp */,
0 /* get_Jprojective_coordinates_GFp */,
ec_GF2m_nist_point_set_affine_coordinates,
ec_GF2m_nist_point_get_affine_coordinates,
0,0,0,
ec_GF2m_nist_add,
ec_GF2m_nist_dbl,
ec_GF2m_nist_invert,
ec_GF2m_nist_is_at_infinity,
ec_GF2m_nist_is_on_curve,
ec_GF2m_nist_cmp,
ec_GF2m_nist_make_affine,
ec_GF2m_nist_points_make_affine,
ec_GF2m_nist409_field_mult,
ec_GF2m_nist409_field_sqr,
ec_GF2m_nist409_field_div,
0 /* field_encode */,
0 /* field_decode */,
0 /* field_set_to_one */);
return &ret;
}

const EC_METHOD *EC_GF2m_nist571_method(void) {
static const EC_METHOD ret = {
EC_FLAGS_DEFAULT_DCT,
NID_X9_62_characteristic_two_field,
ec_GF2m_simple_group_init,
ec_GF2m_simple_group_finish,
ec_GF2m_simple_group_clear_finish,
ec_GF2m_nist_group_copy,
ec_GF2m_nist_group_set_curve,
ec_GF2m_nist_group_get_curve,
ec_GF2m_nist_group_get_degree,
ec_GF2m_nist_group_check_discriminant,
ec_GF2m_simple_point_init,
ec_GF2m_simple_point_finish,
ec_GF2m_simple_point_clear_finish,
ec_GF2m_nist_point_copy,
ec_GF2m_nist_point_set_to_infinity,
0 /* set_Jprojective_coordinates_GFp */,
0 /* get_Jprojective_coordinates_GFp */,
ec_GF2m_nist_point_set_affine_coordinates,
ec_GF2m_nist_point_get_affine_coordinates,
0,0,0,
ec_GF2m_nist_add,
ec_GF2m_nist_dbl,
ec_GF2m_nist_invert,
ec_GF2m_nist_is_at_infinity,
ec_GF2m_nist_is_on_curve,
ec_GF2m_nist_cmp,
ec_GF2m_nist_point_add_affine,
ec_GF2m_nist_point_make_affine,
ec_GF2m_nist_point_set_affine,
/* the following method function is defined in ec2_nist_mult.c */
ec_GF2m_nist_mul,
ec_GF2m_precompute_mult,
ec_GF2m_have_precompute_mult,
ec_GF2m_nist571_field_mul,
ec_GF2m_nist571_field_sqr,
ec_GF2m_nist571_field_div,
0 /* field_encode */, 0 /* field_decode */, 0 /* field_set_to_one */;
/* group->b */
if( ! BN_GF2m_const_init (& group->b, field_size)) goto err;
if (! BN_GF2m_copy (& group->b, b)) goto err;
ret = 1;
err:
return ret;
}

int ec_GF2m_nist_point_copy(EC_POINT *dest, const EC_POINT *src) {
if (! BN_GF2m_const_init (& dest->X, (& src->X)->top)) return 0;
if (! BN_GF2m_const_init (& dest->Y, (& src->Y)->top)) return 0;
if (! BN_GF2m_const_copy (& dest->X, &src->X)) return 0;
if (! BN_GF2m_const_copy (& dest->Y, &src->Y)) return 0;
dest->Z_is_one = src->Z_is_one;
return 1;
}

int ec_GF2m_nist_point_set_to_infinity(const EC_GROUP *group, EC_POINT *point) {
point->Z_is_one = 0;
BN_zero(&point->Z);
return 1;
}

int ec_GF2m_nist_point_set_affine_coordinates(const EC_GROUP *group, EC_POINT *point, const BIGNUM *x, const BIGNUM *y, BN_CTX *ctx) {
int field_size, ret = 0;
if (x == NULL || y == NULL) {
ECerr(EC_F_EC_GF2M_SIMPLE_POINT_SET_AFFINE_COORDINATES, ERR_R_PASSED_NULL_PARAMETER);
return 0;
}
field_size = (group->poly[0] / BN_BITS2) + 1;
if (! BN_GF2m_const_init(&point->X, field_size)) goto err;
if (! BN_GF2m_const_init(&point->Y, field_size)) goto err;
if (! BN_GF2m_copy(&point->X, x)) goto err;
BN_set_negative(&point->X, 0);
if (! BN_GF2m_copy(&point->Y, y)) goto err;
BN_set_negative(&point->Y, 0);
if (! BN_copy(&point->Z, BN_value_one())) goto err;
point->Z_is_one = 1;
ret = 1;
err:
return ret;
}

int ec_GF2m_nist_point_get_affine_coordinates(const EC_GROUP *group, const EC_POINT *point, BIGNUM *x, BIGNUM *y, BN_CTX *ctx) {
int field_size, ret = 0;
field_size = (group->poly[0] / BN_BITS2) + 1;
if (EC_POINT_is_at_infinity(group, point)) {
...
ECerr(EC_F_EC_GF2M_SIMPLE_POINT_GET_AFFINE_COORDINATES, EC_R_POINT_AT_INFINITY);
  return 0;
}

if (BN_cmp(&point->Z, BN_value_one()))
{
  ECerr(EC_F_EC_GF2M_SIMPLE_POINT_GET_AFFINE_COORDINATES,
       ERR_R_SHOULD_NOT_HAVE_BEEN_CALLED);
  return 0;
}

if (x != NULL)
{
  BN_GF2m_const_init(x, field_size); x->neg = 0;
  BN_set_negative(x, 0);
}

if (y != NULL)
{
  BN_GF2m_const_init(y, field_size); y->neg = 0;
  BN_set_negative(y, 0);
}

ret = 1;

err:
  return ret;
}

/* Computes a + b and stores the result in r. r could be a or b, a could be b. */
int ec_GF2m_nist_add(const EC_GROUP *group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b, BN_CTX *ctx)
{
  BN_CTX *new_ctx = NULL;
  BIGNUM *x0, *y0, *x1, *y1, *x2, *y2, *s, *t;
  int ret = 0;

  if (EC_POINT_is_at_infinity(group, a))
    {
      if (!EC_POINT_copy(r, b)) return 0;
      ret = 1;
    }

  if (EC_POINT_is_at_infinity(group, b))
    {
      if (!EC_POINT_copy(r, a)) return 0;
      ret = 1;
    }

  if (ctx == NULL)
    {
      ctx = new_ctx = BN_CTX_new();
      if (ctx == NULL) goto err;
    }

  BN_CTX_start(ctx);
  x0 = BN_CTX_get(ctx);
  y0 = BN_CTX_get(ctx);
  x1 = BN_CTX_get(ctx);
  y1 = BN_CTX_get(ctx);
  x2 = BN_CTX_get(ctx);
  y2 = BN_CTX_get(ctx);
  s = BN_CTX_get(ctx);
  t = BN_CTX_get(ctx);
  if (t == NULL) goto err;

  if (!BN_GF2m_const_init(x0, field_size)) goto err;
  if (!BN_GF2m_const_init(y0, field_size)) goto err;
  if (!BN_GF2m_const_init(x1, field_size)) goto err;
  if (!BN_GF2m_const_init(y1, field_size)) goto err;
  if (!BN_GF2m_const_init(x2, field_size)) goto err;
  if (!BN_GF2m_const_init(y2, field_size)) goto err;
+ if(!BN_GF2m_const_init(s, field_size)) goto err;
+ if(!BN_GF2m_const_init(t, field_size)) goto err;
+ if(a->Z_is_one)
+ {
+ if(!BN_GF2m_const_copy(x0, &a->X)) goto err;
+ if(!BN_GF2m_const_copy(y0, &a->Y)) goto err;
+ } else
+ {
+ if(!EC_POINT_get_affine_coordinates_GF2m(group, a, x0, y0, ctx)) goto err;
+ }
+ if(b->Z_is_one)
+ {
+ if(!BN_GF2m_const_copy(x1, &b->X)) goto err;
+ if(!BN_GF2m_const_copy(y1, &b->Y)) goto err;
+ } else
+ {
+ if(!EC_POINT_get_affine_coordinates_GF2m(group, b, x1, y1, ctx)) goto err;
+ }
+ if(!BN_GF2m_const_cmp_eq(x0, x1))
+ {
+ if(!BN_GF2m_const_add(t, x0, x1)) goto err;
+ if(!BN_GF2m_const_add(s, y0, y1)) goto err;
+ if(!BN_GF2m_const_add(x, x0, x1, t)) goto err;
+ if(!BN_GF2m_const_add(x2, x, s)) goto err;
+ if(!BN_GF2m_const_add(x2, x2, &group->a)) goto err;
+ } else
+ { /* Computes 2 * a and stores the result in r. r could be a */
+ if(!BN_GF2m_const_cmp_eq(y0, y1) || BN_GF2m_const_cmp_eq(x1))
+ {
+ if(!EC_POINT_set_to_infinity(group, r)) goto err;
+ ret = 1;
+ goto err;
+ }
+ if(!BN_GF2m_const_add(x2, x, s)) goto err;
+ if(!BN_GF2m_const_add(x2, x, &group->a)) goto err;
+ if(!BN_GF2m_const_add(x2, x2, s)) goto err;
+ if(!BN_GF2m_const_add(x2, x2, t)) goto err;
+ } if(!BN_GF2m_const_add(y2, x1, x2)) goto err;
+ if(!BN_GF2m_const_add(y2, x2, y2, s, ctx)) goto err;
+ } if(!BN_GF2m_const_add(y2, x2, y2, ctx)) goto err;
+ ret = 1;
+ +
+ err:
+ BN_CTX_end(ctx);
+ if(new_ctx != NULL)
+ BN_CTX_free(new_ctx);
+ return ret;
+ }
+
+ /* Computes Z * a and stores the result in r. r could be a */
+ /* Uses algorithm 4.10.2 of IEEE P1363 */
+ +int ec_GF2m_nist_add(const EC_GROUP *group, const EC_POINT *a, BN_CTX *ctx)
+ { /* Computes Z * a and stores the result in r. r could be a */
+ return ec_GF2m_nist_add(group, r, a, a, ctx);
+ }
+ +int ec_GF2m_nist_invert(const EC_GROUP *group, const EC_POINT *point, BN_CTX *ctx)
+ { /* point is its own inverse */
+ if(EC_POINT_is_at_infinity(group, point)) || BN_is_zero((point->Y))
+ /* point is its own inverse */
+ return 1;
if (! EC_POINT_make_affine (group, point, ctx)) return 0;
return BN_GF2m_const_add(&point->Y, &point->X, &point->Y);
}

/* Indicates whether the given point is the point at infinity. */
int ec_GF2m_nist_is_at_infinity(const EC_GROUP *group, const EC_POINT *point)
{
    return BN_is_zero(&point->Z);
}

/* Indicates whether two points are equal. */
int ec_GF2m_nist_cmp(const EC_GROUP *group, const EC_POINT *a, const EC_POINT *b, BN_CTX *ctx)
{
    int ret = -1;
    BN_CTX *new_ctx = NULL;
    int (* field_mul )( const EC_GROUP *, BIGNUM *, const BIGNUM *, const BIGNUM *, BN_CTX *);
    int (* field_sqr )( const EC_GROUP *, BIGNUM *, const BIGNUM *, BN_CTX *);
    int field_size;
    
    if ( EC_POINT_is_at_infinity ( group , point ))
        return 1;
    
    field_mul = group ->meth -> field_mul ;
    field_sqr = group ->meth -> field_sqr ;
    field_size = ( group -> poly[0] / BN_BITS2 ) + 1;
    
    if (ctx == NULL)
    {
        ctx = new_ctx = BN_CTX_new();
        if (ctx == NULL)
            return -1;
    }
    
    BN_CTX_start(ctx);
    y2 = BN_CTX_get(ctx);
    lh = BN_CTX_get(ctx);
    if (lh == NULL) goto err;
    
    if (! BN_GF2m_const_init(y2 , field_size )) goto err;
    if (! BN_GF2m_const_init(lh , field_size )) goto err;
    
    /* We have a curve defined by a Weierstrass equation
       y^2 = xy = z^3 + a*z^2 + b.
       <=> (a + a) * z + y ) * z + b + y^2 = 0
    */
    if (! BN_GF2m_const_add(lh , &point->Y, &group->a)) goto err;
    if (! field_mul ( group , lh , lh , &point->X, ctx)) goto err;
    if (! BN_GF2m_const_add(lh , lh , &group->b)) goto err;
    if (! field_sqr ( group , y2 , &point->Y, ctx)) goto err;
    if (! BN_GF2m_const_add(lh , lh , y2)) goto err;
    
    ret = BN_GF2m_const_cmp_zero(lh);
    err:
    if (ctx) BN_CTX_end(ctx);
    if (new_ctx) BN_CTX_free(new_ctx);
    return ret;
}

/* Indicates whether the given EC_POINT is an actual point on the curve defined
 * in the EC_GROUP. A point is valid if it satisfies the Weierstrass equation:
 * y^2 + x*y = z^3 + a*z^2 + b.
 */
int ec_GF2m_nist_is_on_curve(const EC_GROUP *group, const EC_POINT *point, BN_CTX *ctx)
{
    int ret = -1;
    BN_CTX *new_ctx = NULL;
    BIGNUM *lh , *y2;
    int (* field_mul )( const EC_GROUP *, BIGNUM *, const BIGNUM *, const BIGNUM *, BN_CTX *);
    int (* field_sqr )( const EC_GROUP *, BIGNUM *, const BIGNUM *, BN_CTX *);
    int field_size;
    
    if ( EC_POINT_is_at_infinity ( group , point ))
        return 1;
    
    field_mul = group ->meth -> field_mul ;
    field_sqr = group ->meth -> field_sqr ;
    field_size = ( group -> poly[0] / BN_BITS2 ) + 1;
    
    if (ctx == NULL)
    {
        ctx = new_ctx = BN_CTX_new();
        if (ctx == NULL)
            return -1;
    }
    
    BN_CTX_start(ctx);
    y2 = BN_CTX_get(ctx);
    lh = BN_CTX_get(ctx);
    if (lh == NULL) goto err;
    
    if (! BN_GF2m_const_init(y2 , field_size )) goto err;
    if (! BN_GF2m_const_init(lh , field_size )) goto err;
    
    /* only support affine coordinates */
    if (!point->Z_is_one) return -1;
    
    if (ctx == NULL)
    {
        ctx = new_ctx = BN_CTX_new();
        if (ctx == NULL)
            return NULL;
    }
    
    BN_CTX_start(ctx);
    y2 = BN_CTX_get(ctx);
    lh = BN_CTX_get(ctx);
    if (lh == NULL) goto err;
    
    if (! BN_GF2m_const_init(y2 , field_size )) goto err;
    if (! BN_GF2m_const_init(lh , field_size )) goto err;
    
    /* We have a curve defined by a Weierstrass equation
       y^2 + x*y = z^3 + a*z^2 + b.
       <=> (a + a) * z + y ) * z + b + y^2 = 0
    */
    if (! BN_GF2m_const_add(lh , &point->Y, &group->a)) goto err;
    if (! field_mul ( group , lh , lh , &point->X, ctx)) goto err;
    if (! BN_GF2m_const_add(lh , lh , &group->b)) goto err;
    if (! field_sqr ( group , y2 , &point->Y, ctx)) goto err;
    if (! BN_GF2m_const_add(lh , lh , y2)) goto err;
    
    ret = BN_GF2m_const_cmp_zero(lh);
    err:
    if (ctx) BN_CTX_end(ctx);
    if (new_ctx) BN_CTX_free(new_ctx);
    return ret;
+ "ctx"
+ {  
+  BIGNUM *aX, *aY, *bX, *bY;
+  BN_CTX *new_ctx = NULL;
+  int field_size, ret = -1;
+  field_size = (group->poly[0] / BN_BITS2) + 1;
+  if (EC_POINT_is_at_infinity(group, a))  
+  {  
+    return EC_POINT_is_at_infinity(group, b) ? 0 : 1;
+  }
+  if (EC_POINT_is_at_infinity(group, b))  
+  {  
+    ret urn 1;
+  }
+  if (a->Z_is_one && b->Z_is_one)  
+  {  
+    ret urn 0;
+  }
+  if (ctx == NULL)  
+  {  
+    ctx = new_ctx = BN_CTX_new();
+    if (ctx == NULL)  
+      ret urn -1;
+  }
+  BN_CTX_start(ctx);
+  aX = BN_CTX_get(ctx);
+  aY = BN_CTX_get(ctx);
+  bX = BN_CTX_get(ctx);
+  bY = BN_CTX_get(ctx);
+  if (bY == NULL) goto err;
+  if (! BN_GF2m_const_init(aX, field_size)) goto err;
+  if (! BN_GF2m_const_init(aY, field_size)) goto err;
+  if (! BN_GF2m_const_init(bX, field_size)) goto err;
+  if (! BN_GF2m_const_init(bY, field_size)) goto err;
+  if (! EC_POINT_get_affine_coordinates_GF2m(group, a, aX, aY, ctx)) goto err;
+  if (! EC_POINT_get_affine_coordinates_GF2m(group, b, bX, bY, ctx)) goto err;
+  ret = ((BN_GF2m_const_cmp_eq(aX, bX) == 0) && BN_GF2m_const_cmp_eq(aY, bY) == 0) ? 0 : 1;
+  }
+  if (ctx == NULL)  
+  {  
+    BN_CTX_end(ctx);
+    if (new_ctx) BN_CTX_free(new_ctx);
+    ret urn ret;
+    }  
+  }
+  BN_CTX_nist_make_affine(const EC_GROUP *group, const EC_POINT *point, BN_CTX *ctx)  
+  {  
+    BN_CTX *new_ctx = NULL;
+    BIGNUM *x, *y;
+    int ret = 0;
+    int field_size = (group->poly[0] / BN_BITS2) + 1;
+    BN_CTX_start(ctx);
+    x = BN_CTX_get(ctx);
+    y = BN_CTX_get(ctx);
+    if (y == NULL) goto err;
+    if (! BN_GF2m_const_init(x, field_size)) goto err;
+    if (! BN_GF2m_const_init(y, field_size)) goto err;
+    if (! EC_POINT_get_affine_coordinates_GF2m(group, point, x, y, ctx)) goto err;
+ if (!BN_GF2m_const_copy(&point->X, x)) goto err;
+ if (!BN_GF2m_const_copy(&point->Y, y)) goto err;
+ if (!BN_one(&point->Z)) goto err;
+
+ ret = 1;
+
+ err:
+ if (ctx) BN_CTX_end(ctx);
+ if (new_ctx) BN_CTX_free(new_ctx);
+ return ret;
+
+} /* Forces each of the EC_POINTs in the given array to use affine coordinates. */
+int ec_GF2m_nist_points_make_affine(const EC_GROUP *group, size_t num, EC_POINT *points[],
+ BN_CTX *ctx) {
+ size_t i;
+ for (i = 0; i < num; i++) {
+ if (!group->meth->make_affine(group, points[i], ctx)) return 0;
+ }
+ return 1;
+}
+
+/* Wrapper to binary polynomial field multiplication implementation. */
+int ec_GF2m_nist163_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx) {
+ return BN_GF2m_mul_xmm_nist163(r, a, b);
+}
+
+/* Wrapper to binary polynomial field squaring implementation. */
+int ec_GF2m_nist163_field_sqr(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, BN_CTX *ctx)
+ {
+ return BN_GF2m_sqr_xmm_nist163(r, a);
+}
+
+/* Wrapper to binary polynomial field division implementation. */
+int ec_GF2m_nist163_field_div(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx) {
+ return BN_GF2m_div_xmm_nist163(r, a, b);
+}
+
+/* Wrapper to binary polynomial field multiplication implementation. */
+int ec_GF2m_sect193_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx) {
+ return BN_GF2m_mul_xmm_sect193(r, a, b);
+}
+
+/* Wrapper to binary polynomial field squaring implementation. */
+int ec_GF2m_sect193_field_sqr(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, BN_CTX *ctx)
+ {
+ return BN_GF2m_sqr_xmm_sect193(r, a);
+}
+
+/* Wrapper to binary polynomial field division implementation. */
+int ec_GF2m_sect193_field_div(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx) {
+ return BN_GF2m_div_xmm_sect193(r, a, b);
+}
+
+/* Wrapper to binary polynomial field multiplication implementation. */
+int ec_GF2m_nist233_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx) {
+ return BN_GF2m_mul_xmm_nist233(r, a, b);
+}
/* Wrapper to binary polynomial field squaring implementation. */
int ec_GF2m_nist233_field_sqr (const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, BN_CTX *ctx)
{
    return BN_GF2m_sqr_xmm_nist233(r, a);
}

/* Wrapper to binary polynomial field division implementation. */
int ec_GF2m_nist233_field_div(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
BIGNUM *b, BN_CTX *ctx)
{
    return BN_GF2m_div_xmm_nist233(r, a, b);
}

/* Wrapper to binary polynomial field multiplication implementation. */
int ec_GF2m_sect239_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
BIGNUM *b, BN_CTX *ctx)
{
    return BN_GF2m_mul_xmm_sect239(r, a, b);
}

/* Wrapper to binary polynomial field squaring implementation. */
int ec_GF2m_sect239_field_sqr(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, BN_CTX *ctx)
{
    return BN_GF2m_sqr_xmm_sect239(r, a);
}

/* Wrapper to binary polynomial field division implementation. */
int ec_GF2m_sect239_field_div(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
BIGNUM *b, BN_CTX *ctx)
{
    return BN_GF2m_div_xmm_sect239(r, a, b);
}

/* Wrapper to binary polynomial field multiplication implementation. */
int ec_GF2m_nist283_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
BIGNUM *b, BN_CTX *ctx)
{
    return BN_GF2m_mul_xmm_nist283(r, a, b);
}

/* Wrapper to binary polynomial field squaring implementation. */
int ec_GF2m_nist283_field_sqr(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, BN_CTX *ctx)
{
    return BN_GF2m_sqr_xmm_nist283(r, a);
}

/* Wrapper to binary polynomial field division implementation. */
int ec_GF2m_nist283_field_div(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
BIGNUM *b, BN_CTX *ctx)
{
    return BN_GF2m_div_xmm_nist283(r, a, b);
}

/* Wrapper to binary polynomial field multiplication implementation. */
int ec_GF2m_nist409_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
BIGNUM *b, BN_CTX *ctx)
{
    return BN_GF2m_mul_xmm_nist409(r, a, b);
}

/* Wrapper to binary polynomial field squaring implementation. */
int ec_GF2m_nist409_field_sqr(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, BN_CTX *ctx)
{
    return BN_GF2m_sqr_xmm_nist409(r, a);
}
+ return BN_GF2m_sqr_xmm_nist409(r, a);
+ }
+ /* Wrapper to binary polynomial field division implementation. */
+ int ec_GF2m_nist409_field_div(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx)
+ { + return BN_GF2m_div_xmm_nist409(r, a, b);
+ }
+ /* Wrapper to binary polynomial field multiplication implementation. */
+ int ec_GF2m_nist571_field_mul(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx)
+ { + return BN_GF2m_mul_xmm_nist571(r, a, b);
+ }
+ /* Wrapper to binary polynomial field squaring implementation. */
+ int ec_GF2m_nist571_field_sqr(const EC_GROUP *group, BIGNUM *r, const BIGNUM *a, const
+ BIGNUM *b, BN_CTX *ctx)
+ { + return BN_GF2m_sqr_xmm_nist571(r, a);
+ }
+ /* Fallback to simple method */
+ const EC_METHOD *EC_GF2m_nist163_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ const EC_METHOD *EC_GF2m_sect193_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ const EC_METHOD *EC_GF2m_nist233_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ const EC_METHOD *EC_GF2m_sect239_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ const EC_METHOD *EC_GF2m_nist283_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ const EC_METHOD *EC_GF2m_nist409_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ const EC_METHOD *EC_GF2m_nist571_method(void)
+ { + return EC_GF2m_simple_method();
+ }
+ #endif
#endif
+ if (!BN_GF2m_Maddle_xmm_nist163r(x, x1, x2, z2, k, c)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_sect193r(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle_xmm_sect193r(x, x1, x2, z1, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist233k(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle_xmm_nist233k(x, x1, x2, z1, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist233r(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle_xmm_nist233r(x, x1, x2, z1, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_sect239k(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle_xmm_sect239k(x, x1, x2, z1, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist283k(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle_xmm_nist283k(x, x1, x2, z1, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist283r(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
z1,
+ const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle xmm_nist283r(x, x1, z1, x2, z2, k, c)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist409k(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *
+ z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle xmm_nist409k(x, x1, z1, x2, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist409r(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *
+ z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle xmm_nist409r(x, x1, z1, x2, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist571k(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *
+ z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle xmm_nist571k(x, x1, z1, x2, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+static int gf2m_Maddle_nist571r(const EC_GROUP *group, const BIGNUM *x, BIGNUM *x1, BIGNUM *
+ z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c, BN_CTX *ctx)
+ {
+ int ret = 0;
+ if (!BN_GF2m_Maddle xmm_nist571r(x, x1, z1, x2, z2, k)) goto err;
+ ret = 1;
+ err:
+ return ret;
+ }

