A Cryptographic Processor for Arbitrary Elliptic Curves over $GF(2^m)$

Hans Eberle, Nils Gura,
Sheueling Chang Shantz, and Vipul Gupta

A Cryptographic Processor for Arbitrary Elliptic Curves over GF(2^m)

Hans Eberle, Nils Gura,
Sheueling Chang Shantz, and Vipul Gupta

SMLI TR-2003-123 May 2003

Abstract:

We describe a cryptographic processor for Elliptic Curve Cryptography (ECC). ECC is evolving as an attractive alternative to other public-key cryptosystems such as the Rivest-Shamir-Adleman algorithm (RSA) by offering the smallest key size and the highest strength per bit. The cryptographic processor performs point multiplication for elliptic curves over binary polynomial fields GF(2^m). In contrast to other designs that only support one curve at a time, our processor is capable of handling arbitrary curves without requiring reconfiguration. More specifically, it can handle both named curves as standardized by the National Institute for Standards and Technology (NIST) as well as any other generic curves up to a field degree of 255. Efficient support for arbitrary curves is particularly important for the targeted server applications that need to handle requests for secure connections generated by a multitude of heterogeneous client devices. Such requests may specify curves which are infrequently used or not even known at implementation time.

We have implemented the cryptographic processor in a field-programmable gate array (FPGA) running at a clock frequency of 66.4 MHz. Its performance is 6955 point multiplications per second for named curves over GF(2^{163}) and 3308 point multiplications per second for generic curves over GF(2^{163}). We have integrated the cryptographic processor into the open source toolkit OpenSSL, which implements the Secure Sockets Layer (SSL) which is today's dominant Internet security protocol.

This report is an extended version of a paper presented at the IEEE 14th International Conference on Application-specific Systems, Architectures and Processors, The Hague, June 2003 where it received the "Best Paper Award".

M/S MTV29-01
2600 Casey Avenue
Mountain View, CA 94043

email address:
hans.eberle@sun.com
nils.gura@sun.com
sheueling.chang@sun.com
vipul.gupta@sun.com

Copyright 2003 Sun Microsystems, Inc. and The Institute of Electrical and Electronics Engineers, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, and Sun Fire 280R are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.All technical reports are available online on our Website, http://research.sun.com/techrep/.

A Cryptographic Processor for Arbitrary Elliptic Curves over GF(2^m)

Hans Eberle, Nils Gura, Sheueling Chang Shantz, and Vipul Gupta
Sun Microsystems Laboratories
2600 Casey Avenue
Mountain View, CA 94043
{Nils.Gura, Hans.Eberle, Sheueling.Chang, Vipul.Gupta}@sun.com

1 Introduction

In this report, we describe the architecture and implementation of a processor for Elliptic Curve Cryptography (ECC). ECC is a public-key cryptosystem that is rapidly evolving as an attractive alternative to other schemes such as the Rivest-Shamir-Adleman algorithm (RSA) by offering the smallest key size and the highest strength per bit. For example, a 163-bit ECC key offers the same security strength as a 1024-bit RSA key. The ratio of key sizes is going to favor ECC even more as larger keys are adopted. To illustrate this with another example, a 571-bit ECC key is comparable in security strength to a 15,360-bit RSA key. As a result of the smaller key size, some cryptographic operations such as signing can be executed much faster with ECC [11]. For this reason, ECC is most interesting to emerging wireless technologies that use Internet-ready mobile phones, PDAs, smart cards and sensor networks. Since these types of client devices have modest to little compute resources, the computational efficiency makes ECC a good match.

Several standards have been created to specify the use of ECC. The United States government has adopted ECC for the Elliptic Curve Digital Signature Algorithm (ECDSA) and recommended a set of curves. For binary polynomial fields, the curves are specified for key sizes of 163, 233, 283, 409, and 571 bit [21]. Additional curves for commercial use were recommended by the Standards for Efficient Cryptography Group (SECG) [6]. Also, efforts are underway to include ECC into security protocols. In particular, ECC has recently been integrated into the open source toolkit OpenSSL which implements the Secure Sockets Layer (SSL), today's dominant Internet security protocol [18, 10]; SSL is used by numerous applications, the most well-known being the Apache webserver. Given the endorsement by the United States government, the existence of ECC standards, and the availability of ECC-enabled security protocols, we can expect that ECC-enabled client devices will soon appear on the market, thus creating demand for high-performance ECC implementations on server machines.

General-purpose CPUs are not optimized for fast execution of cryptographic algorithms such as RSA and ECC mainly because they lack instructions for modular arithmetic operations on long operands. Typical operand lengths are 1024 bits for RSA and 163 to 571 bits for ECC. Modular arithmetic operations include multiplication and squaring for RSA and multiplication, squaring, division, and addition for ECC. Implementations of ECC are further complicated as it is defined for both prime integer fields $GF(p)$ and binary poly-

nomial fields $GF(2^m)$, the latter requiring arithmetic operations that cannot be performed by standard integer or floating-point units.

Dedicated cryptographic hardware is used on server machines to optimize the throughput of secure web-based applications. Servers running security protocols such as SSL or the Internet Protocol Security (IPSec) are confronted with an aggregation of secure connections created by a multitude of heterogeneous clients. Terminating secure connections on the server side not only demands high computational power but also flexibility in responding to client devices that are limited in the set of cryptographic algorithms supported. As clients are often limited in processing power and memory capacity, they may be capable of supporting only a small number of curves. With respect to ECC, a client might possibly support only a single curve. To be able to establish a secure connection and, with it, provide service, a server, in turn, is required to be flexible enough to support any such curve requested by a client.[1] While a server certainly needs to implement the standardized curves and the associated irreducible polynomials, it should further implement any other arbitrary curve and, thus, any arbitrary irreducible polynomial. In the following, we refer to the former as *named curves* and to the latter as *generic curves*. There are several reasons why generic curves need to be supported. The standards only recommend curves and, thus, new curves might emerge in the future. Furthermore, curves might be abandoned for security reasons and replaced by different ones that were not known at implementation time.

While previous work has targeted implementations optimized for specific curves, our design has the unique property of providing optimized performance for multiple named curves and support for arbitrary generic curves. In a previous publication [13], we introduced a technique called *partial reduction* that allows for generic curve-independent implementations of ECC. While our previous publication only described a firmware implementation of this technique, we are now introducing a novel digit-serial multiplier that implements modular multiplication in hardware for both named and generic curves. By adding hardware support for generic curves, we were able to significantly increase performance for generic curves over $GF(2^{163})$ from 1075 point multiplications per second for the firmware implementation to 3308 point multiplications per second for the new design. Compared with 6955 point multiplications per second for named curves over $GF(2^{163})$, the performance penalty for named curves is now roughly a factor of two which is low given the complexity of the problem.

The report is structured as follows. Section 2 summarizes related work. Section 3 describes the ECC-enhanced protocol stack that interfaces the cryptographic processor. Section 4 briefly explains ECC point multiplication. In Section 5, we describe the arithmetic underlying ECC and address the problem of modular reduction. The architecture and implementation of the cryptographic processor is presented in Section 6. The program code for the point multiplication is shown in Section 7. Section 8 analyzes the design and gives performance numbers. Finally, Section 9 contains the conclusions.

[1]While the current version of TLS (see footnote on page 3) only allows for the server to choose the curve, future extensions could consider the capabilities of both the server and the client.

2 Related Work

Hardware and firmware implementations of ECC point multiplication over different fields $GF(2^m)$ have been reported in numerous publications. A design for $GF((2^8-17)^{17})$, optimized for 8-bit processors, is described by Woodbury et al. in [22]. The implementation targets a SmartCard based on an Intel 8051 microcontroller. Orlando and Paar describe a programmable elliptic curve processor for reconfigurable logic in [19]. Different curves can be handled by parameterizing the hardware architecture and reconfiguring the logic. The prototype performs point multiplication on curves over $GF(2^{167})$. Bednara et al. [2] designed an FPGA-based cryptographic processor architecture that allows for using multiple squarers, adders and multipliers. They researched hybrid coordinate representations in affine, projective, Jacobian and López-Dahab form. Two prototypes were synthesized for $GF(2^{191})$. Agnew et al. [1] built an application-specific integrated circuit implementing ECC point multiplication for $GF(2^{155})$. The chip uses an optimal normal basis multiplier exploiting the composite field property of $GF(2^{155})$. Goodman and Chandrakasan [9] designed a generic public-key processor that executes modular operations on integer and binary polynomial fields. The internal data path can be reconfigured to support different field degrees. Point multiplication over binary polynomial fields is computed by a microcoded double-and-add algorithm. To our knowledge, this is the only implementation that supports $GF(2^m)$ for variable field degrees m. However, the architecture is optimized for low power consumption and its performance cannot be scaled to levels required by server-type applications. All other implementations described above target either one or a small number of specific curves. That is, none of them can handle a curve that is not specified at implementation time without requiring the software to be modified or the hardware to be reconfigured.