+/* Compute the x, y affine coordinates from the point (x1, z1) (x2, z2)
+ * using Montgomery point multiplication algorithm Mxy() in appendix of
+ * Lopez, J. and Dahab, R. "Fast multiplication on elliptic curves over
+ * GF(2^m) without precomputation" (CNES '99, LNCS 1717).
+ * Returns:
+ * 0 on error
+ * 1 if return value should be the point at infinity
+ * 2 otherwise
+ */
+static int gf2m_Mxy(const EC_GROUP *group, const BIGNUM *x, const BIGNUM *y, BIGNUM *x1,
+ BIGNUM *z1, BIGNUM *x2, BIGNUM *z2, BN_ULONG field_size, BN_CTX *ctx)
BIGNUM *t3, *t4, *t5;
int ret = 0;

if (BN_GF2m_const_cmp_zero(z1)) {
    BN_GF2m_const_setword(x2, 0);
    BN_GF2m_const_setword(z2, 0);
    return 1;
}

if (BN_GF2m_const_cmp_zero(z2)) {
    if (!BN_GF2m_const_copy(x2, x)) return 0;
    if (!BN_GF2m_const_add(x2, x, y)) return 0;
    return 2;
}

/* Since Mxy is static we can guarantee that ctx != NULL. */
BN_CTX_start(ctx);
t3 = BN_CTX_get(ctx);
t4 = BN_CTX_get(ctx);
t5 = BN_CTX_get(ctx);
if (t5 == NULL) goto err;

if (!BN_GF2m_const_init(t3, field_size)) goto err;
if (!BN_GF2m_const_init(t4, field_size)) goto err;
if (!BN_GF2m_const_init(t5, field_size)) goto err;

if (!BN_GF2m_const_setone(t5)) goto err;

if (!group->meth->field_mul(group, t3, z1, z2, ctx)) goto err;
if (!group->meth->field_mul(group, z1, z1, x, ctx)) goto err;
if (!BN_GF2m_const_add(z1, z1, x1)) goto err;
if (!group->meth->field_mul(group, z2, z2, x, ctx)) goto err;
if (!group->meth->field_mul(group, x1, z2, x1, ctx)) goto err;
if (!BN_GF2m_const_add(z2, z2, x2)) goto err;

if (!group->meth->field_mul(group, z2, z2, z1, ctx)) goto err;
if (!group->meth->field_sqr(group, t4, x, ctx)) goto err;
if (!BN_GF2m_const_add(t4, t4, y)) goto err;
if (!group->meth->field_mul(group, t4, t4, t3, ctx)) goto err;
if (!BN_GF2m_const_add(t4, t4, z2)) goto err;

if (!group->meth->field_mul(group, t3, t3, x, ctx)) goto err;
if (!group->meth->field_div(group, t3, t5, t3, ctx)) goto err;
if (!group->meth->field_mul(group, t4, t4, t3, ctx)) goto err;
if (!BN_GF2m_const_add(z2, x2, x)) goto err;

if (!group->meth->field_mul(group, z2, z2, t4, ctx)) goto err;
if (!BN_GF2m_const_add(z2, z2, y)) goto err;

ret = 2;
+
err:
BN_CTX_end(ctx);
return ret;
+
}

/* Computes scalar*point and stores the result in r.
 * point can not equal r.
 * Uses algorithm 2P of
 * Lopez, J. and Dahab, R. "Fast multiplication on elliptic curves over
 * GF(2^m) without precomputation" (CHES '99, LNCS 1717).
 * +*/
int ec_GF2m_montgomery_point_multiply(const EC_GROUP *group, EC_POINT *r, const BIGNUM *
scalar,
const EC_POINT *point, BN_CTX *ctx) {
+
/* Init */
int ret = 0, i;
BIGNUM *x1, *x2, *z1, *z2, *fscalar, *forder;
const BIGNUM *c = NULL;
BN_ULONG mask, word, keybit, field_size;
static int (*gf2m_Maddle)(const EC_GROUP *, const BIGNUM *, BIGNUM *, BIGNUM *,
const BIGNUM *, const BIGNUM *, BN_ULONG, const BIGNUM *, BN_CTX *);
if (r == point) {
    ECerr(EC_F_EC_GF2M_MONTGOMERY_POINT_MULTIPLY, EC_R_INVALID_ARGUMENT);
    return 0;
}
/* if result should be point at infinity */
if (scalar == NULL) || BN_is_zero(scalar) || (point == NULL) ||
EC_POINT_is_at_infinity(group, point)) {
    return EC_POINT_set_to_infinity(group, r);
}
/* only support affine coordinates */
if (!point->Z_is_one) return 0;
/* Since point_multiply is static we can guarantee that ctx != NULL. */
BN_CTX_start(ctx);
order = BN_CTX_get(ctx);
fscalar = BN_CTX_get(ctx);
z1 = BN_CTX_get(ctx);
if (z1 == NULL) goto err;
/* Set constant field size of BN_BITS2-words */
field_size = (group->poly[0] / BN_BITS2) + 1;
/* Initialize constant size elements */
if (!BN_GF2m_const_init(fscalar, (&group->order)->top)) goto err;
if (!BN_GF2m_const_init(order, (&group->order)->top)) goto err;
if (!BN_GF2m_const_init(z1, field_size)) goto err;
if (!BN_GF2m_const_init(x1, field_size)) goto err;
if (!BN_GF2m_const_init(x2, field_size)) goto err;
if (!BN_GF2m_const_init(y1, field_size)) goto err;
if (!BN_GF2m_const_init(y2, field_size)) goto err;
x1 = &r->X;
z1 = &r->Y;
/* Set Double & Add method for improved 2P algorithm */
switch (group->curve_name) {
case NID_sect163k1: gf2m_Maddle = gf2m_Maddle_nist163k; break;
case NID_sect163r1: gf2m_Maddle = gf2m_Maddle_nist163r; break;
case NID_sect193r1: gf2m_Maddle = gf2m_Maddle_sect193r; break;
case NID_sect233k1: gf2m_Maddle = gf2m_Maddle_nist233k; break;
case NID_sect233r1: gf2m_Maddle = gf2m_Maddle_nist233r; break;
case NID_sect283k1: gf2m_Maddle = gf2m_Maddle_nist283k; break;
case NID_sect283r1: gf2m_Maddle = gf2m_Maddle_nist283r; break;
case NID_sect409k1: gf2m_Maddle = gf2m_Maddle_nist409k; break;
case NID_sect409r1: gf2m_Maddle = gf2m_Maddle_nist409r; break;
case NID_sect571k1: gf2m_Maddle = gf2m_Maddle_nist571k; break;
case NID_sect571r1: gf2m_Maddle = gf2m_Maddle_nist571r; break;
default: goto err;
}
/* Load precomputed value c = sqrt(b) = b^((2^m-1)) only if group parameter b != 1 */
if (BN_GF2m(const_cmp_one(&group->b))) {
    if ((c = ec_GF2m_get_sqrt_b(group)) == NULL) goto err;
}
/* Precompute coordinates */
if (BN_GF2m(const_copy(x1, &point->X)) goto err; // x1 = x
if (BN_GF2m(const_setone(z1)) goto err; // z1 = 1
if (!NULL->meth->field_sqr(group, x2, x1, ctx)) goto err; // x2 = x1^2 = x^2
if (!NULL->meth->field_sqr(group, x2, x2, ctx)) goto err;
if (!BN_GF2m(const_add(x2, x2, &group->b)) goto err; // x2 = x^4 + b
/* Add group order to scalar to assert certain bit length */
if (BN_GF2m(const_int_add(fscalar, fscalar, &group->order)) goto err;
i = (group->order)->top - 1;
mask = BN_TBIT;
word = (group->order)->d[1];
while (((word & mask) mask >>= 1;
+ mask <<= 1;
+ mask = (0 - ((mask & (fscalar->d[fscalar->top-1]))==0));
+ for (i=0; i < forder->top; i++)
+ {
+ forder->d[i] = mask & (&group->order)->d[i];
+ }
+ if (!BN_GF2m_const_int_add(fscalar, fscalar, forder)) goto err;
+ /* find top most bit and go one past it */
+ i = fscalar->top - 1;
+ mask = BN_TBIT;
+ word = fscalar->d[i];
+ while (!(word & mask)) mask >>= 1;
+ mask >>= 1;
+ /* if top most bit was at word break , go to next word */
+ if (mask)
+ {
+ i--;
+ mask = BN_TBIT;
+ }
+ for (; i >= 0; i--)
+ {
+ word = fscalar->d[i];
+ while (mask)
+ {
+ /* Set keybit */
+ keybit = ((word & mask) != 0);
+ /* Execute MAdd & Mdouble */
+ if (!(*gf2m_Maddle)(group, &point->X, x1, z1, x2, z2, keybit, c, ctx)) goto err;
+ mask >>= 1;
+ }
+ mask = BN_TBIT;
+ }
+ /* convert out of "projective" coordinates */
+ i = gf2m_Mxy(group, &point->X, &point->Y, x1, z1, x2, z2, field_size, ctx);
+ if (i == 0) goto err;
+ else if (i == 1)
+ {
+ if (!EC_POINT_set_to_infinity(group, r)) goto err;
+ }
+ else
+ {
+ if (!BN_one(&r->Z)) goto err;
+ r->Z_is_one = 1;
+ }
+ /* GF(2^m) field elements should always have BIGNUM::neg = 0 */
+ (&r->X)->neg = 0;
+ (&r->Y)->neg = 0;
+ ret = 1;
+ err:
+ BN_CTX_end(ctx);
+ return ret;
+ }
+
+ /* Computes the sum */
+ scalar*generator = scalars[0]*points[0] + ... + scalars[num-1]*points[num-1]
+ * gracefully ignoring NULL scalar values.
+ */
+ int ec_GF2m_nist_mul(const EC_GROUP *group, EC_POINT *r, const BIGNUM *scalar,
+ size_t num, const EC_POINT *points[], const BIGNUM *scalars[], BN_CTX *ctx)
+ {
+ BN_CTX *new_ctx = NULL;
+ int ret = 0;
+ size_t i;
+ EC_POINT *p=NULL;
+ EC_POINT *acc = NULL;
+ if (ctx == NULL)
+ {
ctx = new_ctx = BN_CTX_new();
if (ctx == NULL)
    return 0;
}

/* This implementation never uses ec_wNAF_mult. */

if ((p = EC_POINT_new(group)) == NULL) goto err;
if ((acc = EC_POINT_new(group)) == NULL) goto err;
if (!EC_POINT_set_to_infinity(group, acc)) goto err;

if (scalar)
    {
        if (!ec_GF2m_montgomery_point_multiply(group, p, scalar, group->generator, ctx)) goto err;
        if (BN_is_negative(scalar))
            if (!group->meth->invert(group, p, ctx)) goto err;
        if (!group->meth->add(group, acc, acc, p, ctx)) goto err;
    }

for (i = 0; i < num; i++)
    {
        if (!ec_GF2m_montgomery_point_multiply(group, p, scalars[i], points[i], ctx)) goto err;
        if (BN_is_negative(scalars[i]))
            if (!group->meth->invert(group, p, ctx)) goto err;
        if (!group->meth->add(group, acc, acc, p, ctx)) goto err;
    }

if (!EC_POINT_copy(r, acc)) goto err;
ret = 1;

err:
if (p) EC_POINT_free(p);
if (acc) EC_POINT_free(acc);
if (new_ctx != NULL)
    BN_CTX_free(new_ctx);
return ret;
}

/* crypto/ec/ec2_nist_prec.c */

/* Written by Manuel Bluhm for the OpenSSL project. */

/* ====================================================================
 * Copyright (c) 1998-2011 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *       "This product includes software developed by the OpenSSL Project
 *       for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without prior
 *    written permission. For written permission, please contact openssl-core@openssl.org.
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 */

133
This product includes cryptographic software written by Eric Young
(eay@cryptsoft.com). This product includes software written by Tim
Hudson (tjh@cryptsoft.com).

*"This product is provided 'as is' and any
expressed or implied warranties, including, but not limited to, the
implied warranties of merchantability and fitness for a particular
purpose are disclaimed. In no event shall the OpenSSL project or
its contributors be liable for any direct, indirect, incidental,
special, exemplary, or consequential damages (including, but
not limited to, procurement of substitute goods or services;
loss of use, data, or profits; or business interruption) in
however caused and on any theory of liability, whether in contract,
strict liability, or tort (including negligence or otherwise).
Arising in any way out of the use of this software, even if advised
of the possibility of such damage.

This product includes software developed by the OpenSSL Project
(http://www.openssl.org/)

Redistributions of any form whatsoever must retain the following
acknowledgment:

"This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)

This product includes cryptographic software written by Eric Young
(eay@cryptsoft.com). This product includes software written by Tim
Hudson (tjh@cryptsoft.com).

#ifndef OPENSSL_FAST_EC2M

#include "openssl/bn.h"
#include "openssl/obj_mac.h"
#include "ec_lcl.h"

#define BN_NIST163_TOP (163 / BN_BITS2) + 1
#define BN_SECT193_TOP (193 / BN_BITS2) + 1
#define BN_NIST233_TOP (233 / BN_BITS2) + 1
#define BN_SECT239_TOP (239 / BN_BITS2) + 1
#define BN_NIST283_TOP (283 / BN_BITS2) + 1
#define BN_NIST409_TOP (409 / BN_BITS2) + 1
#define BN_SECT571_TOP (571 / BN_BITS2) + 1

/* Precomputation values of \(c = \sqrt{b} = b^{2^{(m-1)}}\) for SECT/NIST CURVES */

static const BN_ULONG _gf2m_nist163r1[] = {
  0xCD01BF889B68360ULL, 0x6E1856BC74A472ULL, 0x000000009917A25ULL
};
static const BN_ULONG _gf2m_nist163r2[] = {
  0xDA89C03969F34D5ULL, 0xDF8927593D21C6ULL, 0x00000002C25B85ULL
};
static const BN_ULONG _gf2m_sect193r1[] = {
  0xD43F8BE752FDFB06ULL, 0x139483AFD24E42EULL, 0xDE5FB3DDE67CDULL,
  0x0000000000010ULL
};
static const BN_ULONG _gf2m_sect193r2[] = {
  0x03830909465F6662ULL, 0xBFBA42912F39ACBDULL, 0x5F74B142AEBF00E3ULL,
  0x0000000000001ULL
};
static const BN_ULONG _gf2m_nist233r1[] = {
  0x17442AEDE9B93F6ULL, 0x304224CA17CD082EULL, 0x9FB6F8352F2D20AULL,
  0x5792B188893C00ULL, 0x000000000000001ULL
};
static const BN_ULONG _gf2m_nist283r1[] = {
  0x17442AEDE9B93F6ULL, 0x304224CA17CD082EULL, 0x9FB6F8352F2D20AULL,
  0x5792B188893C00ULL, 0x000000000000001ULL
};
static const BN_ULONG _gf2m_nist409r1[] = {
  0x872ACBF0C265E5FULL, 0x73326C528A48E27ULL, 0xFDE89590CF6576ULL,
  0xDD739A05BFFD582ULL, 0x0732D56640C205ULL
};
/* BIGNUM declarations */
static const BIGNUM _bignum_gf2m_nist163r1 =
{
  (BN_ULONG *) _gf2m_nist163r1,
  BN_NIST163_TOP,
  BN_NIST163_TOP,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_nist163r2 =
{
  (BN_ULONG *) _gf2m_nist163r2,
  BN_NIST163_TOP,
  BN_NIST163_TOP,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_sect193r1 =
{
  (BN_ULONG *) _gf2m_sect193r1,
  BN_SECT193_TOP,
  BN_SECT193_TOP,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_sect193r2 =
{
  (BN_ULONG *) _gf2m_sect193r2,
  BN_SECT193_TOP,
  BN_SECT193_TOP,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_nist233r1 =
{
  (BN_ULONG *) _gf2m_nist233r1,
  BN_NIST233_TOP,
  BN_NIST233_TOP,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_nist283r1 =
{
  (BN_ULONG *) _gf2m_nist283r1,
  BN_NIST283_Top,
  BN_NIST283_Top,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_nist409r1 =
{
  (BN_ULONG *) _gf2m_nist409r1,
  BN_NIST409_Top,
  BN_NIST409_Top,
  0,
  BN_FLG_STATIC_DATA
};

static const BIGNUM _bignum_gf2m_nist571r1 =
{
  (BN_ULONG *) _gf2m_nist571r1,
  BN_SECT571_Top,
  BN_SECT571_Top,
  0,
  BN_FLG_STATIC_DATA
};

/* Returns the precomputation value for a specific curve or NULL. */
const BIGNUM *ec_GF2m_get_sqrt_b(const EC_GROUP *group)
{
  const BIGNUM *ret;
}
```c
+ switch (group->curve_name)
+ {
+   case NID_sect163r1: ret = &_bignum_gf2m_nist163r1; break;
+   case NID_sect163r2: ret = &_bignum_gf2m_nist163r2; break;
+   case NID_sect193r1: ret = &_bignum_gf2m_nist193r1; break;
+   case NID_sect193r2: ret = &_bignum_gf2m_nist193r2; break;
+   case NID_sect233r1: ret = &_bignum_gf2m_nist233r1; break;
+   case NID_sect283r1: ret = &_bignum_gf2m_nist283r1; break;
+   case NID_sect409r1: ret = &_bignum_gf2m_nist409r1; break;
+   case NID_sect571r1: ret = &_bignum_gf2m_nist571r1; break;
+   default: ret = NULL; break;
+ }
+ return ret;
+
+}  // switch
++endif
```

```diff
diff -urN openssl-orig/crypto/ec/ec_curve.c openssl-work/crypto/ec/ec_curve.c
--- openssl-orig/crypto/ec/ec_curve.c 2013-02-11 16:26:04.000000000 +0100
+++ openssl-work/crypto/ec/ec_curve.c 2013-05-12 05:37:39.769151907 +0200
@@ -1874,20 +1874,20 @@
   - { NID_sect163r1, &EC_SECG_CHAR2_163R1.h, 0, "SECG curve over a 163 bit binary field" },
   - { NID_sect163r2, &EC_SECG_CHAR2_163R2.h, 0, "SECG curve over a 163 bit binary field" },
   - { NID_sect193r1, &EC_SECG_CHAR2_193R1.h, 0, "SECG curve over a 193 bit binary field" },
   - { NID_sect193r2, &EC_SECG_CHAR2_193R2.h, 0, "SECG curve over a 193 bit binary field" },
   - { NID_sect233r1, &EC_SECG_CHAR2_233R1.h, 0, "SECG curve over a 233 bit binary field" },
   - { NID_sect233r2, &EC_SECG_CHAR2_233R2.h, 0, "SECG curve over a 233 bit binary field" },
   - { NID_sect283r1, &EC_SECG_CHAR2_283R1.h, 0, "SECG curve over a 283 bit binary field" },
   - { NID_sect283r2, &EC_SECG_CHAR2_283R2.h, 0, "SECG curve over a 283 bit binary field" },
   - { NID_sect409r1, &EC_SECG_CHAR2_409R1.h, 0, "SECG curve over a 409 bit binary field" },
   - { NID_sect409r2, &EC_SECG_CHAR2_409R2.h, 0, "SECG curve over a 409 bit binary field" },
   - { NID_sect571r1, &EC_SECG_CHAR2_571R1.h, 0, "SECG curve over a 571 bit binary field" },
   - { NID_sect571r2, &EC_SECG_CHAR2_571R2.h, 0, "SECG curve over a 571 bit binary field" },
   - { NID_sect113r2, &EC_SECG_CHAR2_113R2.h, 0, "SECG curve over a 113 bit binary field" },
   - { NID_sect131r1, &EC_SECG_CHAR2_131R1.h, 0, "SECG/WTLS curve over a 131 bit binary field" },
   - { NID_sect131r2, &EC_SECG_CHAR2_131R2.h, 0, "SECG curve over a 131 bit binary field" },
   - { NID_sect163k1, &EC_NIST_CHAR2_163K1.h, 0, "SECG curve over a 163 bit binary field" },
   + { NID_sect113r2, &EC_SECG_CHAR2_113R2.h, 0, "SECG curve over a 113 bit binary field" },
   + { NID_sect131r1, &EC_SECG_CHAR2_131R1.h, 0, "SECG/WTLS curve over a 131 bit binary field" },
   + { NID_sect131r2, &EC_SECG_CHAR2_131R2.h, 0, "SECG curve over a 131 bit binary field" },
   + { NID_sect163k1, &EC_NIST_CHAR2_163K1.h, 0, "NIST/SECG/WTLS curve over a 163 bit binary field" },
   + { NID_sect163r1, &EC_SECG_CHAR2_163R1.h, 0, "SECG/WTLS curve over a 163 bit binary field" },
   + { NID_sect163r2, &EC_SECG_CHAR2_163R2.h, 0, "SECG curve over a 163 bit binary field" },
   + { NID_sect193r1, &EC_SECG_CHAR2_193R1.h, 0, "SECG curve over a 193 bit binary field" },
   + { NID_sect193r2, &EC_SECG_CHAR2_193R2.h, 0, "SECG curve over a 193 bit binary field" },
   + { NID_sect233k1, &EC_NIST_CHAR2_233K1.h, 0, "NIST/SECG/WTLS curve over a 233 bit binary field" },
   + { NID_sect233r1, &EC_NIST_CHAR2_233R1.h, 0, "NIST/SECG curve over a 233 bit binary field" },
   + { NID_sect239k1, &EC_NIST_CHAR2_239K1.h, 0, "NIST/SECG/WTLS curve over a 239 bit binary field" },
   + { NID_sect239r1, &EC_NIST_CHAR2_239R1.h, 0, "NIST/SECG curve over a 239 bit binary field" },
   + { NID_sect239r2, &EC_NIST_CHAR2_239R2.h, 0, "NIST/SECG curve over a 239 bit binary field" },
   + { NID_sect283k1, &EC_NIST_CHAR2_283K1.h, 0, "NIST/SECG/WTLS curve over a 283 bit binary field" },
   + { NID_sect283k2, &EC_NIST_CHAR2_283K2.h, 0, "NIST/SECG/WTLS curve over a 283 bit binary field" },
   + { NID_sect283r1, &EC_NIST_CHAR2_283R1.h, 0, "NIST/SECG curve over a 283 bit binary field" },
   + { NID_sect283r2, &EC_NIST_CHAR2_283R2.h, 0, "NIST/SECG curve over a 283 bit binary field" },
   + { NID_sect409k1, &EC_NIST_CHAR2_409K1.h, 0, "NIST/SECG/WTLS curve over a 409 bit binary field" },
   + { NID_sect409k2, &EC_NIST_CHAR2_409K2.h, 0, "NIST/SECG/WTLS curve over a 409 bit binary field" },
   + { NID_sect409r1, &EC_NIST_CHAR2_409R1.h, 0, "NIST/SECG curve over a 409 bit binary field" },
   + { NID_sect409r2, &EC_NIST_CHAR2_409R2.h, 0, "NIST/SECG curve over a 409 bit binary field" },
   + { NID_sect571k1, &EC_NIST_CHAR2_571K1.h, 0, "NIST/SECG/WTLS curve over a 571 bit binary field" },
   + { NID_sect571k2, &EC_NIST_CHAR2_571K2.h, 0, "NIST/SECG/WTLS curve over a 571 bit binary field" },
   + { NID_sect571r1, &EC_NIST_CHAR2_571R1.h, 0, "NIST/SECG curve over a 571 bit binary field" },
   + { NID_sect571r2, &EC_NIST_CHAR2_571R2.h, 0, "NIST/SECG curve over a 571 bit binary field" },
```