3 ECC-enabled Secure Protocol Stack

This section provides the context for the work described in this report by outlining the ECC-enhanced protocol stack that interfaces the cryptographic processor. Figure 1 shows the implemented client/server system. We integrated new cipher suites based on ECC into OpenSSL [18], the most widely used open-source implementation of the Secure Sockets Layer (SSL). More specifically, we added the Elliptic Curve Digital Signature Algorithm (ECDSA), the Elliptic Curve Diffie-Hellman key exchange (ECDH), and means to generate and process X.509 certificates containing ECC keys. We validated our implementation by integrating it with the Apache web server and open-source web browsers Dillo and Lynx running on a handheld client device under Linux. The cryptographic processor accelerates public-key operations on the server side, where client connections are aggregated. The processor is connected to the host machine through an IO interface based on the Peripheral Component Interface (PCI) standard and accessed by a character device driver running under the Solaris™Operating Environment.

The SSL and TLS protocols[2] [8] are the most widely deployed and used security protocol on the Internet today. The protocol has withstood years of scrutiny by the security

[2]TLS is being developed by the Internet Engineering Task Force (IETF); it is based on SSL and intended to replace SSL.

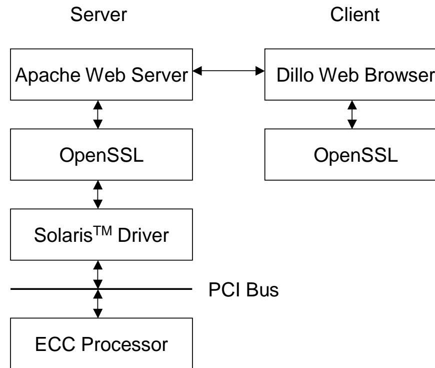

Figure 1. *Secure Client/Server System.*

community and, in the form of the Secure Hyptertext Transport Protocol (HTTPS),[3] is now trusted to secure virtually all sensitive web-based applications ranging from banking to online trading to electronic commerce.

SSL offers encryption, source authentication and integrity protection for data exchanged over insecure, public networks. It operates above a reliable transport service such as the Transport Layer Protocol (TCP) and has the flexibility to accommodate different cryptographic algorithms for key agreement, encryption and hashing. However, the specification does recommend particular combinations of these algorithms, called *cipher suites*, which have well-understood security properties.

The two main components of SSL are the Handshake protocol and the Record Layer protocol. The Handshake protocol allows an SSL client and server to negotiate a common cipher suite, authenticate each other,[4] and establish a shared *master secret* using public-key algorithms. The Record Layer derives symmetric keys from the master secret and uses them with faster symmetric-key algorithms for bulk encryption and authentication of application data. Public-key cryptographic operations are the most computationally expensive portion of SSL processing, and speeding them up remains an active area for research and development.

Figure 2 shows the general structure of a full SSL handshake. Today, the most commonly used public-key cryptosystem for master-key establishment is RSA but the IETF is considering an equivalent mechanism based on ECC [5]. In the following paragraphs we briefly describe the SSL handshakes for RSA- and ECC-based cipher suites, respectively.

RSA-based Handshake: The client and server exchange random nonces[5] and negotiate a cipher suite with ClientHello and ServerHello messages. The server then sends its signed RSA public-key either in the Certificate message or the ServerKeyExchange message. The client verifies the RSA signature, generates a 48-byte random number (the *pre-master secret*) and sends it encrypted with the server's public-key in the ClientKeyExchange. The server uses its RSA private key to decrypt the pre-master secret. Both end-points then use

[3]HTTPS is HTTP over an SSL-secured connection.

[4]Client authentication is optional. Only the server is typically authenticated at the SSL layer and client authentication is achieved at the application layer, e.g., through the use of passwords sent over an SSL-protected channel. However, some deployment scenarios do require stronger client authentication through certificates.

[5]Nonces are single-use random numbers to guard against replay attacks.

the pre-master secret to create a master secret, which, along with previously exchanged nonces, is used to derive the cipher keys, initialization vectors and Message Authentication Code (MAC) keys for bulk encryption by the Record Layer.

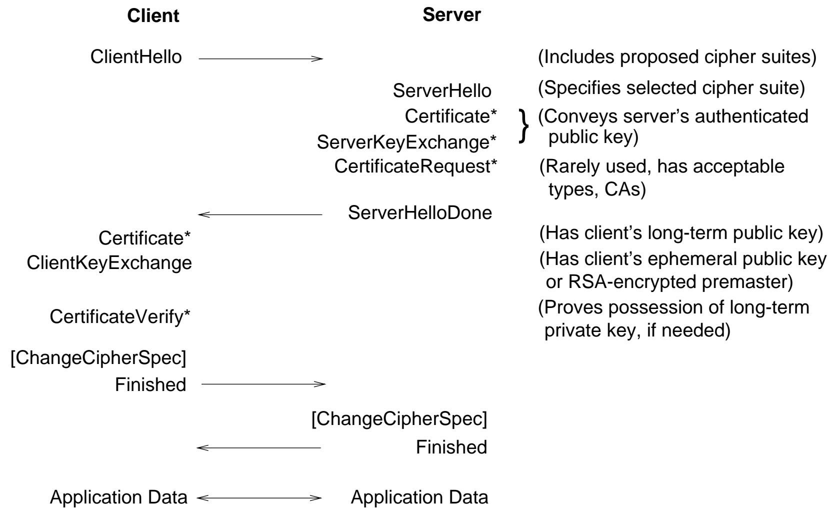

Figure 2. *SSL Handshake for an RSA-based Cipher Suite.*

The server can optionally request client authentication by sending a CertificateRequest message listing acceptable certificate types and certificate authorities. In response, the client sends its private key in the Certificate and proves possession of the corresponding private key by including a digital signature in the CertificateVerify message.

ECC-based Handshake: The processing of the first two messages is the same as for RSA but the Certificate message contains the server's Elliptic Curve Diffie-Hellman (ECDH) public key signed with the Elliptic Curve Digital Signature Algorithm (ECDSA). After validating the ECDSA signature, the client conveys its ECDH public key in the ClientKeyExchange message. Next, each entity uses its own ECDH private key and the other's public key to perform an ECDH operation and arrive at a shared pre-master secret. The derivation of the master secret and symmetric keys is unchanged compared to RSA. Client authentication is still optional and the actual message exchange depends on the type of certificate a client possesses.

4 Point Multiplication

The fundamental and most expensive operation underlying ECC is *point multiplication*, which is defined over finite field operations.[6] The point multiplication kP of an integer k and a point P on an elliptic curve $C : y^2 + xy = x^3 + ax^2 + b; x, y \in GF(2^m)$ with curve parameters $a, b \in GF(2^m)$ over a binary polynomial field $GF(2^m)$ can be decomposed into *point additions* and *point doublings*. For example, $9P$ can be computed with one point addition and three point doublings since $9P = P + 2 * 2 * 2P$.

[6] For a detailed mathematical background on ECC the reader is referred to [4].

Various algorithms have been proposed to efficiently compute point multiplications.[7] We experimented with different point multiplication algorithms and settled on Montgomery's point multiplication algorithm using projective coordinates as proposed by López and Dahab [17]. Affine point coordinates (x, y) are represented as projective triples (X, Y, Z) with $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$ to avoid expensive divisions. Montgomery's algorithm exploits the fact that for a fixed point $P = (x, y) = (X, Y, 1)$ and points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$, $2P_1$, $2P_2$ and $P_1 + P_2$ can be expressed through only the X- and Z-coordinates of P, P_1 and P_2 if $P_2 = P_1 + P$. A point multiplication kP can be computed with $\lfloor log_2(k) \rfloor$ point additions and doublings. The computation is done as follows. The bits of the binary representation of k are examined from left ($k_{\lfloor log_2(k) \rfloor}$) to right ($k_0$). For the first non-zero bit of k, P_1 and P_2 are initialized with $P_{1,\lfloor log_2(k) \rfloor} = P$ and $P_{2,\lfloor log_2(k) \rfloor} = 2P$:

$$
\begin{aligned}
X_{1,\lfloor log_2(k) \rfloor} &= x \\
Z_{1,\lfloor log_2(k) \rfloor} &= 1 \\
X_{2,\lfloor log_2(k) \rfloor} &= x^4 + b \\
Z_{2,\lfloor log_2(k) \rfloor} &= x^2
\end{aligned}
$$