136
const EC_METHOD * EC_GF2m_simple_method(void);

/* Returns improved GF2m EC methods. Falls back to simple method if no improved method is available. */
const EC_METHOD * EC_GF2m_nist163_method(void);
const EC_METHOD * EC_GF2m_sect193_method(void);
const EC_METHOD * EC_GF2m_nist233_method(void);
const EC_METHOD * EC_GF2m_sect233_method(void);
const EC_METHOD * EC_GF2m_nist283_method(void);
const EC_METHOD * EC_GF2m_nist409_method(void);
const EC_METHOD * EC_GF2m_nist571_method(void);

#ifdef OPENSSL_FAST_EC2M

/* method functions in ec2_mult_nist.c */
int ec_GF2m_nist_mul(const EC_GROUP * group, EC_POINT *r, const BIGNUM * scalar, size_t num,
 const EC_POINT *points[], const BIGNUM * scalars[], BN_CTX * ctx);

/* method functions in ec2_nist.c */
int ec_GF2m_nist_group_copy(EC_GROUP *dest, const EC_GROUP * src);
int ec_GF2m_nist_group_set_curve(const EC_GROUP * group, const BIGNUM * p, const BIGNUM *a, +
const BIGNUM *b, BN_CTX *ctx);
int ec_GF2m_nist_point_copy(EC_POINT *dest, const EC_POINT *src);
int ec_GF2m_nist_point_set_to_infinity(const EC_GROUP * group, EC_POINT * point);
int ec_GF2m_nist_point_set_affine_coordinates(const EC_GROUP * group, EC_POINT * point,
const BIGNUM *x, const BIGNUM *y, BN_CTX *ctx);
int ec_GF2m_nist_point_get_affine_coordinates(const EC_GROUP * group, const EC_POINT *point,
+ BIGNUM *x, BIGNUM *y, BN_CTX *ctx);
int ec_GF2m_nist_add(const EC_GROUP * group, EC_POINT *r, const EC_POINT *a, const EC_POINT *b,
BN_CTX *ctx);
int ec_GF2m_nist_dbl(const EC_GROUP * group, EC_POINT *r, const EC_POINT *a, BN_CTX *ctx);
int ec_GF2m_nist_invert(const EC_GROUP * group, EC_POINT * point, BN_CTX *ctx);
int ec_GF2m_nist_is_at_infinity(const EC_GROUP * group, const EC_POINT * point);
int ec_GF2m_nist_is_on_curve(const EC_GROUP * group, const EC_POINT * point, BN_CTX *ctx);
int ec_GF2m_nist_cmp(const EC_GROUP * group, const EC_POINT *a, const EC_POINT *b,
BN_CTX *ctx);
int ec_GF2m_nist_make_affine(const EC_GROUP * group, EC_POINT * point, BN_CTX *ctx);
int ec_GF2m_nist_points_make_affine(const EC_GROUP * group, const EC_POINT * points[],
BN_CTX *ctx);
+ int ec_GF2m_nist163_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist163_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_sect193_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_sect193_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_sect233_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_sect233_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist233_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist233_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist283_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist283_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist409_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist409_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist571_field_mul(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);
+ int ec_GF2m_nist571_field_sqr(const EC_GROUP * group, BIGNUM *r, const BIGNUM * a, const
 BIGNUM *b, BN_CTX *ctx);

#endif

```c
#include <stdio.h>
#include <stdint.h>
#include <openssl/bn.h>

#if defined (__INTEL_COMPILER)
#include <ia32intrin.h>
#elif defined (__GNUC__)
#include <emmintrin.h>
#include <smmintrin.h>
#include <immintrin.h>
#include <x86intrin.h>
#endif

/*
********************************************************************************************
* BN <-> XMM CONVERSATIONS
********************************************************************************************
*/

inline void BN_to_XMM_1term ( __m128i z[1] , BN_ULONG *a);
inline void BN_to_XMM_2term ( __m128i z[1] , BN_ULONG *a);
inline void BN_to_XMM_3term ( __m128i z[2] , BN_ULONG *a);
inline void BN_to_XMM_4term ( __m128i z[2] , BN_ULONG *a);
inline void BN_to_XMM_5term ( __m128i z[3] , BN_ULONG *a);
inline void BN_to_XMM_6term ( __m128i z[3] , BN_ULONG *a);
inline void BN_to_XMM_7term ( __m128i z[4] , BN_ULONG *a);
inline void BN_to_XMM_8term ( __m128i z[4] , BN_ULONG *a);
inline void BN_to_XMM_9term ( __m128i z[5] , BN_ULONG *a);

int BN_to_XMM ( __m128i *z, const BIGNUM *a, int dmin );

/* More convenient, but BAD PERFORMANCE */
int BN_to_XMM_conv ( __m128i *z, const BIGNUM *a, int field_size );

/*
********************************************************************************************
* XMM UTILITY
********************************************************************************************
*/

inline void XMM_GF2m_copy_2term ( __m128i z[2] , const __m128i a[2] ) ;
inline void XMM_GF2m_copy_3term ( __m128i z[3] , const __m128i a[3] ) ;
inline void XMM_GF2m_copy_4term ( __m128i z[4] , const __m128i a[4] ) ;
inline void XMM_GF2m_copy_5term ( __m128i z[5] , const __m128i a[5] ) ;
```

/*
  XMM ADDITION
*/
inline void XMM_GF2m_add_2term(__m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_add_3term(__m128i z[3], __m128i a[3], __m128i b[3]);
inline void XMM_GF2m_add_4term(__m128i z[4], __m128i a[4], __m128i b[4]);
inline void XMM_GF2m_add_5term(__m128i z[5], __m128i a[5], __m128i b[5]);
inline void XMM_GF2m_add_7term(__m128i z[7], __m128i a[7], __m128i b[7]);
inline void XMM_GF2m_add_9term(__m128i z[9], __m128i a[9], __m128i b[9]);

/*
  XMM VEILING
*/
inline void XMM_GF2m_veil_2term(__m128i x1[2], __m128i z1[2], __m128i x2[2], __m128i z2[2],
                                __m128i tx1[2], __m128i tz1[2], __m128i tx2[2], __m128i tz2[2], BN_ULONG k);
inline void XMM_GF2m_veil_3term(__m128i x1[3], __m128i z1[3], __m128i x2[3], __m128i z2[3],
                                __m128i tx1[3], __m128i tz1[3], __m128i tx2[3], __m128i tz2[3], BN_ULONG k);
inline void XMM_GF2m_veil_4term(__m128i x1[4], __m128i z1[4], __m128i x2[4], __m128i z2[4],
                                __m128i tx1[4], __m128i tx2[4], __m128i tz2[4], BN_ULONG k);
inline void XMM_GF2m_veil_5term(__m128i x1[5], __m128i z1[5], __m128i x2[5], __m128i z2[5],
                                __m128i tx1[5], __m128i tz1[5], __m128i tx2[5], __m128i tz2[5], BN_ULONG k);

/*
  XMM REDUCTION
*/
inline void XMM_GF2m_mod_sect163(__m128i z[2], __m128i a[3]);
inline void XMM_GF2m_mod_sect163_pclmul(__m128i z[2], __m128i a[3]);
inline void XMM_GF2m_mod_sect233(__m128i z[2], __m128i a[4]);
inline void XMM_GF2m_mod_sect239(__m128i z[2], __m128i a[4]);
inline void XMM_GF2m_mod_sect283_pclmul(__m128i z[3], __m128i a[5]);
inline void XMM_GF2m_mod_sect409(__m128i z[4], __m128i a[7]);
inline void XMM_GF2m_mod_sect571_pclmul(__m128i z[5], __m128i a[9]);

/*
  XMM SQUAREING
*/
inline void XMM_GF2m_sqr_3term(__m128i z[3], const __m128i a[2]);
inline void XMM_GF2m_sqr_4term(__m128i z[4], const __m128i a[2]);
inline void XMM_GF2m_sqr_5term(__m128i z[5], const __m128i a[3]);
inline void XMM_GF2m_sqr_7term(__m128i z[7], const __m128i a[4]);
inline void XMM_GF2m_sqr_9term(__m128i z[9], const __m128i a[5]);
inline void XMM_GF2m_sqr_sect163(__m128i z[2], const __m128i a[2]);
inline void XMM_GF2m_sqr_sect193(__m128i z[2], const __m128i a[2]);
inline void XMM_GF2m_sqr_sect233(__m128i z[2], const __m128i a[2]);
inline void XMM_GF2m_sqr_sect239(__m128i z[2], const __m128i a[2]);
inline void XMM_GF2m_sqr_sect283(__m128i z[3], const __m128i a[3]);
inline void XMM_GF2m_sqr_sect409(__m128i z[4], const __m128i a[4]);
inline void XMM_GF2m_sqr_sect571(__m128i z[5], const __m128i a[5]);

/*
  XMM MULTIPLICATION
*/
inline void XMM_GF2m_2x2_mul ( __m128i z[2], const __m128i a, const __m128i b);
inline void XMM_GF2m_3x3_mul ( __m128i z[3], const __m128i a[2], const __m128i b[2]);
inline void XMM_GF2m_4x4_mul ( __m128i z[4], const __m128i a[2], const __m128i b[2]);
inline void XMM_GF2m_5x5_mul ( __m128i z[5], const __m128i a[3], const __m128i b[3]);
inline void XMM_GF2m_7x7_mul ( __m128i z[7], const __m128i a[4], const __m128i b[4]);
inline void XMM_GF2m_9x9_mul ( __m128i z[9], __m128i a[5], __m128i b[5]);
inline void XMM_GF2m_mod_mul_sect163 ( __m128i z[2], const __m128i a[2], const __m128i b[2]);
inline void XMM_GF2m_mod_mul_sect193 ( __m128i z[2], const __m128i a[2], const __m128i b[2]);
inline void XMM_GF2m_mod_mul_sect233 ( __m128i z[2], const __m128i a[2], const __m128i b[2]);
inline void XMM_GF2m_mod_mul_sect239 ( __m128i z[2], const __m128i a[2], const __m128i b[2]);
inline void XMM_GF2m_mod_mul_sect283 ( __m128i z[3], const __m128i a[3], const __m128i b[3]);
inline void XMM_GF2m_mod_mul_sect409 ( __m128i z[4], const __m128i a[4], const __m128i b[4]);
inline void XMM_GF2m_mod_mul_sect571 ( __m128i z[5], __m128i a[5], __m128i b[5]);

/* XMM LAZY REDUCTION OPERATIONS */

/* Square + Square */
inline void XMM_GF2m_add2sqr_sect163 ( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_add2sqr_sect193 ( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_add2sqr_sect233 ( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_add2sqr_sect239 ( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_add2sqr_sect283 ( __m128i z[3], __m128i a[3], __m128i b[3]);
inline void XMM_GF2m_add2sqr_sect409 ( __m128i z[4], __m128i a[4], __m128i b[4]);
inline void XMM_GF2m_add2sqr_sect571 ( __m128i z[5], __m128i a[5], __m128i b[5]);

/* Square + Multiply */
inline void XMM_GF2m_addsqrmul_sect163 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2]);
inline void XMM_GF2m_addsqrmul_sect193 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2]);
inline void XMM_GF2m_addsqrmul_sect233 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2]);
inline void XMM_GF2m_addsqrmul_sect239 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2]);
inline void XMM_GF2m_addsqrmul_sect283 ( __m128i z[3], __m128i a[3], __m128i b[3], __m128i c[3]);
inline void XMM_GF2m_addsqrmul_sect409 ( __m128i z[4], __m128i a[4], __m128i b[4], __m128i c[4]);
inline void XMM_GF2m_addsqrmul_sect571 ( __m128i z[5], __m128i a[5], __m128i b[5], __m128i c[5]);

/* Multiply + Multiply */
inline void XMM_GF2m_add2mul_sect163 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_add2mul_sect193 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_add2mul_sect233 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_add2mul_sect239 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_add2mul_sect283 ( __m128i z[3], __m128i a[3], __m128i b[3], __m128i c[3], __m128i d[3]);
inline void XMM_GF2m_add2mul_sect409 ( __m128i z[4], __m128i a[4], __m128i b[4], __m128i c[4], __m128i d[4]);
inline void XMM_GF2m_add2mul_sect571 ( __m128i z[5], __m128i a[5], __m128i b[5], __m128i c[5], __m128i d[5]);

/* XMM INVERSION */

/* Square + Multiply */
inline void XMM_GF2m_mod_inv_sect163 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_mod_inv_sect193 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_mod_inv_sect233 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_mod_inv_sect239 ( __m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2]);
inline void XMM_GF2m_mod_inv_sect283 ( __m128i z[3], __m128i a[3], __m128i b[3], __m128i c[3], __m128i d[3]);
inline void XMM_GF2m_mod_inv_sect409 ( __m128i z[4], __m128i a[4], __m128i b[4], __m128i c[4], __m128i d[4]);
inline void XMM_GF2m_mod_inv_sect571 ( __m128i z[5], __m128i a[5], __m128i b[5], __m128i c[5], __m128i d[5]);

/* sDiv */
inline void XMM_GF2m_div_sect163 ( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_div_sect193( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_div_sect233( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_div_sect239( __m128i z[2], __m128i a[2], __m128i b[2]);
inline void XMM_GF2m_div_sect283( __m128i z[3], __m128i a[3], __m128i b[3]);
inline void XMM_GF2m_div_sect409( __m128i z[4], __m128i a[4], __m128i b[4]);
inline void XMM_GF2m_div_sect571( __m128i z[5], __m128i a[5], __m128i b[5]);

/*
 ********************************************************************************************
* BIGNUM WRAPPER FUNCTIONS
 ********************************************************************************************
*/
int BN_GF2m_mod_xmm_sect163( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_pclmul_sect163( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_sect193( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_sect233( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_sect239( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_sect283( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_sect409( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mod_xmm_sect571( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect163( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect193( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect233( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect239( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect283( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect409( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_sqr_xmm_sect571( BIGNUM *z, const BIGNUM *a);
int BN_GF2m_mul_xmm_sect163( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect193( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect233( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect239( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect283( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect409( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_mul_xmm_sect571( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect163( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect193( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect233( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect239( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect283( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect409( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);
int BN_GF2m_div_xmm_sect571( BIGNUM *z, const BIGNUM *a, const BIGNUM *b);

/*
 ********************************************************************************************
* BIGNUM MADDLE FUNCTIONS
 ********************************************************************************************
*/
int BN_GF2m_Maddle_xmm_nist163k( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist163r( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist193r( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist233k( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist233r( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist239k( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist239r( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist283k( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist283r( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist409k( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist409r( const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);
int BN_GF2m_Maddle_xmm_nist571k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, 
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k);
int BN_GF2m_Maddle_xmm_nist571r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, 
    const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c);

/*
 * xmm_gf2m_nist.c
 *
 * Created on: Feb 10, 2013
 * Author: sh1phty
 */
#include <stdio.h>
#include <stdint.h>
#include <openssl/bn.h>
#if defined (__INTEL_COMPILER)
#include <ia32intrin.h>
#elif defined (__GNUC__)
#include <wmmintrin.h> // PCLMUL
#else
#include <smmintrin.h> // SSE
#include <immintrin.h> // AVX
#endif
/* LOOP to unroll squarings */
#define LOOP ( function , number ) for ( i = 0; i < number; i++ ) { function ;}
/* Macros for SSE and AVX compiler intrinsics */
#if defined AVX
#define XOR256 _mm256_xor_pd
#define LOAD256 _mm256_loadu_pd
#define STORE256 _mm256_storeu_pd
#else
#define XOR _mm_xor_si128
#define LOAD128 _mm_load_si128
#define STORE128 _mm_store_si128
#define SET64 _mm_set_epi64x
#define GET64 _mm_extract_epi64
#define XOR _mm_xor_si128
#define AND _mm_and_si128
#define NAND _mm_andnot_si128
#define OR _mm_or_si128
#define SHL _mm_slli_epi64
#define SHR _mm_srli_epi64
#define SHL128 _mm_slli_si128
#define SHR128 _mm_srli_si128
#define ALIGNR _mm_alignr_epi8
#define MOVE64 _mm_move_epi64
#define UNPACKLO8 _mm_unpacklo_epi8
#define UNPACKHI8 _mm_unpackhi_epi8
#define UNPACKLO64 _mm_unpacklo_epi64
#define UNPACKHI64 _mm_unpackhi_epi64
#define ZERO _mm_setzero_si128 ()
#endif

//--------------------------------------------------------
/* BN <-> XMM CONVERSATIONS */
//--------------------------------------------------------
inline void BN_to_XMM_1term(___m128i z[1], BN_ULONG *a)
{
    z[0] = LOAD_64((___m128i*)(a));
}

inline void BN_to_XMM_2term(___m128i z[1], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
}

inline void BN_to_XMM_3term(___m128i z[2], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD_64((___m128i*)(a + 2));
}

inline void BN_to_XMM_4term(___m128i z[2], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD128((___m128i*)(a + 2));
}

inline void BN_to_XMM_5term(___m128i z[3], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD128((___m128i*)(a + 2));
    z[2] = LOAD_64((___m128i*)(a + 4));
}

inline void BN_to_XMM_6term(___m128i z[3], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD128((___m128i*)(a + 2));
    z[2] = LOAD128((___m128i*)(a + 4));
}

inline void BN_to_XMM_7term(___m128i z[4], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD128((___m128i*)(a + 2));
    z[2] = LOAD128((___m128i*)(a + 4));
    z[3] = LOAD_64((___m128i*)(a + 6));
}

inline void BN_to_XMM_8term(___m128i z[4], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD128((___m128i*)(a + 2));
    z[2] = LOAD128((___m128i*)(a + 4));
    z[3] = LOAD128((___m128i*)(a + 6));
}

inline void BN_to_XMM_9term(___m128i z[5], BN_ULONG *a)
{
    z[0] = LOAD128((___m128i*)(a));
    z[1] = LOAD128((___m128i*)(a + 2));
    z[2] = LOAD128((___m128i*)(a + 4));
    z[3] = LOAD128((___m128i*)(a + 6));
    z[4] = LOAD_64((___m128i*)(a + 8));
}

int BN_to_XMM(___m128i *z, const BIGNUM *a, int field_size)
{
    int i, ret=0;

    /* Load BIGNUM to XMM */
    if (a->top == 1) BN_to_XMM_1term(z, a->d);
    else if (a->top == 2) BN_to_XMM_2term(z, a->d);
    else if (a->top == 3) BN_to_XMM_3term(z, a->d);
    else if (a->top == 4) BN_to_XMM_4term(z, a->d);
    else if (a->top == 6) BN_to_XMM_6term(z, a->d);
    else if (a->top == 7) BN_to_XMM_7term(z, a->d);
    else if (a->top == 8) BN_to_XMM_8term(z, a->d);
    else
        BN_to_XMM_9term(z, a->d);

    /* Assert that number of words is fixed for the field */
for (i = a->top+(a->top &1); i < field_size; i+=2) z[i/2] = ZERO;
ret = 1;
return ret;
}

/* More convenient, but BAD PERFORMANCE */
int BN_to_XMM_conv(__m128i *z, const BIGNUM *a, int field_size)
{
    int i, ret=0, top = (a->top+1)/2;
    /* Load bignum to XMM */
    for (i = 0; i < top-((a->top) &1); i++)
    {
        z[i] = LOAD128( (__m128i*) (a->d + 2*i) );
    }
    if ((a->top) &1)
    {
        z[i] = LOAD_64( (__m128i*) (a->d + 2*i) );
    }
    /* Assert that number of words is fixed for the field */
    for (i = a->top+(a->top &1); i < field_size; i+=2) z[i/2] = ZERO;
    ret = 1;
    return ret;
}

inline void XMM_to_BN_1term( BN_ULONG *z, __m128i a[1])
{
    STORE_64( (__m128i*)(z), a[0]);
}

inline void XMM_to_BN_2term( BN_ULONG *z, __m128i a[1])
{
    STORE128( (__m128i*)(z), a[0]);
}

inline void XMM_to_BN_3term( BN_ULONG *z, __m128i a[2])
{
    STORE128( (__m128i*)(z ), a[0]);
    STORE_64( (__m128i*)(z + 2), a[1]);
}

inline void XMM_to_BN_4term( BN_ULONG *z, __m128i a[2])
{
    STORE128( (__m128i*)(z ), a[0]);
    STORE128( (__m128i*)(z + 2), a[1]);
}

inline void XMM_to_BN_5term( BN_ULONG *z, __m128i a[3])
{
    STORE128( (__m128i*)(z ), a[0]);
    STORE128( (__m128i*)(z + 2), a[1]);
    STORE_64( (__m128i*)(z + 4), a[2]);
}

inline void XMM_to_BN_6term( BN_ULONG *z, __m128i a[3])
{
    STORE128( (__m128i*)(z ), a[0]);
    STORE128( (__m128i*)(z + 2), a[1]);
    STORE128( (__m128i*)(z + 4), a[2]);
}

inline void XMM_to_BN_7term( BN_ULONG *z, __m128i a[4])
{
    STORE128( (__m128i*)(z ), a[0]);
    STORE128( (__m128i*)(z + 2), a[1]);
    STORE128( (__m128i*)(z + 4), a[2]);
    STORE_64( (__m128i*)(z + 6), a[3]);
}

inline void XMM_to_BN_8term( BN_ULONG *z, __m128i a[4])
{
    STORE128( (__m128i*)(z ), a[0]);
    STORE128( (__m128i*)(z + 2), a[1]);
    STORE128( (__m128i*)(z + 4), a[2]);
    STORE128( (__m128i*)(z + 6), a[3]);
}
inline void XMM_to_BN_9term( BN_ULONG *z, __m128i a[5])
{
    STORE128( (__m128i *)&z[0], a[0]);
    STORE128( (__m128i *)&z[2], a[1]);
    STORE128( (__m128i *)&z[4], a[2]);
    STORE128( (__m128i *)&z[6], a[3]);
    STORE_64( (__m128i *)&z[8], a[4]);
}

int XMM_to_BN( BIGNUM *z, __m128i *a, int dmin)
{
    int ret=0;
    /* Expand target */
    if (! bn_wexpand(z, dmin)) goto err;
    z->top = dmin;
    /* Store */
    if ( dmin == 1 ) XMM_to_BN_1term(z->d, a);
    else if ( dmin == 2 ) XMM_to_BN_2term(z->d, a);
    else if ( dmin == 3 ) XMM_to_BN_3term(z->d, a);
    else if ( dmin == 4 ) XMM_to_BN_4term(z->d, a);
    else if ( dmin == 5 ) XMM_to_BN_5term(z->d, a);
    else if ( dmin == 6 ) XMM_to_BN_6term(z->d, a);
    else if ( dmin == 7 ) XMM_to_BN_7term(z->d, a);
    else if ( dmin == 8 ) XMM_to_BN_8term(z->d, a);
    else if ( dmin > 8 ) XMM_to_BN_9term(z->d, a);
    /* Correct top */
    bn_correct_top(z);
    ret = 1;
    err:
    return ret;
}