For all following bits of k, with $k_i = 0$, $P_{1,i}$ is set to $2P_{1,i+1}$ (1) and $P_{2,i}$ is set to $P_{1,i+1} + P_{2,i+1}$ (2).

$$
\begin{aligned}
X_{1,i} &= X_{1,i+1}^4 + bZ_{1,i+1}^4 \\
Z_{1,i} &= Z_{1,i+1}^2 * X_{1,i+1}^2 && (1) \\
X_{2,i} &= xZ_{2,i} + (X_{1,i+1}Z_{2,i+1})(X_{2,i+1}Z_{1,i+1}) \\
Z_{2,i} &= (X_{1,i+1} * Z_{2,i+1} + X_{2,i+1} * Z_{1,i+1})^2 && (2)
\end{aligned}
$$

Similarly, for $k_i = 1$, $P_{1,i}$ is set to $P_{1,i+1} + P_{2,i+1}$ and $P_{2,i}$ is set to $2P_{2,i+1}$. The Y-coordinate of kP can be retrieved from its X- and Z-coordinates using the curve equation. The result $kP = (x_{kP}, y_{kP})$ in affine coordinates is given by

$$
\begin{aligned}
x_{kP} &= \frac{X_{1,0}}{Z_{1,0}} \\
y_{kP} &= (\frac{X_{1,0}}{Z_{1,0}} + x) * \frac{(\frac{X_{1,0}}{Z_{1,0}} + x)(\frac{X_{2,0}}{Z_{2,0}} + x) + x^2 + y}{x} + y \\
kP &= 0 \quad \text{if} \quad Z_{1,0} = 0 \\
kP &= (x, x + y) \quad \text{if} \quad Z_{2,0} = 0
\end{aligned}
$$

Using projective coordinates, Montgomery point multiplication requires $6\lfloor log_2(k) \rfloor + 9$ multiplications, $5\lfloor log_2(k) \rfloor + 3$ squarings, $3\lfloor log_2(k) \rfloor + 7$ additions and 1 division.

5 ECC Arithmetic in GF(2^m)

ECC over finite fields is based on modular addition, subtraction, multiplication, squaring and division. In this report, we will focus on binary polynomial fields $GF(2^m)$. Using

[7] For a survey of point multiplication algorithms the reader is referred to [15].

```
1011  *   1001 (t^3+t+1) * (t^3+1)
--------------
          1001
xor      1001
xor     0000
xor    1001
--------------
=      1010011 (t^6 + t^4 + t + 1)
```

Figure 3. *Polynomial Multiplication.*

```
    1010011  (t^6 + t^4 + t + 1)
xor 11001    (t^6 + t^5 + t^2)
    -------
    0110111  (t^5 + t^4 + t^2 + t + 1)
xor  11001   (t^5 + t^4 + t^1)
    -------
=   0000101  (t^2 + 1)
```

Figure 4. *Polynomial Reduction.*

polynomial bases, a polynomial $a \in GF(2^m)$ in canonical form can be written as $a = a_{m-1}t^{m-1} + a_{m-2}t^{m-2} + \cdots + a_1 t + a_0, a_i \in GF(2)$.

The addition of two elements $a, b \in GF(2^m)$ is defined as the sum of the two polynomials obtained by adding the coefficients a_i and b_i, which corresponds to a bitwise XOR operation. For example, a polynomial addition $(t^4 + t^2 + 1) + (t^3 + t^2 + t) = t^4 + t^3 + t + 1$ can be computed as 10101 xor 1110 = 11011. Since every element of $GF(2^m)$ is identical to its additive inverse, subtraction is identical to addition.

Multiplication of two elements $a, b \in GF(2^m)$ is carried out in two steps. First, the operands are multiplied using polynomial multiplication resulting in

$$\begin{aligned} c_0 = a * b = & c_{0,2(m-1)}t^{2(m-1)} \\ & + c_{0,2(m-1)-1}t^{2(m-1)-1} \\ & + \cdots + c_{0,1}t + c_{0,0} \end{aligned}$$

The degree of c_0 is less than $2m-1$, i.e. $\deg(c_0) < 2m-1$. The coefficients of c_0 are calculated through convolution of a and b

$$c_{0,j} = \sum_{k=0}^{j} a_k b_{j-k} \tag{3}$$

c_0 may not be in reduced canonical form since its degree may be greater than $m-1$. Second, c_0 is reduced by an irreducible polynomial M. M is of degree m and defines $GF(2^m)$ for a chosen field degree m. The reduced canonical result $c \equiv c_0 \mod M$, $\deg(c) < m$ is defined as the residue of the polynomial division of c_0 by M. For example, given polynomials $a = t^3 + t + 1$ and $b = t^3 + 1$ of $GF(2^4)$, represented as $a = 1011$ and $b = 1001$, $c_0 = a * b = t^6 + t^4 + t + 1$ can be computed as shown in Figure 3. Assuming $M = t^4 + t^3 + 1$, represented as $M = 11001$, the reduction $c = c_0 \mod M = t^2 + 1$ can be performed as shown in Figure 4. An illustrative way to look at reduction is that M is aligned with the most significant bit of the operand and added until the degree of the result is smaller than m.

Polynomial multiplication can be efficiently implemented using well-known techniques such as the Ofman-Karatsuba method [16]. Field multiplication, i.e. polynomial multiplication combined with reduction, can be implemented using techniques such as the least

significant digit (LSD) first or most significant digit (MSD) first multiplication method [20]. Implementations of reduction will be discussed in detail in Sections 5.1 and 5.2.

Division $\frac{a}{b}$, $a, b \in GF(2^m)$ is defined as a multiplication of the dividend a with the multiplicative inverse of the divisor b. Algorithms for finding the inverse element include the extended Euclidean algorithm [3] and methods employing Fermat's little theorem $a^{p-1} \equiv 1 \mod p$ for $GF(2^m)$. A method for efficiently implementing division was proposed by Chang-Shantz [7].

5.1 Reduction

Field multiplication and squaring operations require reduction by an irreducible polynomial M. Rather than computing a full polynomial division, reduction can be done by executing a sequence of polynomial multiplications and additions based on the congruency

$$u \equiv u + vM \mod M \tag{4}$$

for an irreducible polynomial M and arbitrary polynomials u and v over $GF(2)$. Reduction of a product $c_0 = a*b$, $a, b \in GF(2^m)$, $\deg(a) < m$, $\deg(b) < m$ can be computed iteratively as follows. Since the degree of c_0 is less than $2m-1$, c_0 can be split up into two polynomials $c_{0,h}$ and $c_{0,l}$ with $\deg(c_{0,h}) < m-1$, $\deg(c_{0,l}) < m$ such that

$$c_0 = a * b = c_{0,h} * t^m + c_{0,l} \tag{5}$$

Subsequent polynomials c_{j+1} can be computed iteratively by setting

$$\begin{aligned} &c_{j+1} = c_{j,h} * (M - t^m) + c_{j,l} = c_{j+1,h} * t^m + c_{j+1,l} \\ &\text{until} \quad c_{j,h} = 0 \Leftrightarrow \deg(c_j) < m \end{aligned} \tag{6}$$

Using $t^m \equiv M - t^m \, mod \, M$ as a special case of (4), it is obvious that $c_{j+1} \equiv c_0 \, mod \, M$. The reduced result $c = c_i, \deg(c) < m$ can be computed in a maximum of $i \leq m-1$ reduction iterations. The minimum number of required iterations depends on the second highest term t^k of the irreducible polynomial M [20, 14]. For

$$M = t^m + t^k + \sum_{j=1}^{k-1} M_j t^j + 1, 1 \leq k < m \tag{7}$$

it follows that $\deg(c_{j+1})$ gradually decreases such that

$$\deg(c_{j+1,h}) = \begin{cases} \text{if } \deg(c_{j,h}) > m-k: \\ \quad \deg(c_{j,h}) + k - m \\ \text{if } \deg(c_{j,h}) \leq m-k: \\ \quad 0 \end{cases} \tag{8}$$

The minimum number of iterations i is given by

$$m - 1 - i(m-k) \leq 0 \Leftrightarrow i \geq \lceil \frac{m-1}{m-k} \rceil \tag{9}$$

That is, i is the number of iterations that need to be executed if the test $c_{j,h} = 0$ is considered too costly. Choosing M such that $k \leq \frac{m+1}{2}$ apparently limits the number of reduction iterations to two. This is the case for all irreducible polynomials recommended by NIST [21] and SECG [6].