/*
********************************************************************************************
* XMM UTILITY
*********************************************************************************************/

inline void XMM_GF2m_copy_2term( __m128i z[2] , const __m128i a[2])
{
    z[0] = a[0]; z[1] = a[1];
}

inline void XMM_GF2m_copy_3term( __m128i z[3] , const __m128i a[3])
{
    z[0] = a[0]; z[1] = a[1];
    z[2] = a[2];
}

inline void XMM_GF2m_copy_4term( __m128i z[4] , const __m128i a[4])
{
    z[0] = a[0]; z[1] = a[1];
}

inline void XMM_GF2m_copy_5term( __m128i z[5] , const __m128i a[5])
{
    z[0] = a[0]; z[1] = a[1];
    z[4] = a[4];
}

/*
********************************************************************************************
* XMM ADDITION
*********************************************************************************************/

inline void XMM_GF2m_add_2term( __m128i z[2] , __m128i a[2], __m128i b[2])
{
/*
*/
}

#endif AVX

146
z[0] = XOR(a[0], b[0]);
z[1] = XOR(a[1], b[1]);
#else
STORE256((double*)z, (__m256d) XOR256(LOAD256((double*)a), LOAD256((double*)b)));
#endif
}

inline void XMM_GF2m_add_3term(__m128i z[3], __m128i a[3], __m128i b[3])
{
    #ifndef AVX
        z[0] = XOR(a[0], b[0]);
        z[1] = XOR(a[1], b[1]);
    #else
        STORE256((double*)z, (__m256d) XOR256(LOAD256((double*)a), LOAD256((double*)b)));
    #endif
}

inline void XMM_GF2m_add_4term(__m128i z[4], __m128i a[4], __m128i b[4])
{
    #ifndef AVX
        z[0] = XOR(a[0], b[0]);
        z[1] = XOR(a[1], b[1]);
        z[2] = XOR(a[2], b[2]);
        z[3] = XOR(a[3], b[3]);
    #else
        STORE256((double*)z, (__m256d) XOR256(LOAD256((double*)a), LOAD256((double*)b)));
        STORE256((double*)z+4, (__m256d) XOR256(LOAD256((double*)a+4), LOAD256((double*)b+4)));
    #endif
}

inline void XMM_GF2m_add_5term(__m128i z[5], __m128i a[5], __m128i b[5])
{
    #ifndef AVX
        z[0] = XOR(a[0], b[0]);
        z[1] = XOR(a[1], b[1]);
        z[2] = XOR(a[2], b[2]);
        z[3] = XOR(a[3], b[3]);
        z[4] = XOR(a[4], b[4]);
    #else
        STORE256((double*)z, (__m256d) XOR256(LOAD256((double*)a), LOAD256((double*)b)));
        STORE256((double*)z+4, (__m256d) XOR256(LOAD256((double*)a+4), LOAD256((double*)b+4)));
        STORE256((double*)z+8, (__m256d) XOR256(LOAD256((double*)a+8), LOAD256((double*)b+8)));
    #endif
}

inline void XMM_GF2m_add_7term(__m128i z[7], __m128i a[7], __m128i b[7])
{
    #ifndef AVX
        z[0] = XOR(a[0], b[0]);
        z[1] = XOR(a[1], b[1]);
        z[2] = XOR(a[2], b[2]);
        z[3] = XOR(a[3], b[3]);
        z[4] = XOR(a[4], b[4]);
        z[5] = XOR(a[5], b[5]);
    #else
        STORE256((double*)z, (__m256d) XOR256(LOAD256((double*)a), LOAD256((double*)b)));
        STORE256((double*)z+4, (__m256d) XOR256(LOAD256((double*)a+4), LOAD256((double*)b+4)));
        STORE256((double*)z+8, (__m256d) XOR256(LOAD256((double*)a+8), LOAD256((double*)b+8)));
    #endif
}

inline void XMM_GF2m_add_9term(__m128i z[9], __m128i a[9], __m128i b[9])
{
    #ifndef AVX
        z[0] = XOR(a[0], b[0]);
        z[1] = XOR(a[1], b[1]);
        z[2] = XOR(a[2], b[2]);
        z[3] = XOR(a[3], b[3]);
        z[4] = XOR(a[4], b[4]);
        z[5] = XOR(a[5], b[5]);
    #else
        STORE256((double*)z, (__m256d) XOR256(LOAD256((double*)a), LOAD256((double*)b)));
        STORE256((double*)z+4, (__m256d) XOR256(LOAD256((double*)a+4), LOAD256((double*)b+4)));
        STORE256((double*)z+8, (__m256d) XOR256(LOAD256((double*)a+8), LOAD256((double*)b+8)));
    #endif
}
z[6] = XOR(a[6], b[6])

z[7] = XOR(a[7], b[7])

#else
STORE256((double *)z, (__m256d) XOR256((double *)a), LOAD256((double *)b))

STORE256((double *)z+4, (__m256d) XOR256(LOAD256((double *)a+4), LOAD256((double *)b+4)))

STORE256((double *)z+8, (__m256d) XOR256(LOAD256((double *)a+8), LOAD256((double *)b+8)))

STORE256((double *)z+12, (__m256d) XOR256(LOAD256((double *)a+12), LOAD256((double *)b+12)))

#endif

z[8] = XOR(a[8], b[8]);

}

static inline void XMM_GF2m_mask_2term(__m128i z[2], __m128i y[2], __m128i a[2], __m128i b[2], __m128i mask)
{
    z[0] = AND(mask, a[0]);
    z[1] = AND(mask, a[1]);
    y[0] = NAND(mask, b[0]);
    y[1] = NAND(mask, b[1]);
}

static inline void XMM_GF2m_mask_3term(__m128i z[3], __m128i y[3], __m128i a[3], __m128i b[3], __m128i mask)
{
    z[0] = AND(mask, a[0]);
    z[1] = AND(mask, a[1]);
    z[2] = AND(mask, a[2]);
    y[0] = NAND(mask, b[0]);
    y[1] = NAND(mask, b[1]);
    y[2] = NAND(mask, b[2]);
}

static inline void XMM_GF2m_mask_4term(__m128i z[4], __m128i y[4], __m128i a[4], __m128i b[4], __m128i mask)
{
    z[0] = AND(mask, a[0]);
    z[1] = AND(mask, a[1]);
    z[2] = AND(mask, a[2]);
    z[3] = AND(mask, a[3]);
    y[0] = NAND(mask, b[0]);
    y[1] = NAND(mask, b[1]);
    y[2] = NAND(mask, b[2]);
    y[3] = NAND(mask, b[3]);
}

static inline void XMM_GF2m_mask_5term(__m128i z[4], __m128i y[4], __m128i a[4], __m128i b[4], __m128i mask)
{
    z[0] = AND(mask, a[0]);
    z[1] = AND(mask, a[1]);
    z[2] = AND(mask, a[2]);
    z[3] = AND(mask, a[3]);
    z[4] = AND(mask, a[4]);
    y[0] = NAND(mask, b[0]);
    y[1] = NAND(mask, b[1]);
    y[2] = NAND(mask, b[2]);
    y[3] = NAND(mask, b[3]);
    y[4] = NAND(mask, b[4]);
}

inline void XMM_GF2m_veil_2term(__m128i x1[2], __m128i y1[2], __m128i a[2], __m128i b[2], __m128i z1[2], __m128i x2[2], __m128i y2[2], __m128i a2[2], __m128i b2[2], BN_ULONG k)
{
    __m128i mask, t1[2], t2[2];
    BN_ULONG mk;
    mk = (0 - k);
    mask = SET64(mk, mk);
    XMM_GF2m_mask_2term(t1, t2, x1, mask);
    XMM_GF2m_add_2term(a1, t1, t2);
    XMM_GF2m_mask_2term(t1, t2, x2, mask);
}

148
inline void XMM_GF2m_veil_3term(__m128i x1[3], __m128i z1[3], __m128i x2[3], __m128i z2[3], __m128i tx1[3], __m128i tx2[3], __m128i tz1[3], __m128i tz2[3], __m128i mask, BN ULONG k) {
    __m128i mask, t1[3], t2[3];
    BN ULONG nk;
    mk = (0 - k);
    mask = SET64(nk, nk);
    XMM_GF2m_mask_3term(t1, t2, tx1, tx2, mask);
    XMM_GF2m_add_3term(x1, t1, t2);
    XMM_GF2m_mask_3term(t1, t2, tx2, tx1, mask);
    XMM_GF2m_add_3term(x2, t1, t2);
    XMM_GF2m_mask_3term(t1, t2, tz1, tz2, mask);
    XMM_GF2m_add_3term(z1, t1, t2);
    XMM_GF2m_mask_3term(t1, t2, tz2, tz1, mask);
    XMM_GF2m_add_3term(z2, t1, t2);
}

inline void XMM_GF2m_veil_4term(__m128i x1[4], __m128i z1[4], __m128i x2[4], __m128i z2[4], __m128i tx1[4], __m128i tx2[4], __m128i tz1[4], __m128i tz2[4], __m128i mask, BN ULONG k) {
    __m128i mask, t1[4], t2[4];
    BN ULONG nk;
    mk = (0 - k);
    mask = SET64(nk, nk);
    XMM_GF2m_mask_4term(t1, t2, tx1, tx2, mask);
    XMM_GF2m_add_4term(x1, t1, t2);
    XMM_GF2m_mask_4term(t1, t2, tx2, tx1, mask);
    XMM_GF2m_add_4term(x2, t1, t2);
    XMM_GF2m_mask_4term(t1, t2, tz1, tz2, mask);
    XMM_GF2m_add_4term(z1, t1, t2);
    XMM_GF2m_mask_4term(t1, t2, tz2, tz1, mask);
    XMM_GF2m_add_4term(z2, t1, t2);
}

inline void XMM_GF2m_veil_5term(__m128i x1[5], __m128i z1[5], __m128i x2[5], __m128i z2[5], __m128i tx1[5], __m128i tx2[5], __m128i tz1[5], __m128i tz2[5], __m128i mask, BN ULONG k) {
    __m128i mask, t1[5], t2[5];
    BN ULONG nk;
    mk = (0 - k);
    mask = SET64(nk, nk);
    XMM_GF2m_mask_5term(t1, t2, tx1, tx2, mask);
    XMM_GF2m_add_5term(x1, t1, t2);
    XMM_GF2m_mask_5term(t1, t2, tx2, tx1, mask);
    XMM_GF2m_add_5term(x2, t1, t2);
    XMM_GF2m_mask_5term(t1, t2, tz1, tz2, mask);
    XMM_GF2m_add_5term(z1, t1, t2);
    XMM_GF2m_mask_5term(t1, t2, tz2, tz1, mask);
    XMM_GF2m_add_5term(z2, t1, t2);
}
inline void XMM_GF2m_mod_sect163(__m128i z[2], __m128i a[3])
{
    __m128i x[5];
    x[0] = SHR(a[2], 35);
    x[1] = SHL(a[2], 29);
    x[3] = SHL128(a[2], 4);
    x[1] = XOR(x[1], x[3]);
    x[2] = SHR(a[2], 29);
    x[3] = SHL(a[2], 36);
    x[0] = XOR(x[0], x[2]);
    x[1] = XOR(x[1], x[3]);
    x[2] = SHR(a[2], 28);
    x[3] = SHL(a[2], 35);
    x[0] = XOR(x[0], x[2]);
    x[1] = XOR(x[1], x[3]);
    x[2] = SHL128(x[1], 8);
    x[1] = SHR128(x[1], 8);
    x[0] = XOR(x[0], x[1]);
    z[0] = XOR(a[0], x[2]);
    z[1] = XOR(a[1], x[0]);
}

inline void XMM_GF2m_mod_sect163_pclmul (__m128i z[2], __m128i a[3])
{
    __m128i _p, x[4];
    _p = SET64(0, 0x00000000000000C9);
    x[0] = SHR(a[2], 35);
    x[1] = SHL(a[2], 29);
    x[2] = SHL128(x[1], 8);
    x[0] = XOR(x[0], x[2]);
    x[3] = PCLMUL(x[0], _p, 0x00);
    z[1] = XOR(a[1], x[3]);
    x[0] = SET64(0x0000000000000000, 0x000000000000C9);
    x[2] = AND(x[0], x[1]);
    z[1] = NAND(x[0], z[1]);
    x[0] = SHR(x[2], 35);
    x[2] = SHL(x[2], 29);
    x[2] = ALIGNR(x[1], x[2], 8);
inline void XMM_GF2m_mod_sect193(__m128i z[2], __m128i a[4])
{
    __m128i x[5];

    x[0] = SHL (a[3], 14);
    x[1] = SHR (a[3], 1);
    x[2] = SHL (a[3], 63);
    z[1] = XOR (a[1], x[2]);
    x[4] = XOR (x[0], x[1]);

    // ------------------------------------------
    x[0] = SHR (a[2], 50);
    x[1] = SHL (a[2], 14);
    x[2] = SHR (a[2], 1);
    x[3] = SHL (a[2], 63);
    z[1] = XOR (z[1], x[0]);
    z[0] = XOR (a[0], x[3]);
    x[4] = ALIGNR (x[4], x[0], 8);
    z[1] = XOR (z[1], x[4]);

    // ------------------------------------------
    /* Clear top */
    x[3] = SET64 (0x0000000000000001, 0xFFFFFFFFFFFFFF);
    x[4] = NAND (x[3], z[1]);
    z[1] = AND (z[1], x[3]);
    x[0] = SHR (x[0], 1);
    z[0] = XOR (z[0], x[0]);
    x[0] = XOR (x[1], x[2]);
    x[4] = NAND (x[4], x[0]);
    z[1] = XOR (z[1], x[4]);
    x[2] = SHR (x[2], 50);
    z[0] = XOR (z[0], x[2]);
}

inline void XMM_GF2m_mod_sect233(__m128i z[2], __m128i a[4])
{
    /* Init */
    __m128i x[6];

    /* Clear first chunk */
    x[0] = SHL (a[3], 33);
    x[1] = SHR (a[3], 31);
    x[2] = SHL (a[3], 31);
    x[3] = SHR128 (x[4], 8);
    z[1] = XOR (a[1], x[2]);
    a[2] = XOR (a[2], x[1]);
    a[2] = XOR (a[2], x[3]);

    // ------------------------------------------
    /* Clear second chunk */
    x[0] = SHL (a[2], 33);
    x[1] = SHR (a[2], 31);
    x[2] = SHL (a[2], 31);
    x[3] = SHR (a[2], 41);
    x[6] = XOR (x[0], x[3]);
    x[3] = ALIGNR (x[4], x[6], 8);
    z[0] = XOR (z[0], x[3]);
    z[1] = XOR (z[1], x[1]);
    z[1] = XOR (z[1], x[3]);
// Clear top /

/* Clear top */

x[2] = SET64( 0x000001FFFFFFFFF, 0x0FFFFFFFFFFFF);
x[0] = NAND( x[2], z[1] );
x[0] = SHR( x[0], 41);
x[1] = ALIGNR( x[0], x[0], 8);
z[0] = XOR( z[0], x[1] );
x[1] = SHR( z[0], 10);
x[0] = XOR( z[0], x[1] );
z[1] = AND( x[1], x[2] );

/* Only wrapper function for the use with other XMM functions */

inline void XMM_GF2m_mod_sect239( __m128i z[2], __m128i a[4])
{
    uint64_t zz, w[8];
    __m128i x[4];
    x[0] = ALIGNR( a[3], a[2], 8);
    x[1] = ALIGNR( a[4], a[3], 8);
    x[3] = a[2];
    a[4] = SHR( a[4], 27);
    a[3] = SHR( a[3], 27);
    x[2] = SHL( x[1], 37);
    a[3] = XOR( a[3], x[2] );
    a[2] = SHR( a[2], 27);
    x[2] = SHL( x[0], 37);
    a[2] = XOR( a[2], x[2] );
    x[0] = ALIGNR( a[4], a[3], 16);
    x[2] = SHR( x[0], 1);
a[4] = XOR( a[4], x[2]);
x[1] = ALIGNR( a[3], a[2], 8);
x[2] = SHL( a[3], 7);
a[3] = XOR( a[3], x[1]);
x[2] = SHR( x[1], 57);
a[3] = XOR( a[3], x[2]);
x[0] = ALIGNR( a[2], ZERO, 8);
x[2] = SHL( a[2], 7);
a[2] = XOR( a[2], x[2]);
x[2] = SHR( x[0], 57);
a[2] = XOR( a[2], x[2]);
x[0] = ALIGNR( a[4], a[3], 15);
x[1] = SHR( x[0], 3);
a[4] = XOR( a[4], x[1]);
x[1] = ALIGNR( a[3], a[2], 8);
x[2] = SHL( a[3], 6);
a[3] = XOR( a[3], x[2]);
x[2] = SHR( x[1], 59);
a[3] = XOR( a[3], x[2]);
x[0] = ALIGNR( a[2], ZERO, 8);
x[2] = SHL( a[2], 6);
a[2] = XOR( a[2], x[2]);
x[2] = SHR( x[0], 59);
a[2] = XOR( a[2], x[2]);
z[0] = XOR( a[0], a[2]);
z[1] = XOR( a[1], a[3]);
z[2] = XOR( x[3], a[4]);

// Init

/* Do multiplication for lower half */
x[0] = PCLMUL( a[4], _p, 0x00);
x[2] = SHR128( x[0], 8);
z[2] = XOR( a[2], x[2]);
/* Do multiplication for upper half */
x[0] = PCLMUL( a[3], _p, 0x01);
z[1] = XOR( x[0], a[1]);
/* Do multiplication for lower half */
x[1] = PCLMUL( a[3], _p, 0x00);
x[2] = ALIGNR( x[3], x[1], 8);
z[1] = XOR( x[2], z[1]);
/* Final round of reduction */
x[0] = SHL128( x[1], 8);
z[0] = XOR( x[0], a[0]);
x[2] = SET64( 0x0000000000000000, 0x0000000007FFFFFF);
x[3] = NAND( x[2], z[2]);
z[2] = AND( z[2], x[2]);
x[1] = PCLMUL( x[3], _p, 0x01);
z[0] = XOR( x[1], z[0]);
```c
inline void XMM_GF2m_mod_sect409(__m128i z[4], __m128i a[7]) {
    /* Init */
    __m128i x[3], m[12];
    /* Shift top of reminder */
    m[0] = SHR( a[6], 2);
    m[1] = SHL( a[6], 62);
    m[2] = SHR( a[6], 25);
    m[3] = SHL( a[6], 39);
    m[4] = SHR( a[5], 2);
    m[5] = SHL( a[5], 62);
    m[6] = SHR( a[5], 25);
    m[7] = SHL( a[5], 39);
    m[8] = SHR( a[4], 2);
    m[9] = SHL( a[4], 62);
    m[10] = SHR( a[4], 25);
    m[11] = SHL( a[4], 39);
    // ------------------------------------------
    /* Xor to reminder */
    x[0] = XOR( m[1], m[2]);
    z[3] = XOR( a[3], x[0]);
    x[1] = XOR( m[4], m[3]);
    z[3] = XOR( z[3], x[2]);
    x[0] = XOR( m[5], m[6]);
    z[2] = XOR( a[2], x[0]);
    m[7] = XOR( m[7], m[8]);
    x[1] = XOR( x[1], m[7]);
    z[2] = XOR( z[2], x[1]);
    x[2] = XOR( m[9], m[10]);
    z[1] = XOR( a[1], x[2]);
    // ------------------------------------------
    /* Clear last chunk */
    x[0] = SET64( 0xFFFFFFFFFFFFFFFF, 0xFFFFFFFFFE000000);
    z[3] = XOR( z[3], x[0]);
    m[0] = SHR( x[0], 2);
    m[1] = SHL( x[0], 62);
    m[2] = SHR( x[0], 25);
    m[3] = SHL( x[0], 39);
    x[0] = XOR( m[11], m[0]);
    x[1] = XOR( x[1], m[11]);
    x[2] = XOR( m[1], m[2]);
    z[0] = XOR( a[0], x[2]);
    z[0] = XOR( z[0], x[0]);
}

inline void XMM_GF2m_mod_sect571(__m128i z[5], __m128i a[9]) {
    const int top = 9, n = 4;
    int i;
    __m128i x[5];
    /* Reduce */
    x[4] = ZERO;
    for (i = (top-1); i > n; i--)
    {
        // Component -
        x[0] = SHL( a[3], 6);
        x[1] = SHR( a[1], 69);
```
XMM_GF2m_mod_sect571pclmul ( __m128i z[5], __m128i a[9])
{
    __m128i _p, x[10];
    _p = SET64( 0, 0x00000000000084A0);
    // Do multiplication */
    x[0] = PCLMUL( a[0], _p, 0x01);
}
x[1] = PCLMUL( a[8], _p, 0x00);
x[2] = PCLMUL( a[7], _p, 0x01);
x[3] = PCLMUL( a[7], _p, 0x00);
x[4] = PCLMUL( a[6], _p, 0x01);
x[5] = PCLMUL( a[6], _p, 0x00);
x[6] = PCLMUL( a[6], _p, 0x01);
x[7] = PCLMUL( a[6], _p, 0x00);

/* Add result to reminder */
a[4] = XOR( x[0], a[4]);
a[3] = XOR( x[2], a[3]);
a[2] = XOR( x[4], a[2]);
a[1] = XOR( x[6], a[1]);
x[8] = SHR128( x[1], 8);
a[4] = XOR( x[8], a[4]);
x[9] = ALIGNSR( x[1], x[3], 8);
a[3] = XOR( x[9], a[3]);
x[8] = ALIGNSR( x[3], x[5], 8);
a[2] = XOR( x[8], a[2]);
x[9] = ALIGNSR( x[5], x[7], 8);
a[1] = XOR( x[9], a[1]);
x[0] = SHL128( x[7], 8);
a[0] = XOR( x[0], a[0]);

/* final round of reduction */
x[3] = SET64( 0xfffffffffffffff, 0x8000000000000000);
x[3] = AND( a[4], x[3]);
a[4] = XOR( a[4], x[3]);
x[1] = PCLMUL( x[3], _p, 0x01);
a[0] = XOR( x[1], a[0]);
x[2] = PCLMUL( x[3], _p, 0x00);
x[2] = SHR128( x[2], 8);
a[0] = XOR( x[2], a[0]);

XMM_GF2m_copy_5term(z, a);

/* -------------------------------------------------------------------------- */

* XMM SQUAREING
* -------------------------------------------------------------------------- */

inline void XMM_GF2m_sqr_3term(__m128i z[3], const __m128i a[2])
{
  /* Init */
  __m128i x[2], sqrT, mask;
  sqrT = SET64( 0x6554515045444140, 0x1514111000000000 );
  mask = SET64( 0x0f0f0f0f0f0f0f0f, 0x0f0f0f0f0f0f0f0f );
  x[0] = AND( a[0], mask);
  x[1] = SHR( a[0], 4);
  x[1] = AND( x[1], mask);
  x[0] = SHUFFLE( sqrT, x[0] );
  x[1] = SHUFFLE( sqrT, x[1] );
  z[0] = UNPACKLO8( x[0], x[1] );
  z[1] = UNPACKHI8( x[0], x[1] );
  x[0] = AND( a[1], mask);
  x[1] = SHR( a[1], 4);
  x[1] = AND( x[1], mask);
  x[0] = SHUFFLE( sqrT, x[0] );
  x[1] = SHUFFLE( sqrT, x[1] );
  z[2] = UNPACKLO8( x[0], x[1] );
}

inline void XMM_GF2m_sqr_4term(__m128i z[4], const __m128i a[2])
{
  /* Init */
  __m128i x[2], sqrT, mask;
  sqrT = SET64( 0x6554515045444140, 0x1514111000000000 );
  mask = SET64( 0x0f0f0f0f0f0f0f0f, 0x0f0f0f0f0f0f0f0f );

inline void XMM_GF2m_sqr_5term(__m128i z[5], const __m128i a[3]) {
    /* Init */
    __m128i x[2], sqrT, mask;
    sqrT = SET64(0x5554515045444140, 0x1514111005040100);
    mask = SET64(0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F);
    x[0] = AND(a[0], mask);
    x[1] = SHR(a[0], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE(sqrT, x[0]);
    z[0] = UNPACHLD8(x[0], x[1]);
    z[1] = UNPACHH8(x[0], x[1]);
    x[0] = AND(a[1], mask);
    x[1] = SHR(a[1], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE(sqrT, x[0]);
    z[0] = UNPACHLD8(x[0], x[1]);
    z[1] = UNPACHH8(x[0], x[1]);
    x[0] = AND(a[2], mask);
    x[1] = SHR(a[2], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE(sqrT, x[0]);
    z[2] = UNPACHLD8(x[0], x[1]);
    z[3] = UNPACHH8(x[0], x[1]);
};

inline void XMM_GF2m_sqr_7term(__m128i z[7], const __m128i a[4]) {
    /* Init */
    __m128i x[2], sqrT, mask;
    sqrT = SET64(0x5554515045444140, 0x1514111005040100);
    mask = SET64(0x0F0F0F0F0F0F0F0F, 0x0F0F0F0F0F0F0F0F);
    x[0] = AND(a[0], mask);
    x[1] = SHR(a[0], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE(sqrT, x[0]);
    z[0] = UNPACHLD8(x[0], x[1]);
    z[1] = UNPACHH8(x[0], x[1]);
    x[0] = AND(a[1], mask);
    x[1] = SHR(a[1], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE(sqrT, x[0]);
    z[2] = UNPACHLD8(x[0], x[1]);
    z[3] = UNPACHH8(x[0], x[1]);
    x[0] = AND(a[2], mask);
    x[1] = SHR(a[2], 4);
    x[1] = AND(x[1], mask);
    x[0] = SHUFFLE(sqrT, x[0]);
}
```c
inline void XMM_GF2m_sqr_3term ( __m128i z[3] , const __m128i a[3])
{
    /* Init */
    __m128i x[2] , sqrT , mask ;
    sqrT = SET64 ( 0x6545454545454545 , 0x0000000000000000 ) ;
    mask = SET64 ( 0x0000000000000000 , 0x0000000000000000 ) ;
    x[0] = AND ( a[0] , mask );
    x[1] = SHR ( a[0] , 4);  
    x[1] = AND ( x[1] , mask );
    x[0] = SHUFFLE ( sqrT , x[0] );
    x[1] = SHUFFLE ( sqrT , x[1] );
    z[0] = UNPACKLO8 ( x[0] , x[1] ) ;
    z[1] = UNPACKHI8 ( x[0] , x[1] ) ;
    x[0] = AND ( a[1] , mask );
    x[1] = SHR ( a[1] , 4);  
    x[1] = AND ( x[1] , mask );
    x[0] = SHUFFLE ( sqrT , x[0] );
    x[1] = SHUFFLE ( sqrT , x[1] );
    z[2] = UNPACKLO8 ( x[0] , x[1] ) ;
    z[3] = UNPACKHI8 ( x[0] , x[1] ) ;
    x[0] = AND ( a[2] , mask );
    x[1] = SHR ( a[2] , 4);  
    x[1] = AND ( x[1] , mask );
    x[0] = SHUFFLE ( sqrT , x[0] );
    x[1] = SHUFFLE ( sqrT , x[1] );
    z[4] = UNPACKLO8 ( x[0] , x[1] ) ;
    z[5] = UNPACKHI8 ( x[0] , x[1] ) ;
    x[0] = AND ( a[3] , mask );
    x[1] = SHR ( a[3] , 4);  
    x[1] = AND ( x[1] , mask );
    x[0] = SHUFFLE ( sqrT , x[0] );
    x[1] = SHUFFLE ( sqrT , x[1] );
    z[6] = UNPACKLO8 ( x[0] , x[1] ) ;
    z[7] = UNPACKHI8 ( x[0] , x[1] ) ;
    x[0] = AND ( a[4] , mask );
    x[1] = SHR ( a[4] , 4);  
    x[1] = AND ( x[1] , mask );
    x[0] = SHUFFLE ( sqrT , x[0] );
    x[1] = SHUFFLE ( sqrT , x[1] );
    z[8] = UNPACKLO8 ( x[0] , x[1] ) ;
}
```
inline void XMM_GF2m_sqr_sect233( __m128i z[2], const __m128i a[2])
{
    __m128i t[4];
    /* Square */
    XMM_GF2m_sqr_4term( t, a);
    /* Reduce */
    XMM_GF2m_mod_sect233( z, t);
}

inline void XMM_GF2m_sqr_sect239( __m128i z[2], const __m128i a[2])
{
    __m128i t[4];
    /* Square */
    XMM_GF2m_sqr_4term( t, a);
    /* Reduce */
    XMM_GF2m_mod_sect239( z, t);
}

inline void XMM_GF2m_sqr_sect283( __m128i z[3], const __m128i a[3])
{
    __m128i t[5];
    /* Square */
    XMM_GF2m_sqr_5term( t, a);
    /* Reduce */
    XMM_GF2m_mod_sect283( z, t);
}

inline void XMM_GF2m_sqr_sect409( __m128i z[4], const __m128i a[4])
{
    __m128i t[7];
    /* Square */
    XMM_GF2m_sqr_7term( t, a);
    /* Reduce */
    XMM_GF2m_mod_sect409( z, t);
}

inline void XMM_GF2m_sqr_sect571( __m128i z[5], const __m128i a[5])
{
    __m128i t[9];
    /* Square */
    XMM_GF2m_sqr_9term( t, a);
    /* Reduce */
    XMM_GF2m_mod_sect571( z, t);
}

/**
   ******************************************************
   */

inline void XMM_GF2m_2x2_mul(__m128i z[2], const __m128i a, const __m128i b)
{
    /* Init */
    __m128i x[4];
    x[0] = SHR128( a, 8);
    x[1] = XOR( a, x[0]);
    x[2] = SHR128( b, 8);
    x[3] = XOR( b, x[2]);
z[0] = PCLMUL(a, b, 0x00);
z[1] = PCLMUL(a, b, 0x11);
x[0] = PCLMUL(x[1], x[3], 0x00);

x[1] = XOR(z[0], z[1]);
x[0] = XOR(x[0], x[1]);

x[1] = SHL128(x[0], 8);
x[2] = SHR128(x[0], 8);

z[0] = XOR(z[0], x[1]);
z[1] = XOR(z[1], x[2]);

inline void XMM_GF2m_3x3_mul(__m128i z[3], const __m128i a[2], const __m128i b[2])
{
    /* Init */
    __m128i m[3], t[4];

    t[0] = ALIGNR(a[1], a[0], 8);
    t[1] = XOR(a[0], t[0]); // t1 = [ a2+d1 | a1+a0 ]
    t[2] = ALIGNR(b[1], b[0], 8);
    t[3] = XOR(b[0], t[2]); // t3 = [ b2+b1 | b1+b0 ]
    t[0] = XOR(a[0], a[1]); // t0 = [ ##### | a0+a2 ]
    t[2] = XOR(b[0], b[1]); // t2 = [ ##### | b0+b2 ]

    z[0] = PCLMUL(a[0], b[0], 0x00); // z0 = a0*b0
    z[1] = PCLMUL(a[0], b[0], 0x11); // z1 = a1*b1
    z[2] = PCLMUL(a[1], b[1], 0x00); // z2 = a2*b2
    z[0] = PCLMUL(t[1], t[3], 0x00); // z0 = (a1+a0)(b1+b0)
    z[1] = PCLMUL(t[2], t[0], 0x00); // z1 = (a2+a0)(b2+b0)
    m[0] = PCLMUL(t[1], t[3], 0x00); // m0 = (a1+a0)(b1+b0)
    m[1] = PCLMUL(t[2], t[0], 0x00); // m1 = (a2+a0)(b2+b0)

    m[0] = XOR(m[0], z[0]); // m0 = a0*b1 + a1*b0
    m[0] = XOR(m[0], z[1]); // m1 = a0*b2 + a2*b0
    m[1] = XOR(m[1], z[0]); // m1 = a0*b1 + a1*b0
    m[1] = XOR(m[1], z[1]); // m2 = a0*b2 + a2*b0

    t[0] = SHL128(m[0], 8);
    z[0] = XOR(z[0], t[0]);
    z[1] = XOR(z[1], t[1]);
    z[2] = XOR(z[2], t[2]);
}

inline void XMM_GF2m_4x4_mul(__m128i z[4], const __m128i a[2], const __m128i b[2])
{
    /* Init */
    __m128i m[4], t[4];

    XMM_GF2m_2x2_mul(z, a[0], b[0]);
    XMM_GF2m_2x2_mul(z+2, a[1], b[1]);
    t[2] = XOR(a[0], a[1]);
    t[3] = XOR(b[0], b[1]);

    t[0] = XOR(t[0], z[0]);
    t[0] = XOR(t[0], z[2]);
    t[1] = XOR(t[1], z[1]);
    t[1] = XOR(t[1], z[3]);
    z[1] = XOR(z[1], t[0]);
    z[2] = XOR(z[2], t[1]);
}

inline void XMM_GF2m_5x5_mul(__m128i z[5], const __m128i a[3], const __m128i b[3])
{
    __m128i m[13], t[13];
    /* Prepare temporary operands */
\[
\begin{align*}
    t[0] &= \text{UNPACKLO64}(a[0], b[0]); \\
    t[1] &= \text{UNPACKLO64}(a[0], b[0]); \\
    t[2] &= \text{UNPACKLO64}(a[1], b[1]); \\
    t[3] &= \text{UNPACKLO64}(a[1], b[1]); \\
    t[4] &= \text{UNPACKLO64}(a[2], b[2]); \\
    t[5] &= \text{XOR}(t[0], t[1]); \\
    t[6] &= \text{XOR}(t[0], t[2]); \\
    t[7] &= \text{XOR}(t[2], t[4]); \\
    t[8] &= \text{XOR}(t[3], t[4]); \\
    t[9] &= \text{XOR}(t[3], t[6]); \\
    t[10] &= \text{XOR}(t[1], t[7]); \\
    t[11] &= \text{XOR}(t[5], t[8]); \\
    t[12] &= \text{XOR}(t[2], t[11]); \\
    \end{align*}
\]
/* Do multiplications */
\[
\begin{align*}
    m[0] &= \text{PCLMUL}(t[0], t[0], 0x01); \\
    m[1] &= \text{PCLMUL}(t[1], t[1], 0x01); \\
    m[2] &= \text{PCLMUL}(t[2], t[2], 0x01); \\
    m[3] &= \text{PCLMUL}(t[3], t[3], 0x01); \\
    m[4] &= \text{PCLMUL}(t[4], t[4], 0x01); \\
    m[5] &= \text{PCLMUL}(t[5], t[5], 0x01); \\
    m[6] &= \text{PCLMUL}(t[6], t[6], 0x01); \\
    m[7] &= \text{PCLMUL}(t[7], t[7], 0x01); \\
    m[8] &= \text{PCLMUL}(t[8], t[8], 0x01); \\
    m[9] &= \text{PCLMUL}(t[9], t[9], 0x01); \\
    m[10] &= \text{PCLMUL}(t[10], t[10], 0x01); \\
    m[11] &= \text{PCLMUL}(t[11], t[11], 0x01); \\
    m[12] &= \text{PCLMUL}(t[12], t[12], 0x01); \\
    \end{align*}
\]
/* Combine results */
\[
\begin{align*}
    t[0] &= m[0]; \\
    t[8] &= m[4]; \\
    t[1] &= \text{XOR}(t[0], m[1]); \\
    t[2] &= \text{XOR}(t[1], m[6]); \\
    t[1] &= \text{XOR}(t[1], m[6]); \\
    t[2] &= \text{XOR}(t[2], m[2]); \\
    t[7] &= \text{XOR}(t[8], m[9]); \\
    t[6] &= \text{XOR}(t[7], m[7]); \\
    t[7] &= \text{XOR}(t[7], m[8]); \\
    t[6] &= \text{XOR}(t[6], m[2]); \\
    t[6] &= \text{XOR}(m[11], m[12]); \\
    t[3] &= \text{XOR}(t[5], m[9]); \\
    t[3] &= \text{XOR}(t[3], t[0]); \\
    t[3] &= \text{XOR}(t[3], t[6]); \\
    t[4] &= \text{XOR}(t[1], t[7]); \\
    t[4] &= \text{XOR}(t[4], m[9]); \\
    t[4] &= \text{XOR}(t[4], m[10]); \\
    t[4] &= \text{XOR}(t[4], m[12]); \\
    t[5] &= \text{XOR}(t[5], t[2]); \\
    t[5] &= \text{XOR}(t[6], t[8]); \\
    t[5] &= \text{XOR}(t[8], m[10]); \\
    t[9] &= \text{SHR128}(t[7], 8); \\
    t[7] &= \text{ALIGNR}(t[7], t[6], 8); \\
    t[6] &= \text{ALIGNR}(t[5], t[3], 8); \\
    t[3] &= \text{ALIGNR}(t[3], t[1], 8); \\
    t[1] &= \text{SHL128}(t[1], 8); \\
    z[0] &= \text{XOR}(t[0], t[1]); \\
    z[1] &= \text{XOR}(t[2], t[3]); \\
    z[2] &= \text{XOR}(t[4], t[5]); \\
    z[3] &= \text{XOR}(t[6], t[7]); \\
    z[4] &= \text{XOR}(t[8], t[9]); \\
\end{align*}
\]

```c
inline void XMM_GF2m_7x7_mul(__m128i z[7], const __m128i a[4], const __m128i b[4])
{
    __m128i t[4], e[4];
    /* Multiply lower part */
    XMM_GF2m_4x4_mul(z, a, b);
    /* Multiply upper part */
    XMM_GF2m_3x3_mul(z+4, a+2, b+2);
}
```
t[0] = XOR(a[0], a[2]);
t[1] = XOR(a[1], a[3]);
t[2] = XOR(b[0], b[2]);
t[3] = XOR(b[1], b[3]);

/* Multiply middle part */
XMM_GF2m_4x4_mul(e, t+2, t);

/* Combine results */
t[0] = XOR(e[0], z[4]);
t[1] = XOR(e[1], z[5]);
t[2] = XOR(e[2], z[6]);
t[3] = XOR(e[3], z[3]);
e[0] = XOR(t[0], z[0]);
e[1] = XOR(t[1], z[1]);
e[2] = XOR(t[2], z[2]);
z[2] = XOR(z[2], e[0]);
z[3] = XOR(z[3], e[1]);
z[4] = XOR(z[4], e[2]);
z[5] = XOR(z[5], t[3]);

/** Multiply lower part */

// Make local copy of a,b to not destroy them */
at[4] = ALIGNR(a[4], a[3], 8);
at[3] = ALIGNR(a[3], a[2], 8);
at[2] = MOVE64(a[2]);
XMM_GF2m_copy_2term(at, a);
bt[4] = ALIGNR(b[4], b[3], 8);
bt[3] = ALIGNR(b[3], b[2], 8);
bt[2] = MOVE64(b[2]);
XMM_GF2m_copy_2term(bt, b);

/* Prepare operands */
t[0] = XOR(at[0], at[3]); // t0 = [(a6+a1);(a5+a0)]
t[1] = XOR(at[1], at[4]); // t1 = [(a8+a3);(a7+a2)]
t[2] = at[2]; // t2 = (0;a4)
e[0] = XOR(bt[0], bt[3]); // e0 = [(b6+b1);(b5+b0)]
e[1] = XOR(bt[1], bt[4]); // e1 = [(b8+b3);(b7+b2)]
e[2] = bt[2]; // e2 = (0;b4)

/* Multiply middle part */
XMM_GF2m_5x5_mul(f, t, e);

/* Multiply upper part */
XMM_GF2m_4x4_mul(e+5, at+3, bt+3);

/* Combine results */
e[0] = XOR(t[0], z[5]);
e[1] = XOR(t[1], z[6]);
e[2] = XOR(t[2], z[7]);
e[3] = XOR(t[3], z[8]);
f[0] = SHL128(e[0], 8);
z[2] = XOR(z[2], f[0]);
f[1] = ALIGNR(e[1], e[0], 8);
z[3] = XOR(z[3], f[1]);
f[2] = ALIGNR(e[2], e[1], 8);
z[4] = XOR(z[4], f[2]);
f[3] = ALIGNR(e[3], e[2], 8);
z[5] = XOR(z[5], f[3]);
f[4] = ALIGNR(t[4], e[3], 8);
inline void XMM_GF2m_mod_mul_sect163(__m128i z[2], const __m128i a[2], const __m128i b[2])
{
    __m128i t[3];
    // Do multiplication
    XMM_GF2m_3x3_mul(t, a, b);
    // Reduce
    XMM_GF2m_mod sect163(z, t);
}

inline void XMM_GF2m_mod_mul_sect193(__m128i z[2], const __m128i a[2], const __m128i b[2])
{
    __m128i t[4];
    // Do multiplication
    XMM_GF2m_4x4_mul(t, a, b);
    // Reduce
    XMM_GF2m_mod sect193(z, t);
}

inline void XMM_GF2m_mod_mul_sect233(__m128i z[2], const __m128i a[2], const __m128i b[2])
{
    __m128i t[4];
    // Do multiplication
    XMM_GF2m_4x4_mul(t, a, b);
    // Reduce
    XMM_GF2m_mod sect233(z, t);
}

inline void XMM_GF2m_mod_mul_sect239(__m128i z[2], const __m128i a[2], const __m128i b[2])
{
    __m128i t[4];
    // Do multiplication
    XMM_GF2m_4x4_mul(t, a, b);
    // Reduce
    XMM_GF2m_mod sect239(z, t);
}

inline void XMM_GF2m_mod_mul_sect283(__m128i z[3], const __m128i a[3], const __m128i b[3])
{
    __m128i t[5];
    // Do multiplication
    XMM_GF2m_5x5_mul(t, a, b);
    // Reduce
    XMM_GF2m_mod sect283(z, t);
}

inline void XMM_GF2m_mod_mul_sect409(__m128i z[4], const __m128i a[4], const __m128i b[4])
{
    __m128i t[7];
    // Do multiplication
    XMM_GF2m_7x7_mul(t, a, b);
    // Reduce
    XMM_GF2m_mod sect409(z, t);
}
inline void XMM_GF2m_mod_mul_sect571(__m128i z[5], const __m128i a[5], const __m128i b[5])
{
    __m128i t[9];
    // Do multiplication */
    XMM_GF2m_9x9_mul(t, a, b);
    // Reduce */
    XMM_GF2m_mod_sect571(z, t);
}

/*
*********************************************************************************
* XMM LAZY REDUCTION OPERATIONS
*********************************************************************************/

// Square + Multiply */
inline void XMM_GF2m_addsqrmul_sect163(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2])
{
    /* Init */
    __m128i t0[3], t1[3];
    /* Square */
    XMM_GF2m_sqr_3term(t0, a);
    /* Multiply */
    XMM_GF2m_3x3_mul(t1, b, c);
    /* Add */
    XMM_GF2m_add_3term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect163(z, t0);
}

inline void XMM_GF2m_addsqrmul_sect193(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2])
{
    /* Init */
    __m128i t0[4], t1[4];
    /* Square */
    XMM_GF2m_sqr_4term(t0, a);
    /* Multiply */
    XMM_GF2m_4x4_mul(t1, b, c);
    /* Add */
    XMM_GF2m_add_4term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect193(z, t0);
}

inline void XMM_GF2m_addsqrmul_sect233(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2])
{
    /* Init */
    __m128i t0[4], t1[4];
    /* Square */
    XMM_GF2m_sqr_4term(t0, a);
    /* Multiply */
    XMM_GF2m_4x4_mul(t1, b, c);
    /* Add */
    XMM_GF2m_add_4term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect233(z, t0);
}

inline void XMM_GF2m_addsqrmul_sect239(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2])
{


```c
/* Init */
__m128i t0[4], t1[4];
/* Square */
XMM_GF2m_sqr_4term(t0, a);
/* Multiply */
XMM_GF2m_4x4_mul(t1, b, c);
/* Add */
XMM_GF2m_add_4term(t0, t0, t1);
/* Reduce */
XMM_GF2m_mod_sect239(z, t0);
}

inline void XMM_GF2m_addsqrmul_sect283(__m128i z[3], __m128i a[3], __m128i b[3], __m128i c[3])
{
/* Init */
__m128i t0[5], t1[5];
/* Square */
XMM_GF2m_sqr_5term(t0, a);
/* Multiply */
XMM_GF2m_5x5_mul(t1, b, c);
/* Add */
XMM_GF2m_add_5term(t0, t0, t1);
/* Reduce */
XMM_GF2m_mod_sect283(z, t0);
}

inline void XMM_GF2m_addsqrmul_sect409(__m128i z[4], __m128i a[4], __m128i b[4], __m128i c[4])
{
/* Init */
__m128i t0[7], t1[7];
/* Square */
XMM_GF2m_sqr_7term(t0, a);
/* Multiply */
XMM_GF2m_7x7_mul(t1, b, c);
/* Add */
XMM_GF2m_add_7term(t0, t0, t1);
/* Reduce */
XMM_GF2m_mod_sect409(z, t0);
}

inline void XMM_GF2m_addsqrmul_sect571(__m128i z[5], __m128i a[5], __m128i b[5], __m128i c[5])
{
/* Init */
__m128i t0[9], t1[9];
/* Square */
XMM_GF2m_sqr_9term(t0, a);
/* Multiply */
XMM_GF2m_9x9_mul(t1, b, c);
/* Add */
XMM_GF2m_add_9term(t0, t0, t1);
/* Reduce */
XMM_GF2m_mod_sect571(z, t0);
}

/* Multiply + Multiply */
inline void XMM_GF2m_add2mul_sect163(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2])
```

165
inline void XMM_GF2m_add2mul_sect163(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2])
{
    /* Init */
    __m128i t0[2], t1[2];
    /* Multiply */
    XMM_GF2m_3x3_mul( t0, a, b);
    XMM_GF2m_3x3_mul( t1, c, d);
    /* Add */
    XMM_GF2m_add_3term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect163(z, t0);
}

inline void XMM_GF2m_add2mul_sect193(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2])
{
    /* Init */
    __m128i t0[4], t1[4];
    /* Multiply */
    XMM_GF2m_4x4_mul( t0, a, b);
    XMM_GF2m_4x4_mul( t1, c, d);
    /* Add */
    XMM_GF2m_add_4term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect193(z, t0);
}

inline void XMM_GF2m_add2mul_sect233(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2])
{
    /* Init */
    __m128i t0[4], t1[4];
    /* Multiply */
    XMM_GF2m_4x4_mul( t0, a, b);
    XMM_GF2m_4x4_mul( t1, c, d);
    /* Add */
    XMM_GF2m_add_4term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect233(z, t0);
}

inline void XMM_GF2m_add2mul_sect239(__m128i z[2], __m128i a[2], __m128i b[2], __m128i c[2], __m128i d[2])
{
    /* Init */
    __m128i t0[4], t1[4];
    /* Multiply */
    XMM_GF2m_4x4_mul( t0, a, b);
    XMM_GF2m_4x4_mul( t1, c, d);
    /* Add */
    XMM_GF2m_add_4term(t0, t0, t1);
    /* Reduce */
    XMM_GF2m_mod_sect239(z, t0);
}

inline void XMM_GF2m_add2mul_sect283(__m128i z[3], __m128i a[3], __m128i b[3], __m128i c[3], __m128i d[3])
{
    /* Init */
    __m128i t0[6], t1[6];
/* Multiply */
XMM_GF2m_5x5_mul( t0 , a , b);
/* Multiply */
XMM_GF2m_5x5_mul( t1 , c , d);
/* Add */
XMM_GF2m_add_5term( t0 , t0 , t1);
/* Reduce */
XMM_GF2m_mod_sect283( z , t0);
}

{
/* Init */
__m128i t0[7] , t1[7];
/* Multiply */
XMM_GF2m_7x7_mul( t0 , a , b);
/* Multiply */
XMM_GF2m_7x7_mul( t1 , c , d);
/* Add */
XMM_GF2m_add_7term( t0 , t0 , t1);
/* Reduce */
XMM_GF2m_mod_sect409( z , t0);
}

{
/* Init */
__m128i t0[9] , t1[9];
/* Multiply */
XMM_GF2m_9x9_mul( t0 , a , b);
/* Multiply */
XMM_GF2m_9x9_mul( t1 , c , d);
/* Add */
XMM_GF2m_add_9term( t0 , t0 , t1);
/* Reduce */
XMM_GF2m_mod_sect571( z , t0);
}

/*
 ***********************************************
 * XMM INVERSION
 ***********************************************
*/

/* Calculates z = a for any a GF(2^163) with exponent decomposition:
  * (1 + 2)(1 + 2^2)((1 + 2^2)(1 + 2^4)(1 + 2^8)(1 + 2^16)(1 + 2^32)(1 + 2^64))
 */
inline void XMM_GF2m_mod_inv_sect163( __m128i z[2] , __m128i a[2] )
{
  int i;
  __m128i t0[2] , t1[2] , t2[2];
  /* Exponent chain */
  const int chain[] = { 1 , 2 , 4 , 8 , 16 , 32 , 64 , 32 , 2 };
  /* Initial Square z = a */
XMM_GF2m_sqr_sect163( z , a);
  /* Square t0 = z^2^1 */
XMM_GF2m_sqr_sect163( t0 , z);
  /* Multiply component z = z*t0 z = z*t0 */
}
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Save component t1 = z */
XMM_GF2m_copy_2term(t1, z);

*/ Square t0 = z^2 */
XMM_GF2m_sqr_sec163( t0, z);
XMM_GF2m_sqr_sec163( t0, t0);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Square t0 = z^4 */
XMM_GF2m_sqr_sec163( t0, z);
SQRLOOP( XMM_GF2m_sqr_sec163( t0, t0), chain[2]-1);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Square t0 = z^8 */
XMM_GF2m_sqr_sec163( t0, z);
SQRLOOP( XMM_GF2m_sqr_sec163( t0, t0), chain[3]-1);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Square t0 = z^16 */
XMM_GF2m_sqr_sec163( t0, z);
SQRLOOP( XMM_GF2m_sqr_sec163( t0, t0), chain[4]-1);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Save component t2 = z */
XMM_GF2m_copy_2term(t2, z);

*/ Square t0 = z^32 */
XMM_GF2m_sqr_sec163( t0, z);
SQRLOOP( XMM_GF2m_sqr_sec163( t0, t0), chain[5]-1);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Square t0 = z^64 */
XMM_GF2m_sqr_sec163( t0, z);
SQRLOOP( XMM_GF2m_sqr_sec163( t0, t0), chain[6]-1);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, z, t0);

*/ Square t0 = z^128 */
XMM_GF2m_sqr_sec163( t0, z);
SQRLOOP( XMM_GF2m_sqr_sec163( t0, t0), chain[7]-1);

*/ Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec163( z, t2, t0);

*/ Multiply component z = t1*t0 */
XMM_GF2m_mod_mul_sec163( z, t1, t0);

}
/* Square t0 = z^2 *1 */
XMM_GF2m_sqr_sect193( t0, z);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *2 */
XMM_GF2m_sqr_sect193( t0, z);
XMM_GF2m_sqr_sect193( t0, t0);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *4 */
XMM_GF2m_sqr_sect193( t0, z);
XMM_GF2m_sqr_sect193( t0, t0);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *8 */
XMM_GF2m_sqr_sect193( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect193( t0, t0), chain[2]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *16 */
XMM_GF2m_sqr_sect193( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect193( t0, t0), chain[3]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *32 */
XMM_GF2m_sqr_sect193( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect193( t0, t0), chain[4]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Save component t1 = z */
XMM_GF2m_copy_2term( t1, z);
/* Square t0 = z^2 *64 */
XMM_GF2m_sqr_sect193( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect193( t0, t0), chain[5]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *64 */
XMM_GF2m_sqr_sect193( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect193( t0, t0), chain[6]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sect193( z, z, t0);
/* Square t0 = z^2 *64 */
XMM_GF2m_sqr_sect193( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect193( t0, t0), chain[7]-1);
/* Multiply component z = t1*t0 */
XMM_GF2m_mod_mul_sect193( z, t1, t0);
}
/* Multiply component z = z* t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);

/* Square t0 = z^2 */
XMM_GF2m_sqr_sect233(t0, z);
XMM_GF2m_sqr_sect233(t0, t0);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);

/* Square t0 = z^4 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[2]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);
/* Save component t1 = z */
XMM_GF2m_copy_2term(t1, z);
/* Square t0 = z^8 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[3]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);
/* Save component t2 = z */
XMM_GF2m_copy_2term(t2, z);
/* Square t0 = z^16 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[4]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);
/* Save component t3 = z */
XMM_GF2m_copy_2term(t3, z);
/* Square t0 = z^32 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[5]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);
/* Save component t4 = z */
XMM_GF2m_copy_2term(t4, z);
/* Square t0 = z^64 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[6]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t233(z, z, t0);
/* Square t0 = z^128 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[7]-1);
/* Multiply component z = t3*t0 */
XMM_GF2m_mod_mul_sec t233(z, t3, t0);
/* Square t0 = z^256 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[8]-1);
/* Multiply component z = t2*t0 */
XMM_GF2m_mod_mul_sec t233(z, t2, t0);
/* Square t0 = z^512 */
XMM_GF2m_sqr_sect233(t0, z);
SQRLOOP(XMM_GF2m_sqr_sect233(t0, t0), chain[9]-1);
/* Multiply component z = t1*t0 */
XMM_GF2m_mod_mul_sec t233(z, t1, t0);
// Calculates z = a for any a GF(2^239) with exponent decomposition:
// (1 + 2) (1 + 2^2) (1 + 2^4) (1 + 2^4) (1 + 2^8) (1 + 2^8) (1 + 2^16) (1 + 2^32) (1 + 2^32) (1 + 2^64) ((1 + 2^64)))

inline void XMM_GF2m_mod_inv_sect239( __m128i z[2], __m128i a[2])
{
    int i;
    __m128i t0[2], t1[2], t2[2], t3[2], t4[2], t5[2];
    
    // Exponent chain */
    const int chain[] = { 1, 2, 4, 8, 16, 32, 64, 64, 32, 8, 4, 2 };;
    
    // Initial Square z = a */
    XMM_GF2m_sqr_sect239( z, a);
    
    // Square t0 = z^2 */
    XMM_GF2m_sqr_sect239( t0, z);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Save component t1 = z */
    XMM_GF2m_copy_2term( t1, z);
    
    // Square t0 = z^2 */
    XMM_GF2m_sqr_sect239( t0, z);
    XMM_GF2m_sqr_sect239( t0, t0);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Save component t2 = z */
    XMM_GF2m_copy_2term( t2, z);
    
    // Square t0 = z^4 */
    XMM_GF2m_sqr_sect239( t0, z);
    SQRLOOP( XMM_GF2m_sqr_sect239( t0, t0), chain[2]-1);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Save component t3 = z */
    XMM_GF2m_copy_2term( t3, z);
    
    // Square t0 = z^8 */
    XMM_GF2m_sqr_sect239( t0, z);
    SQRLOOP( XMM_GF2m_sqr_sect239( t0, t0), chain[3]-1);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Save component t4 = z */
    XMM_GF2m_copy_2term( t4, z);
    
    // Square t0 = z^16 */
    XMM_GF2m_sqr_sect239( t0, z);
    SQRLOOP( XMM_GF2m_sqr_sect239( t0, t0), chain[4]-1);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Save component t5 = z */
    XMM_GF2m_copy_2term( t5, z);
    
    // Square t0 = z^32 */
    XMM_GF2m_sqr_sect239( t0, z);
    SQRLOOP( XMM_GF2m_sqr_sect239( t0, t0), chain[5]-1);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Save component t6 = z */
    XMM_GF2m_copy_2term( t6, z);
    
    // Square t0 = z^64 */
    XMM_GF2m_sqr_sect239( t0, z);
    SQRLOOP( XMM_GF2m_sqr_sect239( t0, t0), chain[6]-1);
    
    // Multiply component z = z*t0 */
    XMM_GF2m_mod_mul_sect239( z, z, t0);
    
    // Square t0 = z^128 */
}
inline void XMM_GF2m_mod_inv_sect283(__m128i z[3], __m128i a[3])
{
    int i;
    __m128i t0[3], t1[3], t2[3], t3[3];
    /* Exponent chain */
    const int chain[] = { 1, 2, 4, 8, 16, 32, 64, 128, 16, 8, 2};
    /* Initial Square z = a */
    XMM_GF2m_sqr_sect283(z, a);
    /* Square t0 = z^2 */
    XMM_GF2m_sqr_sect283(t0, z);
    /* Multiply component z = z * t0 */
    XMM_GF2m_mod_mul_sect283(z, z, t0);
    /* Save component t1 = z */
    XMM_GF2m_copy_3term(t1, z);
    /* Square t0 = z^2 */
    XMM_GF2m_sqr_sect283(t0, z);
    XMM_GF2m_sqr_sect283(t0, t0);
    /* Multiply component z = z * t0 */
    XMM_GF2m_mod_mul_sect283(z, z, t0);
    /* Save component t2 = z */
    XMM_GF2m_copy_3term(t2, z);
    /* Square t0 = z^2 */
    XMM_GF2m_sqr_sect283(t0, z);
    XMM_GF2m_sqr_sect283(t0, t0);
    /* Multiply component z = z * t0 */
    XMM_GF2m_mod_mul_sect283(z, z, t0);
    /* Save component t1 = z */
    XMM_GF2m_copy_3term(t1, z);
    /* Square t0 = z^2 */
    XMM_GF2m_sqr_sect283(t0, z);
    XMM_GF2m_sqr_sect283(t0, t0);
    /* Multiply component z = z * t0 */
    XMM_GF2m_mod_mul_sect283(z, z, t0);
    /* Save component t2 = z */
    XMM_GF2m_copy_3term(t2, z);
    /* Square t0 = z^2 */
    XMM_GF2m_sqr_sect283(t0, z);
    XMM_GF2m_sqr_sect283(t0, t0);
    /* Multiply component z = z * t0 */
    XMM_GF2m_mod_mul_sect283(z, z, t0);
    /* Save component t1 = z */
    XMM_GF2m_copy_3term(t1, z);
    /* Square t0 = z^2 */
    XMM_GF2m_sqr_sect283(t0, z);
    XMM_GF2m_sqr_sect283(t0, t0);
    /* Multiply component z = z * t0 */
    XMM_GF2m_mod_mul_sect283(z, z, t0);
    /* Save component t2 = z */
    XMM_GF2m_copy_3term(t2, z);
}

}
/* Square t0 = z^2^8 */
XMM_GF2m_sqr_sec t283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[3]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t283( z, z, t0);
/* Save component t3 = z */
XMM_GF2m_copy_3term( t3, z);
/* Square t0 = z^2^16 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[4]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t283( z, z, t0);
/* Save component t3 = z */
XMM_GF2m_copy_3term( t3, z);
/* Square t0 = z^2^32 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[5]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t283( z, z, t0);
/* Save component t3 = z */
XMM_GF2m_copy_3term( t3, z);
/* Square t0 = z^2^64 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[6]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t283( z, z, t0);
/* Save component t3 = z */
XMM_GF2m_copy_3term( t3, z);
/* Square t0 = z^2^128 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[7]-1);
/* Multiply component z = z*t0 */
XMM_GF2m_mod_mul_sec t283( z, z, t0);
/* Save component t3 = z */
XMM_GF2m_copy_3term( t3, z);
/* Square t0 = z^2^16 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[8]-1);
/* Multiply component z = t3*t0 */
XMM_GF2m_mod_mul_sec t283( z, t3, t0);
/* Square t0 = z^2^32 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[9]-1);
/* Multiply component z = t3*t0 */
XMM_GF2m_mod_mul_sec t283( z, t3, t0);
/* Square t0 = z^2^64 */
XMM_GF2m_sqr_sect283( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect283( t0, t0), chain[10]-1);
/* Multiply component z = t3*t0 */
XMM_GF2m_mod_mul_sec t283( z, t3, t0);
/* Square t0 = z^2^128 */
XMM_GF2m_sqr_sect283( t0, z);
/* Multiply component z = t1*t0 */
XMM_GF2m_mod_mul_sec t283( z, t1, t0);
}

/* Calculates z = a for any a GF(2^409) with exponent decomposition:
* (1 + 2)(1 + 2^2)(1 + 2^4)(1 + 2^8)(1 + 2^16)(1 + 2^32)(1 + 2^64)
* (1 + 2^128 (1 + 2^128 )))
*/
inline void XMM_GF2m_mod_inv_sect409( __m128i z[4], __m128i a[4])
{
  int i;
  __m128i t0[4], t1[4], t2[4], t3[4];
  /* Exponent chain */
  const int chain[] = { 1, 2, 4, 8, 16, 32, 64, 128, 128, 16, 8 };
  /* Initial Square z = a */
  XMM_GF2m_sqr_sect409( z, a);
/* Square $t_0 = z^{2^1}$ */
XMM_GF2m_sqr_sect409( t0, z);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Square $t_0 = z^{2^2}$ */
XMM_GF2m_sqr_sect409( t0, z);
XMM_GF2m_sqr_sect409( t0, t0);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Square $t_0 = z^{2^4}$ */
XMM_GF2m_sqr_sect409( t0, z);

SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[2] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Save component $t_1 = z$ */
XMM_GF2m_copy_4term( t1, z);

/* Square $t_0 = z^{2^8}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[3] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Save component $t_2 = z$ */
XMM_GF2m_copy_4term( t2, z);

/* Square $t_0 = z^{2^{16}}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[4] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Square $t_0 = z^{2^{32}}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[5] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Save component $t_1 = z$ */
XMM_GF2m_copy_4term( t1, z);

/* Square $t_0 = z^{2^{64}}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[6] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Save component $t_1 = z$ */
XMM_GF2m_copy_4term( t1, z);

/* Square $t_0 = z^{2^{128}}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[7] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Square $t_0 = z^{2^{256}}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[8] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Square $t_0 = z^{2^{512}}$ */
XMM_GF2m_sqr_sect409( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect409( t0, t0), chain[9] -1);

/* Multiply component $z = z \times t_0$ */
XMM_GF2m_mod_mul_sect409( z, z, t0);

/* Save component $t_1 = z$ */
XMM_GF2m_copy_4term( t1, z);
inline void XMM_GF2m_mod_inv_sect571(__m128i z[5], __m128i a[5])
{
    int i;
    __m128i t0[5], t1[5], t2[5], t3[5], t4[5];
    /* Exponent chain */
    const int chain[] = {1, 2, 4, 8, 16, 32, 64, 128, 256, 16, 8, 2};

    // Initial Square z = a
    XMM_GF2m_sqr_sect571(z, a);
    // Square t0 = z^2^1
    XMM_GF2m_sqr_sect571(t0, z);
    XMM_GF2m_sqr_sect571(t0, t0);
    // Multiply component z = z*t0
    XMM_GF2m_mod_mul_sect571(z, z, t0);
    // Save component t1 = z
    XMM_GF2m_copy_term(t1, z);
    // Square t0 = z^2^2
    XMM_GF2m_sqr_sect571(t0, z);
    SQRLOOP(XMM_GF2m_sqr_sect571(t0, t0), chain[2]-1);
    // Multiply component z = z*t0
    XMM_GF2m_mod_mul_sect571(z, z, t0);
    // Save component t2 = z
    XMM_GF2m_copy_term(t2, z);
    // Square t0 = z^2^4
    XMM_GF2m_sqr_sect571(t0, z);
    SQRLOOP(XMM_GF2m_sqr_sect571(t0, t0), chain[3]-1);
    // Multiply component z = z*t0
    XMM_GF2m_mod_mul_sect571(z, z, t0);
    // Save component t3 = z
    XMM_GF2m_copy_term(t3, z);
    // Square t0 = z^2^8
    XMM_GF2m_sqr_sect571(t0, z);
    SQRLOOP(XMM_GF2m_sqr_sect571(t0, t0), chain[4]-1);
    // Multiply component z = z*t0
    XMM_GF2m_mod_mul_sect571(z, z, t0);
    // Save component t4 = z
    XMM_GF2m_copy_term(t4, z);
    // Square t0 = z^2^16
    XMM_GF2m_sqr_sect571(t0, z);
    SQRLOOP(XMM_GF2m_sqr_sect571(t0, t0), chain[5]-1);
    // Multiply component z = z*t0
    XMM_GF2m_mod_mul_sect571(z, z, t0);
    // Save component t5 = z
    XMM_GF2m_copy_term(t5, z);
    // Square t0 = z^2^32
    XMM_GF2m_sqr_sect571(t0, z);
    SQRLOOP(XMM_GF2m_sqr_sect571(t0, t0), chain[6]-1);
}
/* Multiply component z = z * t0 */
XMM_GF2m_mod_mul_sec t 571( z, z, t0);
/* Square t0 = z^2^128 */
XMM_GF2m_sqr_sec571( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect571( t0, t0), chain[7] -1);
/* Multiply component z = z * t0 */
XMM_GF2m_mod_mul_sec t 571( z, z, t0);
/* Square t0 = z^2^256 */
XMM_GF2m_sqr_sec571( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect571( t0, t0), chain[8] -1);
/* Multiply component z = z * t0 */
XMM_GF2m_mod_mul_sec t 571( z, z, t0);
/* Square t0 = z^2^32 */
XMM_GF2m_sqr_sec571( t0, z);
SQRLOOP( XMM_GF2m_sqr_sect571( t0, t0), chain[9] -1);
/* Multiply component z = t4 * t0 */
XMM_GF2m_mod_mul_sec t 571( z, t4 , t0);
/* Square t0 = z^2^16 */
XMM_GF2m_sqr_sec t 571( t0 , z);
SQRLOOP ( XMM_GF2m_sqr_sect571 ( t0 , t0), chain[10] -1) ;
/* Multiply component z = t3 * t0 */
XMM_GF2m_mod_mul_sec t 571( z, t3 , t0);
/* Square t0 = z^2^8 */
XMM_GF2m_sqr_sec t 571( t0 , z);
SQRLOOP ( XMM_GF2m_sqr_sect571 ( t0 , t0), chain[11] -1) ;
/* Multiply component z = t2 * t0 */
XMM_GF2m_mod_mul_sec t 571( z, t2 , t0);
/* Square t0 = z^2^2 */
XMM_GF2m_sqr_sec t 571( t0 , z);
XMM_GF2m_sqr_sec t 571( t0 , t0);
/* Multiply component z = t1 * t0 */
XMM_GF2m_mod_mul_sec t 571( z, t1 , t0);
}

/* Calculates z = a/b = a * b for all a, b, z GF (2^163) */
inline void XMM_GF2m_div_sect163 ( __m128i z[2] , __m128i a[2] , __m128i b[2])
{
/* Init */
__m128i c[2];
/* Invert b */
XMM_GF2m_mod_inv_sect163(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mul_sect163(z, a, c);
}

/* Calculates z = a/b = a * b for all a, b, z GF (2^193) */
inline void XMM_GF2m_div_sect193 ( __m128i z[2] , __m128i a[2] , __m128i b[2])
{
/* Init */
__m128i c[2];
/* Invert b */
XMM_GF2m_mod_inv_sect193(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mul_sect193(z, a, c);
}

/* Calculates z = a/b = a * b for all a, b, z GF (2^233) */
inline void XMM_GF2m_div_sect233 ( __m128i z[2] , __m128i a[2] , __m128i b[2])
{
/* Init */
__m128i c[2];
/* Invert b */
XMM_GF2m_mod_inv_sect233(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mul_sect233(z, a, c);
}
#include stdlib.h

__m128i c[2];

/* Invert b */
XMM_GF2m_mod_inv_section233(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mull_section233(z, a, c);

/* Calculates z = a/b = a * b for all a, b, z GF (2^239) */
inline void XMM_GF2m_div_section239(__m128i z[2], __m128i a[2], __m128i b[2])
{
/* Init */
__m128i c[2];
/* Invert b */
XMM_GF2m_mod_inv_section233(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mull_section233(z, a, c);
}

/* Calculates z = a/b = a * b for all a, b, z GF (2^283) */
inline void XMM_GF2m_div_section283(__m128i z[3], __m128i a[3], __m128i b[3])
{
/* Init */
__m128i c[3];
/* Invert b */
XMM_GF2m_mod_inv_section283(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mull_section283(z, a, c);
}

/* Calculates z = a/b = a * b for all a, b, z GF (2^409) */
inline void XMM_GF2m_div_section409(__m128i z[4], __m128i a[4], __m128i b[4])
{
/* Init */
__m128i c[4];
/* Invert b */
XMM_GF2m_mod_inv_section409(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mull_section409(z, a, c);
}

/* Calculates z = a/b = a * b for all a, b, z GF (2^571) */
inline void XMM_GF2m_div_section571(__m128i z[5], __m128i a[5], __m128i b[5])
{
/* Init */
__m128i c[5];
/* Invert b */
XMM_GF2m_mod_inv_section571(c, b);
/* Multiply a * b */
XMM_GF2m_mod_mull_section571(z, a, c);
}

/* BIGNUM WRAPPER FUNCTIONS */
#endif

int BN_GF2m_mod_xmm_section163(BIGNUM *z, const BIGNUM *a)
{
/* Init */
int ret=0;
__m128i _t[2], _a[3];

177
/* Load */
if (! BN_to_XMM(_a, a, 6)) goto err;

/* Reduce */
XMM_GF2m_mod_sect163(_t, _a);

/* Store */
if (! XMM_to_BN(z, _t, 3)) goto err;
ret = 1;

err:
    return ret;
}

int BN_GF2m_mod_xmm_pclmul_sect163(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[3];

    /* Load */
    if (! BN_to_XMM(_a, a, 6)) goto err;

    /* Reduce */
    XMM_GF2m_mod_sect163_pclmul(_t, _a);

    /* Store */
    if (! XMM_to_BN(z, _t, 3)) goto err;
    ret = 1;

err:
    return ret;
}

int BN_GF2m_mod_xmm_sect193(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[4];

    /* Load */
    if (! BN_to_XMM(_a, a, 7)) goto err;

    /* Reduce */
    XMM_GF2m_mod_sect193(_t, _a);

    /* Store */
    if (! XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;

err:
    return ret;
}

int BN_GF2m_mod_xmm_sect233(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[4];

    /* Load */
    if (! BN_to_XMM(_a, a, 8)) goto err;

    /* Reduce */
    XMM_GF2m_mod_sect233(_t, _a);

    /* Store */
    if (! XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;

err:
    return ret;
}

int BN_GF2m_mod_xmm_sect239(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;

    /* Load */
    if (! BN_to_XMM(_a, a, 8)) goto err;

    /* Reduce */
    XMM_GF2m_mod_sect239(_t, _a);

    /* Store */
    if (! XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;

err:
    return ret;
}
int BN_GF2m_mod_xmm_sect239(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[4];
    /* Load */
    if (!BN_to_XMM(_a, a, 8)) goto err;
    /* Reduce */
    XMM_GF2m_mod_sect239(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;
}

int BN_GF2m_mod_xmm_sect283(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[3], _a[5];
    /* Load */
    if (!BN_to_XMM(_a, a, 9)) goto err;
    /* Reduce */
    XMM_GF2m_mod_sect283(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 5)) goto err;
    ret = 1;
}

int BN_GF2m_mod_xmm_pclmul_sect283(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[3], _a[5];
    /* Load */
    if (!BN_to_XMM(_a, a, 9)) goto err;
    /* Reduce */
    XMM_GF2m_mod_sect283_pclmul(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 5)) goto err;
    ret = 1;
}

int BN_GF2m_mod_xmm_sect409(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[4], _a[7];
    /* Load */
    if (!BN_to_XMM_conv(_a, a, 13)) goto err;
    /* Reduce */
    XMM_GF2m_mod_sect409(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 7)) goto err;
    ret = 1;
}

int BN_GF2m_mod_xmm_sect571(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
int ret = 0;
    __m128i _t[5], _a[10];

    /* Load */
    if (!BN_to_XMM_conv(_a, a, 18)) goto err;
    /* Reduce */
    XMM_GF2m_mod_sect571(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 9)) goto err;
    ret = 1;
err:
    return ret;
}

int BN_GF2m_mod_xmm_pclmul_sect571(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[5], _a[10];
    /* Load */
    if (!BN_to_XMM_conv(_a, a, 18)) goto err;
    /* Reduce */
    XMM_GF2m_mod_sect571_pclmul(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 9)) goto err;
    ret = 1;
err:
    return ret;
}

int BN_GF2m_sqr_xmm_sect163(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 3)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect163(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 3)) goto err;
    ret = 1;
err:
    return ret;
}

int BN_GF2m_sqr_xmm_sect193(BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 4)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect193(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;
err:
    return ret;
}
int BN_GF2m_sqr_xmm_sect233 (BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 4)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect233(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;
err:
    return ret;
}

int BN_GF2m_sqr_xmm_sect239 (BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 4)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect239(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;
err:
    return ret;
}

int BN_GF2m_sqr_xmm_sect283 (BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[3], _a[3];
    /* Load */
    if (!BN_to_XMM(_a, a, 5)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect283(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 5)) goto err;
    ret = 1;
err:
    return ret;
}

int BN_GF2m_sqr_xmm_sect409 (BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[4], _a[4];
    /* Load */
    if (!BN_to_XMM(_a, a, 7)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect409(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 7)) goto err;
    ret = 1;
err:
    return ret;
}
```c
int BN_GF2m_sqr_xmm_sect571 (BIGNUM *z, const BIGNUM *a)
{
    /* Init */
    int ret = 0;
    __m128i _t[5], _a[5];
    /* Load */
    if (!BN_to_XMM(_a, a, 9)) goto err;
    /* Square */
    XMM_GF2m_sqr_sect571(_t, _a);
    /* Store */
    if (!XMM_to_BN(z, _t, 9)) goto err;
    ret = 1;
    err:
    return ret;
}

int BN_GF2m_mul_xmm_sect163 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 3)) goto err;
    if (!BN_to_XMM(_b, b, 3)) goto err;
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul_sect163(_t, _a, _b);
    /* Store */
    if (!XMM_to_BN(z, _t, 3)) goto err;
    ret = 1;
    err:
    return ret;
}

int BN_GF2m_mul_xmm_sect193 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 4)) goto err;
    if (!BN_to_XMM(_b, b, 4)) goto err;
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul_sect193(_t, _a, _b);
    /* Store */
    if (!XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;
    err:
    return ret;
}

int BN_GF2m_mul_xmm_sect233 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 4)) goto err;
    if (!BN_to_XMM(_b, b, 4)) goto err;
    /* Multiply & Reduce */
```
XMM_GF2m_mod_mul_sect233(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 4)) goto err;
ret = 1;
}
error:
return ret;
}

int BN_GF2m_mul xmm sect239(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
/* Init */
int ret = 0;
__m128i _t[2], _a[2], _b[2];
/* Load */
if (!BN_to_XMM(_a, a, 4)) goto err;
if (!BN_to_XMM(_b, b, 4)) goto err;
/* Multiply & Reduce*/
XMM_GF2m_mod_mul_sect233(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 4)) goto err;
ret = 1;
}
error:
return ret;
}

int BN_GF2m_mul_xmm sect283(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
/* Init */
int ret = 0;
__m128i _t[3], _a[3], _b[3];
/* Load */
if (!BN_to_XMM(_a, a, 5)) goto err;
if (!BN_to_XMM(_b, b, 5)) goto err;
/* Multiply & Reduce*/
XMM_GF2m_mod_mul_sect283(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 5)) goto err;
ret = 1;
}
error:
return ret;
}

int BN_GF2m_mul_xmm sect409(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
/* Init */
int ret = 0;
__m128i _t[4], _a[4], _b[4];
/* Load */
if (!BN_to_XMM(_a, a, 7)) goto err;
if (!BN_to_XMM(_b, b, 7)) goto err;
/* Multiply & Reduce*/
XMM_GF2m_mod_mul_sect409(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 7)) goto err;
ret = 1;
}
error:
return ret;
}
int BN_GF2m_mul_xmm_sect571 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
    /* Init */
    int ret = 0;
    __m128i _t[5], _a[5], _b[5];
    /* Load */
    if (!BN_to_XMM(_t, a, 9)) goto err;
    if (!BN_to_XMM(_a, b, 9)) goto err;
    /* Multiply & Reduce */
    XMM_GF2m_mod_mul sect571(_t, _a, _b);
    /* Store */
    if (!XMM_to_BN(z, _t, 9)) goto err;
    ret = 1;
err:
    return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect163 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 3)) goto err;
    if (!BN_to_XMM(_b, b, 3)) goto err;
    /* Divide */
    XMM_GF2m_div sect163(_t, _a, _b);
    /* Store */
    if (!XMM_to_BN(z, _t, 3)) goto err;
    ret = 1;
err:
    return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect193 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
    if (!BN_to_XMM(_a, a, 4)) goto err;
    if (!BN_to_XMM(_b, b, 4)) goto err;
    /* Divide */
    XMM_GF2m_div sect193(_t, _a, _b);
    /* Store */
    if (!XMM_to_BN(z, _t, 4)) goto err;
    ret = 1;
err:
    return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect233 (BIGNUM *z, const BIGNUM *a, const BIGNUM *b) {
    /* Init */
    int ret = 0;
    __m128i _t[2], _a[2], _b[2];
    /* Load */
if (!BN_to_XMM(_a, a, 4)) goto err;
if (!BN_to_XMM(_b, b, 4)) goto err;

/* Divide */
XMM_GF2m_div_sect233(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 4)) goto err;
ret = 1;

err:
return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect239(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
/* Init */
int ret = 0;
__m128i _t[2], _a[2], _b[2];
/* Load */
if (!BN_to_XMM(_a, a, 4)) goto err;
if (!BN_to_XMM(_b, b, 4)) goto err;
/* Divide */
XMM_GF2m_div_sect239(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 4)) goto err;
ret = 1;
err:
return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect283(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
/* Init */
int ret = 0;
__m128i _t[3], _a[3], _b[3];
/* Load */
if (!BN_to_XMM(_a, a, 5)) goto err;
if (!BN_to_XMM(_b, b, 5)) goto err;
/* Divide */
XMM_GF2m_div_sect283(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 5)) goto err;
ret = 1;
err:
return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect409(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
/* Init */
int ret = 0;
__m128i _t[4], _a[4], _b[4];
/* Load */
if (!BN_to_XMM(_a, a, 7)) goto err;
if (!BN_to_XMM(_b, b, 7)) goto err;
/* Divide */
XMM_GF2m_div_sect409(_t, _a, _b);
/* Store */
if (!XMM_to_BN(z, _t, 7)) goto err;
ret = 1;
err:
  return ret;
}

/* Wrapper for XMM Division */
int BN_GF2m_div_xmm_sect571(BIGNUM *z, const BIGNUM *a, const BIGNUM *b)
{
  /* Init */
  int ret = 0;
  __m128i _t[5], _a[5], _b[5];
  /* Load */
  if (!BN_to_XMM(_a, a, 9)) goto err;
  if (!BN_to_XMM(_b, b, 9)) goto err;
  /* Divide */
  XMM_GF2m_div_sect571(_t, _a, _b);
  /* Store */
  if (!XMM_to_BN(z, _t, 9)) goto err;
  ret = 1;
}

err:
  return ret;
}

/* BIGNUM MADDLE FUNCTIONS */
int BN_GF2m_Maddle_xmm_nist163k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
  /* Init */
  int ret = 0;
  __m128i _t1[2], _t2[3], _t3[3], _x1[2], _z1[2], _x2[2], _z2[2];
  __m128i _tx1[2], _tx2[2], _tz1[2], _tz2[2];
  /* Load */
  BN_to_XMM_3term(_tx1, x1->d);
  BN_to_XMM_3term(_tz1, z1->d);
  BN_to_XMM_3term(_tz2, z2->d);
  BN_to_XMM_3term(_tx2, x2->d);
  /* Data veiling */
  XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
  /* MADD */
  XMM_GF2m_mod_mul_nist163(_x1, _x1, _z2);
  XMM_GF2m_mod_mul_nist163(_z1, _z1, _x2);
  /* Multiply w/o reduction */
  XMM_GF2m_3x3_mul(_t2, _tx1, _x1);
  XMM_GF2m_add_2term(_z1, _z1, _x1);
  XMM_GF2m_mod_sqr_nist163(_z1, _z1);
  /* Multiply w/o reduction */
  XMM_GF2m_3x3_mul(_t3, _z1, _t1);
int BN_GF2m_Maddle_xmm_nist163r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c) {
    /* Init */
    int ret = 0;
    __m128i _t1[2], _t2[3], _t3[3], _x1[2], _z1[2], _x2[2], _z2[2];
    __m128i _tx1[2], _tz1[2], _tx2[2], _tz2[2];
    /* Load */
    BN_to_XMM_3term(_tx1, x1->d);
    BN_to_XMM_3term(_tz1, z1->d);
    BN_to_XMM_3term(_tz2, z2->d);
    BN_to_XMM_3term(_tx2, x2->d);
    /* Data veiling */
    XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);
    /* MADD */
    XMM_GF2m_mod_mul_nist163(_x1, _x1, _z2);
    XMM_GF2m_mod_mul_nist163(_z1, _z1, _x2);
    /* Multiply w/o reduction */
    XMM_GF2m_3x3_mul(_t2, _x1, _z1);
    XMM_GF2m_add_2term(_x1, _z1, _x1);
    XMM_GF2m_mod_sqr_nist163(_x1, _x1);
    /* Multiply w/o reduction */
    BN_to_XMM_3term(_t1, x->d);
    XMM_GF2m_3x3_mul(_t3, _z1, _t1);
    /* Add the two double-sized numbers and reduce */
    XMM_GF2m_add_3term(_t3, _t3, _t2);
    XMM_GF2m_mod_nist163(_x1, _t3);
    /* MDOUBLE */
    XMM_GF2m_mod_sqr_nist163(_x2, _x2);
    XMM_GF2m_mod_sqr_nist163(_z2, _z2);
    BN_to_XMM_3term(_t1, c->d);
    XMM_GF2m_mod_nist163(_t1, _z2, _t1);
    XMM_GF2m_mod_mul_nist163(_t1, _z2, _t1);
    XMM_GF2m_add_2term(_x2, _x2, _t1);
    XMM_GF2m_mod_sqr_nist163(_x2, _x2);
    /* Unveil data */
    XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);
    /* Store results */
    XMM_to_BN_3term(x1->d, _tx1);
    XMM_to_BN_3term(x2->d, _tx2);
    ret = 1;
    return ret;
}
int BN_GF2m_Maddle_xmm_sect193r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c)
{
    /* Init */
    int ret = 0;
    __m128i _t1[2], _t2[4], _t3[4], _x1[2], _z1[2], _x2[2], _z2[2];
    __m128i _tx1[2], _tz1[2], _tx2[2], _tz2[2];

    /* Load */
    BN_to_XMM_4term(_tx1, x1->d);
    BN_to_XMM_4term(_tz1, z1->d);
    BN_to_XMM_4term(_tx2, x2->d);

    /* Data veiling */
    XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

    /* MADD */
    XMM_GF2m_mod_mul_sect193(_x1, _x1, _z2);
    XMM_GF2m_mod_mul_sect193(_z1, _z1, _x2);

    /* Multiply w/o reduction */
    XMM_GF2m_4x4_mul(_t2, _x1, _z1);
    XMM_GF2m_add_2term(_z1, _z1, _x1);
    XMM_GF2m_mod_sqr_sect193(_z1, _z1);

    /* Multiply w/o reduction */
    BN_to_XMM_4term(_t1, x->d);
    XMM_GF2m_4x4_mul(_t3, _z1, _t1);

    /* Add the two double-sized numbers and reduce */
    XMM_GF2m_add_4term(_t3, _t3, _t2);
    XMM_GF2m_mod_sect193(_x1, _t3);

    /* MDOUBLE */
    XMM_GF2m_mod_sqr_sect193(_x2, _x2);
    XMM_GF2m_mod_sqr_sect193(_z2, _z2);
    BN_to_XMM_4term(_t1, c->d);
    XMM_GF2m_mod_sect193(_t1, _x2, _t1);

    /* Unveil data */
    XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _x1, _z1, _x2, _z2, k);

    /* Store results */
    XMM_to_BN_4term(_t1->d, _tx1);
    XMM_to_BN_4term(_t2->d, _tx2);
    XMM_to_BN_4term(_t2->d, _tx2);
    ret = 1;
    return ret;
}

int BN_GF2m_Maddle_xmm_nist233k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
    /* Init */
    int ret = 0;
    __m128i _t1[2], _t2[4], _t3[4], _x1[2], _x1[2], _z1[2], _z1[2];
    __m128i _tx1[2], _tx2[2], _tz2[2];

    /* Load */
    BN_to_XMM_4term(_tx1, x1->d);
    BN_to_XMM_4term(_tz1, x1->d);
int BN_GF2m_Maddle_xmm_nist233r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c) {
  /* Init */
  int ret = 0;
  __m128i _t1[2], _t2[4], _x1[2], _z1[2], _x2[2], _z2[2], _tx1[2], _tx2[2], _tz1[2], _tz2[2];
  /* Load */
  BN_to_XMM_4term(_tx1, x1->d);
  BN_to_XMM_4term(_tz1, x2->d);
  BN_to_XMM_4term(_tx2, x2->d);
  /* Data veiling */
  XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
  /* MADD */
  XMM_GF2m_mod_mul_nist233(_x1, _x1, _z2);
  XMM_GF2m_mod_mul_nist233(_z1, _z1, _x2);
  /* Multiply w/o reduction */
  XMM_GF2m_4x4_mul(_t2, _x1, _z1);
  XMM_GF2m_add_2term(_z1, _z1, _x1);
  XMM_GF2m_mod_sqr_nist233(_z1, _z1);
  /* Multiply w/o reduction */
  BN_to_XMM_4term(_t1, x->d);
  XMM_GF2m_4x4_mul(_t3, _z1, _t1);
  /* Add the two double-sized numbers and reduce */
  XMM_GF2m_add_4term(_t3, _t3, _t2);
  XMM_GF2m_mod_nist233(_x1, _t3);
  /* MDDOUBLE */
  XMM_GF2m_mod_sqr_nist233(_x2, _x2);
  XMM_GF2m_mod_sqr_nist233(_z2, _z2);
  XMM_GF2m_mod_sqr_nist233(_x2, _t1);
  /* Unveil data */
  XMM_GF2m_veil_2term(_tx1, _tx1, _tx2, _tz1, _tz1, _tx2, _tz2, _tx1, _tz2, _tx2, _tz2, k);
  /* Store results */
  XMM_to_BN_4term(x1->d, _tx1);
  XMM_to_BN_4term(x2->d, _tx2);
  XMM_to_BN_4term(z2->d, _tz2);
  ret = 1;
  return ret;
}

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_4term(_t3, _t3, _t2);
XMM_GF2m_mod_nist233(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_nist233(_x2, _x2);
XMM_GF2m_mod_sqr_nist233(_z2, _z2);
BN_to_XMM_4term(_t1, c->d);
XMM_GF2m_mod_nist233(_t1, _z2, _t1);
XMM_GF2m_mod_nist233(_s2, _x2, _z2);
XMM_GF2m_add_2term(_s2, _z2, _t1);
XMM_GF2m_mod_sqr_nist233(_x2, _x2);

/* Unveil data */
XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);

/* Store results */
XMM_to_BN_4term(x1->d, _tx1);
XMM_to_BN_4term(x2->d, _tx2);
XMM_to_BN_4term(z1->d, _tz1);
XMM_to_BN_4term(z2->d, _tz2);

ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_sect239k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1, const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k){
/* Init */
int ret = 0;

/* Load */
BN_to_XMM_4term(_tx1, x1->d);
BN_to_XMM_4term(_tx2, x2->d);
BN_to_XMM_4term(_tz1, z1->d);
BN_to_XMM_4term(_tz2, z2->d);

/* Data veiling */
XMM_GF2m_veil_2term(_x1, _z1, _x2, _z2, _tx1, _tx2, _tz1, _tz2, k);

/* MADD */
XMM_GF2m_mod_nist_sect239(_x1, _x1, _x2);
XMM_GF2m_mod_nist_sect239(_z1, _z2, _z1);

/* Multiply w/o reduction */
XMM_GF2m_4x4_mul(_t2, _x1, _z1);
XMM_GF2m_add_2term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_sect239(_z1, _z1);

/* Multiply w/o reduction */
BN_to_XMM_4term(_tx1, x->d);
XMM_GF2m_4x4_mul(_t3, _x1, _t1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_4term(_t3, _t3, _t2);
XMM_GF2m_mod_sect239(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_sect239(_x2, _x2);
XMM_GF2m_mod_sqr_sect239(_z2, _z2);
XMM_GF2m_add_2term(_t1, _z2, _z2);
XMM_GF2m_mod_nist_sect239(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_sect239(_x2, _t1);

/* Unveil data */
XMM_GF2m_veil_2term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);
/* Store results */
XMM_to_BN_4term(x1 ->d, _tx1);
XMM_to_BN_4term(z1 ->d, _tz1);
XMM_to_BN_4term(x2 ->d, _tx2);
XMM_to_BN_4term(z2 ->d, _tz2);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist283k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
                                 const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
    /* Init */
    int ret = 0;
    __m128i _t1[3], _t2[5], _t3[5], _x1[3], _z1[3], _x2[3], _z2[3];
    __m128i _tx1[3], _tz1[3], _tx2[3], _tz2[3];
    /* Load */
    BN_to_XMM_5term(_tx1, x1->d);
    BN_to_XMM_5term(_tz1, z1->d);
    BN_to_XMM_5term(_tx2, x2->d);
    BN_to_XMM_5term(_tz2, z2->d);
    /* Data veiling */
    XMM_GF2m_veil_3term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
    /* MADD */
    XMM_GF2m_mod_mul_nist283(_x1, _x1, _z2);
    XMM_GF2m_mod_mul_nist283(_z1, _z1, _x2);
    /* Multiply w/o reduction */
    XMM_GF2m_5x5_mul(_t2, x1, _x1);
    XMM_GF2m_5x5_mul(_t3, _z1, _z1);
    /* Multiply w/o reduction */
    BN_to_XMM_5term(_t1, x->d);
    XMM_GF2m_5x5_mul(_t3, _z1, _t1);
    /* Add the two double-sized numbers and reduce */
    XMM_GF2m_add_5term(_t3, _t3, _t2);
    XMM_GF2m_mod_nist283(_x1, _t3);
    /* MDoppIE */
    XMM_GF2m_mod_sqr_nist283(_x2, _x2);
    XMM_GF2m_mod_sqr_nist283(_z2, _z2);
    XMM_GF2m_add_3term(_t1, _z2, _z2);
    XMM_GF2m_mod_sqr_nist283(_z2, _z2);
    XMM_GF2m_mod_sqr_nist283(_z2, _t1);
    /* Unveil data */
    XMM_GF2m_veil_3term(_tx1, _tx1, _tx2, _tx2, _x1, _x1, _x2, _x2, _k);
    /* Store results */
    XMM_to_BN_5term(_tx1, _tx1);
    XMM_to_BN_5term(_tx2, _tx2);
    XMM_to_BN_5term(_tz1, _tz1);
    XMM_to_BN_5term(_tz2, _tz2);
    ret = 1;
    return ret;
}

int BN_GF2m_Maddle_xmm_nist283r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
                                 const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c)
{
    /* Init */
    int ret = 0;
    __m128i _t1[3], _t2[5], _t3[5], _x1[3], _z1[3], _x2[3], _z2[3];
    __m128i _tx1[3], _tz1[3], _tx2[3], _tz2[3];
    /* Load */
    BN_to_XMM_5term(_tx1, x1->d);
    BN_to_XMM_5term(_tx1, z1->d);
    BN_to_XMM_5term(_tx2, x2->d);
    BN_to_XMM_5term(_tx2, z2->d);
    BN_to_XMM_5term(_tx1, z1->d);
}
BN_to_XMM_5term(_tx2, _z2 ->d);
BN_to_XMM_5term(_tx2, _x2 ->d);

/* Data veiling */
XMM_GF2m_veil_3term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

/* MADD */
XMM_GF2m_mod_nist283(_x1, _x1, _z2);
XMM_GF2m_mod_nist283(_z1, _z1, _x2);

/* Multiply w/o reduction */
XMM_GF2m_5x5_mul(_t2, _x1, _z1);
XMM_GF2m_add_3term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist283(_z1, _z1);

/* Multiply w/o reduction */
BN_to_XMM_5term(_t1, x->d);
XMM_GF2m_5x5_mul(_t3, _z1, _t1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_5term(_t3, _t3, _t2);
XMM_GF2m_mod_nist283(_x1, _t3);

/* MDOUBLE */
XMM_GF2m_mod_sqr_nist283(_x2, _x2);
XMM_GF2m_mod_sqr_nist283(_z2, _z2);

BN_to_XMM_5term(_t1, c->d);
XMM_GF2m_5x5_mul(_t3, _z1, _t1);

/* Unveil data */
XMM_GF2m_veil_3term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);

/* Store results */
XMM_to_BN_5term(_t1, _tx1);
XMM_to_BN_5term(_t2, _tx2);
XMM_to_BN_5term(_t3, _tx3);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist409k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
/* Init */
int ret = 0;
__m128i _t1[4], _t2[7], _t3[7], _x1[4], _z1[4], _x2[4], _z2[4];
__m128i _tx1[4], _tz1[4], _tx2[4], _tz2[4];

/* Load */
BN_to_XMM_7term(_tx1, x1->d);
BN_to_XMM_7term(_tz1, x1->d);
BN_to_XMM_7term(_tx2, x2->d);
BN_to_XMM_7term(_tz2, x2->d);

/* Data veiling */
XMM_GF2m_veil_4term(_x1, _z1, _x2, _z2, _tx1, _tx2, _tx2, _tz2, k);

/* MADD */
XMM_GF2m_mod_nist409(_x1, _x1, _z2);
XMM_GF2m_mod_nist409(_z1, _z1, _x2);

/* Multiply w/o reduction */
XMM_GF2m_7x7_mul(_t2, _x1, _z1);
XMM_GF2m_add_4term(_z1, _z1, _x1);
XMM_GF2m_mod_sqr_nist409(_z1, _z1);

/* Multiply w/o reduction */
BN_to_XMM_7term(_t1, x->d);
XMM_GF2m_7x7_mul(_t3, _z1, _t1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_7term(_t3, _t3, _z2);
XMM_GF2m_mod_nist409(_x1, _t3);

/* MODouble */
XMM_GF2m_mod_sqr_nist409(_x2, _x2);
XMM_GF2m_mod_sqr_nist409(_z2, _z2);

XMM_GF2m_add_4term(_t1, _z2, _x2);
XMM_GF2m_mod_mul_nist409(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_nist409(_x2, _t1);

/* Unveil data */
XMM_GF2m_veil_4term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);

/* Store results */
XMM_to_BN_7term(_tx1->d, _tx1);
XMM_to_BN_7term(_tz1->d, _tz1);
XMM_to_BN_7term(_tz2->d, _tz2);
XMM_to_BN_7term(_tx2->d, _tx2);
ret = 1;
return ret;

int BN_GF2m_Maddle_xmm_nist409r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c) {
  /* Init */
  int ret = 0;
  __m128i _t1[4], _t2[7], _t3[7], _x1[4], _z1[4], _x2[4], _z2[4];
  __m128i _tx1[4], _tx2[4], _tz1[4], _tz2[4];

  /* Load */
  BN_to_XMM_7term(_tx1, x->d);
  BN_to_XMM_7term(_tz1, x1->d);
  BN_to_XMM_7term(_tx2, x2->d);
  BN_to_XMM_7term(_tz2, z2->d);

  /* Data veiling */
  XMM_GF2m_veil_4term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

  /* ADD */
  XMM_GF2m_mod_mul_nist409(_x1, _x1, _x2);
  XMM_GF2m_mod_mul_nist409(_z1, _z1, _z2);

  /* Multiply w/o reduction */
  XMM_GF2m_7x7_mul(_t2, _x1, _z1);
  XMM_GF2m_add_4term(_x1, _z1, _x1);
  XMM_GF2m_mod_sqr_nist409(_z1, _z1);

  /* Multiply w/o reduction */
  BN_to_XMM_7term(_t1, x->d);
  XMM_GF2m_7x7_mul(_t3, _z1, _t1);

  /* Add the two double-sized numbers and reduce */
  XMM_GF2m_add_7term(_t3, _t3, _t2);
  XMM_GF2m_mod_nist409(_x1, _t3);

  /* MODouble */
  XMM_GF2m_mod_sqr_nist409(_x2, _x2);
  XMM_GF2m_mod_sqr_nist409(_z2, _z2);
  BN_to_XMM_7term(_t1, c->d);
  XMM_GF2m_mod_mul_nist409(_t1, _z2, _t1);
  XMM_GF2m_mod_mul_nist409(_z2, _x2, _x2);
  XMM_GF2m_add_4term(_x2, _z2, _x1);
  XMM_GF2m_mod_sqr_nist409(_x2, _x2);

  /* Unveil data */
  XMM_GF2m_veil_4term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);
/* Store results */
XMM_to_BN_7term(x1 ->d, _tx1);
XMM_to_BN_7term(z1 ->d, _tz1);
XMM_to_BN_7term(x2 ->d, _tx2);
XMM_to_BN_7term(z2 ->d, _tz2);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist571k(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k)
{
/* Init */
int ret = 0;
__m128i _t1[5], _t2[9], _t3[9], _x1[5], _z1[5], _x2[5], _z2[5];
__m128i _tx1[5], _tz1[5], _tx2[5], _tz2[5];
/* Load */
BN_to_XMM_9term(_tx1, x1 ->d);
BN_to_XMM_9term(_tz1, z1 ->d);
BN_to_XMM_9term(_tx2, x2 ->d);
BN_to_XMM_9term(_tz2, z2 ->d);
/* Data veiling */
XMM_GF2m_veil_5term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
/* MADD */
XMM_GF2m_mod_sqr_nist571(_x1, _z1, _x2);
XMM_GF2m_mod_sqr_nist571(_z1, _z1, _z2);
/* Multiply w/o reduction */
XMM_GF2m_9x9_mul(_t2, _x1, _z1);
XMM_GF2m_add_5term(_z1, _z1, _x2);  
XMM_GF2m_mod_sqr_nist571(_z1, _z1);
/* Multiply w/o reduction */
BN_to_XMM_9term(_x1, x1 ->d);
XMM_GF2m_9x9_mul(_t3, _x1, _t1);
/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_9terms(_t2, _t3, _t2);
XMM_GF2m_mod_sqr_nist571(_x1, _t3);
/* MDOUBLE */
XMM_GF2m_mod_sqr_nist571(_x2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _z2);
XMM_GF2m_add_5term(_t1, _x2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _t1);
/* Unveil data */
XMM_GF2m_veil_5term(_tx1, _tx2, _tz1, _tz2, x1, _x1, _z2, _z2, _z1, _z1, _x2, _z2, k);
/* Store results */
XMM_to_BN_9term(x1 ->d, _tx1);
XMM_to_BN_9term(z1 ->d, _tz1);
XMM_to_BN_9term(x2 ->d, _tx2);
XMM_to_BN_9term(z2 ->d, _tz2);
ret = 1;
return ret;
}

int BN_GF2m_Maddle_xmm_nist571r(const BIGNUM *x, BIGNUM *x1, BIGNUM *z1,
const BIGNUM *x2, const BIGNUM *z2, BN_ULONG k, const BIGNUM *c)
{
/* Init */
int ret = 0;
__m128i _t1[5], _t2[9], _t3[9], _x1[5], _z1[5], _x2[5], _z2[5];
__m128i _tx1[5], _tz1[5], _tx2[5], _tz2[5];
/* Load */
BN_to_XMM_9term(_tx1, x1 ->d);
BN_to_XMM_9term(_tz1, z1 ->d);
BN_to_XMM_9term(_tx2, x2 ->d);
BN_to_XMM_9term(_tz2, z2 ->d);
/* Data veiling */
XMM_GF2m_veil_5term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);
/* MADD */
XMM_GF2m_mod_sqr_nist571(_x1, _x1, _x2);
XMM_GF2m_mod_sqr_nist571(_z1, _z1, _z2);
/* Multiply w/o reduction */
XMM_GF2m_9x9_mul(_t2, _x1, _z1);
XMM_GF2m_add_5term(_z1, _z1, _x2);
XMM_GF2m_mod_sqr_nist571(_z1, _z1);
/* Multiply w/o reduction */
BN_to_XMM_9term(_x1, x1 ->d);
XMM_GF2m_9x9_mul(_t3, _x1, _t1);
/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_9terms(_t2, _t3, _t2);
XMM_GF2m_mod_sqr_nist571(_x1, _t3);
/* MDOUBLE */
XMM_GF2m_mod_sqr_nist571(_x2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _z2);
XMM_GF2m_add_5term(_t1, _x2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _z2, _x2);
XMM_GF2m_mod_sqr_nist571(_z2, _t1);
/* Unveil data */
XMM_GF2m_veil_5term(_tx1, _tx2, _tz1, _tz2, x1, _x1, _z2, _z2, _z1, _x2, _z2, k);
/* Store results */
XMM_to_BN_9term(x1 ->d, _tx1);
XMM_to_BN_9term(z1 ->d, _tz1);
XMM_to_BN_9term(x2 ->d, _tx2);
XMM_to_BN_9term(z2 ->d, _tz2);
ret = 1;
return ret;
}
BN_to_XMM_9term(_t2, z2->d);
BN_to_XMM_9term(_t2, x2->d);

/* Data veiling */
XMM_GF2m_veil_5term(_x1, _z1, _x2, _z2, _tx1, _tz1, _tx2, _tz2, k);

/* MADD */
XMM_GF2m_mod_mul_nist571(_x1, _x1, _x2);
XMM_GF2m_mod_mul_nist571(_z1, _z1, _z2);

/* Multiply w/o reduction */
XMM_GF2m_9x9_mul(_t2, _x1, _z1);
XMM_GF2m_9x9_mul(_z1, _z1, _x2);

/* Multiply w/o reduction */
BN_to_XMM_9term(_t1, x->d);
XMM_GF2m_9x9_mul(_t3, _z1, _t1);

/* Add the two double-sized numbers and reduce */
XMM_GF2m_add_9term(_t3, _t3, _t2);
XMM_GF2m_mod_nist571(_x1, _t3);

/* MDDOUBLE */
XMM_GF2m_mod_sqr_nist571(_x2, _z2);
XMM_GF2m_mod_sqr_nist571(_z2, _x2);
BN_to_XMM_9term(_t1, c->d);
XMM_GF2m_mod_mul_nist571(_t1, _z2, _t1);
XMM_GF2m_mod_mul_nist571(_z2, _x2, _z2);
XMM_GF2m_mod_mul_nist571(_z2, _x2, _t1);
XMM_GF2m_mod_sqr_nist571(_z2, _x2);

/* Unveil data */
XMM_GF2m_veil_5term(_tx1, _tz1, _tx2, _tz2, _x1, _z1, _x2, _z2, k);

/* Store results */
XMM_to_BN_9term(x1->d, _tx1);
XMM_to_BN_9term(z1->d, _tz1);
XMM_to_BN_9term(x2->d, _tx2);
XMM_to_BN_9term(z2->d, _tz2);
ret = 1;
return ret;
} #endif
D. XMM Test Utilities

```c
/* SPEED UTILITY
   * This utility provides a macro to measure the amount of clock cycles for one or more
   * functions.
   */
#include <stdio.h>
#include <stdlib.h>
#define TEST_RUNS 5

/* This function fetches clocks */
inline unsigned long get_clks ( void )
{
    unsigned long long ret_val;
    __asm__ volatile
    ( "cpuid\n      rdtsc\n      mov %eax,(%0)\n      mov %edx,4(%0):"rm"(& ret_val):"eax","edx","ebx","ecx"
    );
    return ret_val;
}

/*
This MACRO measures the number of cycles "x" runs. This is the flow:
1) it repeats "x" WARMUP times, in order to warm the cache.
2) it reads the Time Stamp Counter at the beginning of the test.
3) it repeats "x" REPEAT number of times.
4) it reads the Time Stamp Counter again at the end of the test
5) it calculates the average number of cycles per one iteration of "x", by calculating
the total number of cycles, and dividing it by REPEAT
 */
#define SPEED_INIT (x) /* INIT */
    unsigned long long start_clk , end_clk ;
    double total_clk , tmp_clk ;
    int round , run ;
    int rounds = x;
    int warmups = rounds /4;
#define SPEED_MEASURE ( function , description )
    printf ("SPEED MEASUREMENT FOR FUNCTION :\t %s\n", description );
    printf("RUN \t\tROUNDS\t\tCYCLES\n");
    printf("--------\t\t------\t\t------\n");
    /* Do WARMUPS */
    for (round = 0; round < warmups ; round++)
    {
        function ;
    }
    total_clk = 1.7976931348623157e+308;
    /* Do MEASUREMENT */
    for (run = 0; run < TEST_RUNS ; run++)
    {
        start_clk = get_clks();
        for (round = 0; round < rounds ; round++)
        {
            function ;
        }
        end_clk = get_clks();
        tmp_clk = ( double )( end_clk - start_clk ) / rounds ;
        if( total_clk > tmp_clk ) total_clk = tmp_clk ;
    }
    /* OUTPUT */
```

196
Listing D.1: Cyclecounter

```c
#include <stdio.h>
#include <openssl/bn.h>
#include <openssl/rand.h>
#include <openssl/bio.h>

#define MAX_SIZE 2*571

static const char rnd_seed[] = "string to make the random number generator think it has
text entropy :)

inline void add2(unsigned long int z[4], unsigned long int a[4], unsigned long int b[4])
{
    z[0] = a[0] ^ b[0];
    z[1] = a[1] ^ b[1];
}

inline void add3(unsigned long int z[6], unsigned long int a[6], unsigned long int b[6])
{
    z[0] = a[0] ^ b[0];
    z[1] = a[1] ^ b[1];
}

inline void add4(unsigned long int z[8], unsigned long int a[8], unsigned long int b[8])
{
    z[0] = a[0] ^ b[0];
    z[1] = a[1] ^ b[1];
}

inline void add5(unsigned long int z[10], unsigned long int a[10], unsigned long int b[10])
{
    z[0] = a[0] ^ b[0];
    z[1] = a[1] ^ b[1];
    z[8] = a[8] ^ b[8];
}

inline void add7(unsigned long int z[14], unsigned long int a[14], unsigned long int b[14])
{

```
inline void add9(unsigned long int z[18], unsigned long int a[18], unsigned long int b[18])
{
    z[0] = a[0] ^ b[0];
    z[1] = a[1] ^ b[1];
    z[8] = a[8] ^ b[8];
    z[12] = a[12] ^ b[12];
    z[14] = a[14] ^ b[14];
    z[16] = a[16] ^ b[16];
    z[17] = a[17] ^ b[17];
}

int test_xmm(BIO *out, BN_CTX *ctx)
{
    int ret = 0;
    BIGNUM *a, *b, *z;
    SPEED_INIT(100000);
    __m128i _a[MAX_SIZE /128+1], _b[MAX_SIZE /128+1], _z[MAX_SIZE /128+1];
    BN_CTX_start(ctx);
    BIO_printf(out, "--- START XMM FIELD TESTS ---\n");
    if ((a = BN_CTX_get(ctx))== NULL) goto err;
    if ((b = BN_CTX_get(ctx))== NULL) goto err;
    if ((z = BN_CTX_get(ctx))== NULL) goto err;
    if (!BN_bntest_rand(a, MAX_SIZE , 0, 0)) goto err;
    if (!BN_bntest_rand(b, MAX_SIZE , 0, 0)) goto err;
    if (!BN_bntest_rand(z, MAX_SIZE , 0, 0)) goto err;
    /* Load */
    BN_to_XMM_conv(a, a, MAX_SIZE/64+1);
    BN_to_XMM_conv(b, b, MAX_SIZE/64+1);
    /* Start tests */
    /*
    * XMM ADDITION
    */
    rounds = 1000000000;
    SPEED_MEASURE( XMM_GF2m_add_2term(_z, _a, _b), "XMM_GF2m_add_2term" );
    SPEED_MEASURE( XMM_GF2m_add_3term(_z, _a, _b), "XMM_GF2m_add_3term" );
    SPEED_MEASURE( XMM_GF2m_add_4term(_z, _a, _b), "XMM_GF2m_add_4term" );
    SPEED_MEASURE( XMM_GF2m_add_5term(_z, _a, _b), "XMM_GF2m_add_5term" );
    SPEED_MEASURE( XMM_GF2m_add_7term(_z, _a, _b), "XMM_GF2m_add_7term" );
    SPEED_MEASURE( XMM_GF2m_add_9term(_z, _a, _b), "XMM_GF2m_add_9term" );
}

SPEED_MEASURE( add2( z->d, a->d, b->d), "64 bit add_2term" );
SPEED_MEASURE( add3( z->d, a->d, b->d), "64 bit add_3term" );
SPEED_MEASURE( add4( z->d, a->d, b->d), "64 bit add_4term" );
SPEED_MEASURE( add7( z->d, a->d, b->d), "64 bit add_7term" );
SPEED_MEASURE( add9( z->d, a->d, b->d), "64 bit add_9term" );

/*
  *******************************************************
  * XMM VEILING
  *******************************************************
*/
rounds = 100000000;
SPEED_MEASURE( XMM_GF2m_veil_2term( _z , _a , _b , _a , _b , _a , _b , _a , 1), " XMM_GF2m_veil_2term " );
SPEED_MEASURE( XMM_GF2m_veil_3term( _z , _a , _b , _a , _b , _a , _b , _a , 1), " XMM_GF2m_veil_3term " );
SPEED_MEASURE( XMM_GF2m_veil_4term( _z , _a , _b , _a , _b , _a , _b , _a , 1), " XMM_GF2m_veil_4term " );
SPEED_MEASURE( XMM_GF2m_veil_5term( _z , _a , _b , _a , _b , _a , _b , _a , 1), " XMM_GF2m_veil_5term " );

/*
  *******************************************************
  * XMM SQUARING
  *******************************************************
*/
rounds = 1000000000;
SPEED_MEASURE( XMM_GF2m_sqr_3term( _z , _a), " XMM_GF2m_sqr_3term " );
SPEED_MEASURE( XMM_GF2m_sqr_4term( _z , _a), " XMM_GF2m_sqr_4term " );
SPEED_MEASURE( XMM_GF2m_sqr_5term( _z , _a), " XMM_GF2m_sqr_5term " );
SPEED_MEASURE( XMM_GF2m_sqr_7term( _z , _a), " XMM_GF2m_sqr_7term " );
SPEED_MEASURE( XMM_GF2m_sqr_9term( _z , _a), " XMM_GF2m_sqr_9term " );
SPEED_MEASURE( XMM_GF2m_sqr_sect163( _z , _a), " XMM_GF2m_sqr_sect163 " );
SPEED_MEASURE( XMM_GF2m_sqr_sect193( _z , _a), " XMM_GF2m_sqr_sect193 " );
SPEED_MEASURE( XMM_GF2m_sqr_sect233( _z , _a), " XMM_GF2m_sqr_sect233 " );
SPEED_MEASURE( XMM_GF2m_sqr_sect239( _z , _a), " XMM_GF2m_sqr_sect239 " );
SPEED_MEASURE( XMM_GF2m_sqr_sect283( _z , _a), " XMM_GF2m_sqr_sect283 " );
SPEED_MEASURE( XMM_GF2m_sqr_sect409( _z , _a), " XMM_GF2m_sqr_sect409 " );
SPEED_MEASURE( XMM_GF2m_sqr_sect571( _z , _a), " XMM_GF2m_sqr_sect571 " );

/*
  *******************************************************
  * XMM MULTIPLICATION
  *******************************************************
*/
rounds = 100000000;
SPEED_MEASURE( XMM_GF2m_2x2_mul( _z ,_a[0] , _b[0]), "XMM_GF2m_2x2_mul" );
SPEED_MEASURE( XMM_GF2m_3x3_mul( _z , _a , _b), "XMM_GF2m_3x3_mul" );
SPEED_MEASURE( XMM_GF2m_4x4_mul( _z , _a , _b), "XMM_GF2m_4x4_mul" );
SPEED_MEASURE( XMM_GF2m_5x5_mul( _z , _a , _b), "XMM_GF2m_5x5_mul" );
SPEED_MEASURE( XMM_GF2m_7x7_mul( _z , _a , _b), "XMM_GF2m_7x7_mul" );
SPEED_MEASURE( XMM_GF2m_9x9_mul( _z , _a , _b), "XMM_GF2m_9x9_mul" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect163( _z , _a , _b), "XMM_GF2m_mod_mul_sect163" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect193( _z , _a , _b), "XMM_GF2m_mod_mul_sect193" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect233( _z , _a , _b), "XMM_GF2m_mod_mul_sect233" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect239( _z , _a , _b), "XMM_GF2m_mod_mul_sect239" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect283( _z , _a , _b), "XMM_GF2m_mod_mul_sect283" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect409( _z , _a , _b), "XMM_GF2m_mod_mul_sect409" );
SPEED_MEASURE( XMM_GF2m_mod_mul_sect571( _z , _a , _b), "XMM_GF2m_mod_mul_sect571" );

/*
  *******************************************************
  * XMM LAZY REDUCTION OPERATIONS
  *******************************************************
*/
rounds = 1000000000;

/* Square + Multiply */
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect163( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect163" );
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect193( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect193" );
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect233( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect233" );
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect239( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect239" );
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect283( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect283" );
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect409( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect409" );
SPEED_MEASURE( XMM_GF2m_addsqrmul_sect571( _z , _a , _b , _a ), "XMM_GF2m_addsqrmul_sect571" );

/* Multiply + Multiply */
SPEED_MEASURE( XMM_GF2m_add2mul_sect163( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect163" );
SPEED_MEASURE( XMM_GF2m_add2mul_sect193( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect193" );
SPEED_MEASURE( XMM_GF2m_add2mul_sect233( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect233" );
SPEED_MEASURE( XMM_GF2m_add2mul_sect239( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect239" );
SPEED_MEASURE( XMM_GF2m_add2mul_sect283( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect283" );
SPEED_MEASURE( XMM_GF2m_add2mul_sect409( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect409" );
SPEED_MEASURE( XMM_GF2m_add2mul_sect571( _z , _a , _b , _a , _b ), "XMM_GF2m_add2mul_sect571" );

/* XMM INVERSION */

*/

rounds = 10000000;

SPEED_MEASURE( XMM_GF2m_mod_inv_sect163( _z , _a ), "XMM_GF2m_mod_inv_sect163" );
SPEED_MEASURE( XMM_GF2m_mod_inv_sect193( _z , _a ), "XMM_GF2m_mod_inv_sect193" );
SPEED_MEASURE( XMM_GF2m_mod_inv_sect233( _z , _a ), "XMM_GF2m_mod_inv_sect233" );
SPEED_MEASURE( XMM_GF2m_mod_inv_sect239( _z , _a ), "XMM_GF2m_mod_inv_sect239" );
SPEED_MEASURE( XMM_GF2m_mod_inv_sect283( _z , _a ), "XMM_GF2m_mod_inv_sect283" );
SPEED_MEASURE( XMM_GF2m_mod_inv_sect409( _z , _a ), "XMM_GF2m_mod_inv_sect409" );
SPEED_MEASURE( XMM_GF2m_mod_inv_sect571( _z , _a ), "XMM_GF2m_mod_inv_sect571" );

SPEED_MEASURE( XMM_GF2m_div_sect163( _z , _a , _b ), "XMM_GF2m_div_sect163" );
SPEED_MEASURE( XMM_GF2m_div_sect193( _z , _a , _b ), "XMM_GF2m_div_sect193" );
SPEED_MEASURE( XMM_GF2m_div_sect233( _z , _a , _b ), "XMM_GF2m_div_sect233" );
SPEED_MEASURE( XMM_GF2m_div_sect239( _z , _a , _b ), "XMM_GF2m_div_sect239" );
SPEED_MEASURE( XMM_GF2m_div_sect283( _z , _a , _b ), "XMM_GF2m_div_sect283" );
SPEED_MEASURE( XMM_GF2m_div_sect409( _z , _a , _b ), "XMM_GF2m_div_sect409" );
SPEED_MEASURE( XMM_GF2m_div_sect571( _z , _a , _b ), "XMM_GF2m_div_sect571" );

/* XMM REDUCTION */

*/

rounds = 1000000000;

SPEED_MEASURE( XMM_GF2m_mod_sect163( _z , _a ), "XMM_GF2m_mod_sect163" );
SPEED_MEASURE( XMM_GF2m_mod_sect163_pclmul( _z , _a ), "XMM_GF2m_mod_sect163_pclmul" );
SPEED_MEASURE( XMM_GF2m_mod_sect193( _z , _a ), "XMM_GF2m_mod_sect193" );
SPEED_MEASURE( XMM_GF2m_mod_sect233( _z , _a ), "XMM_GF2m_mod_sect233" );
SPEED_MEASURE( XMM_GF2m_mod_sect239( _z , _a ), "XMM_GF2m_mod_sect239" );
SPEED_MEASURE( XMM_GF2m_mod_sect283( _z , _a ), "XMM_GF2m_mod_sect283" );
SPEED_MEASURE( XMM_GF2m_mod_sect283_pclmul( _z , _a ), "XMM_GF2m_mod_sect283_pclmul" );
SPEED_MEASURE( XMM_GF2m_mod_sect409( _z , _a ), "XMM_GF2m_mod_sect409" );
SPEED_MEASURE( XMM_GF2m_mod_sect571( _z , _a ), "XMM_GF2m_mod_sect571" );
SPEED_MEASURE( XMM_GF2m_mod_sect571_pclmul( _z , _a ), "XMM_GF2m_mod_sect571_pclmul" );

BIO_printf(out, "\n--- XMM FIELD TESTS DONE ---\n\n");
```c
ret = 1;
err:
    BN_CTX_end(ctx);
    return ret;
}

/* GF2m field operation tests */
int main(int argc, char *argv[])
{
    BN_CTX *ctx;
    BIO *out;
    out = BIO_new_fp(stdout, BIO_NOCLOSE);
    RAND_seed(rnd_seed, sizeof rnd_seed); /* or BN_generate_prime may fail */
    ctx = BN_CTX_new();
    if (ctx == NULL) return(0);
    /* Measure XMM field operations */
    if (!test_xmm(out, ctx)) goto err;
    BN_CTX_free(ctx);
    err:
    return(1);
}
```

Listing D.2: XMMtests
E. Declaration

I hereby declare that this Masters thesis is my own work, and it does not contain other peoples work without this being stated; and was not submitted to any other institute, and that the bibliography contains all the literature that I have used in writing the thesis, and that all references refer to this bibliography.

Hiermit versichere ich, dass ich die Master-Arbeit selbstständig und ausschließlich mit den aus den Richtlinien zu vereinbarenden Hilfsmitteln erstellt habe, dass diese Master-Arbeit keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt wurde, und dass ich jegliche wörtlich und sinngemäß aus veröffentlichten oder nicht veröffentlichten Quellen übernommenen Passagen eindeutig kenntlich gemacht habe.

Bochum, June 17, 2013

Manuel Bluhm