5.2 Partial Reduction

Polynomials $c \in GF(2^m)$ can be represented in reduced canonical form, i.e., $\deg(c) < m$, or in non-reduced form with $\deg(c) \geq m$. Using polynomials in both reduced and non-reduced form is the idea underlying partial reduction. For a chosen integer $n \geq m$, we define a polynomial $c \in GF(2^m)$ to be in partially-reduced representation if $\deg(c) < n$. For hardware implementations, n could, for example, be the maximum operand size of a multiplier. All computations for a point multiplication in $GF(2^m)$ can be executed on polynomials in partially-reduced representation. Reduction of the results to canonical form only needs to be done in a last step.

For a multiplication $c_0 = a * b$ with $a, b \in GF(2^m)$, $\deg(a) < n$, $\deg(b) < n$, c_0 can be partially reduced to $c \equiv c_0 \mod M$, $\deg(c) < n$ as follows: For an integer $n \geq m$, c_0 can be split up into two polynomials $c_{0,h}$ and $c_{0,l}$ with $\deg(c_{0,h}) < n-1$, $\deg(c_{0,l}) < n$. Subsequent polynomials c_{j+1} can be computed similar to (6) by setting

$$\begin{aligned} c_{j+1} &= c_{j,h} * t^{n-m} * (M - t^m) + c_{j,l} = c_{j+1,h} * t^n + c_{j+1,l} \\ \text{until} \quad & c_{j,h} = 0 \Leftrightarrow \deg(c_j) < n \end{aligned} \quad (10)$$

The result $c = c_i, \deg(c) < n$ can be computed in at most $i \leq n-1$ reduction steps. Given M as defined in (7), the minimum number of iterations i is given by

$$n - 1 - i(m-k) \leq 0 \Leftrightarrow i \geq \lceil \frac{n-1}{m-k} \rceil \quad (11)$$

A more detailed description of partial reduction can be found in [13].

6 ECC Processor Architecture

We chose a microprogrammable architecture for the cryptographic processor. The microprogram is stored in static memory and uploaded by the host at initialization time. Although the functionality of the cryptographic processor is fixed, controlling program execution by a microprogram rather than hardwired control logic provided an ideal platform for experimenting with different point multiplication algorithms.

6.1 Data Path

We decided on a bus structure for the data path to keep the design as flexible as possible. This design decision proved to be valuable as it allowed us to easily change the function units without affecting the communication infrastructure. Figure 5 shows the data path and the control unit. Dual-ported instruction and data memories called IMEM and DMEM, respectively, connect the cryptographic processor with the PCI bus of the host system. The data path is $n = 256$ bits wide, that is, the busses, the register file and memories are 256 bits wide and the function units operate on 256-bit operands.

The data memory DMEM, the registers and the function units are connected by the busses SBUS and DBUS. The data memory DMEM has a capacity of 8 kBytes and stores parameters and variables. The register file contains eight general purpose registers R0-R7, a register RM that holds the irreducible polynomial, and a register RC that specifies the field degree and the type of curve; more specifically, it specifies whether a named curve or

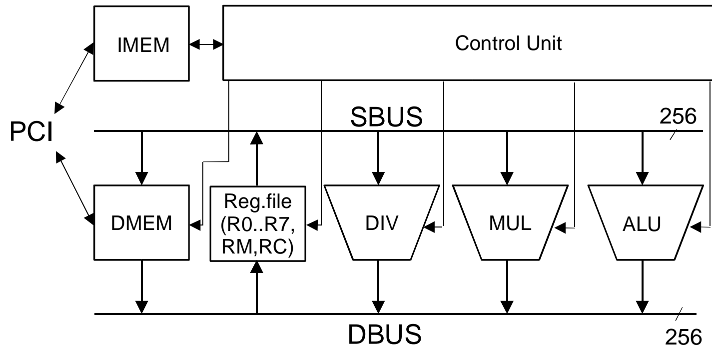

Figure 5. *Data Path and Control Unit.*

a generic curve is to be processed. The function units include a modular divider (DIV), a modular multiplier (MUL), and a multifunction arithmetic and logic unit (ALU). The ALU provides addition, modular squaring, shift, and comparison functions.

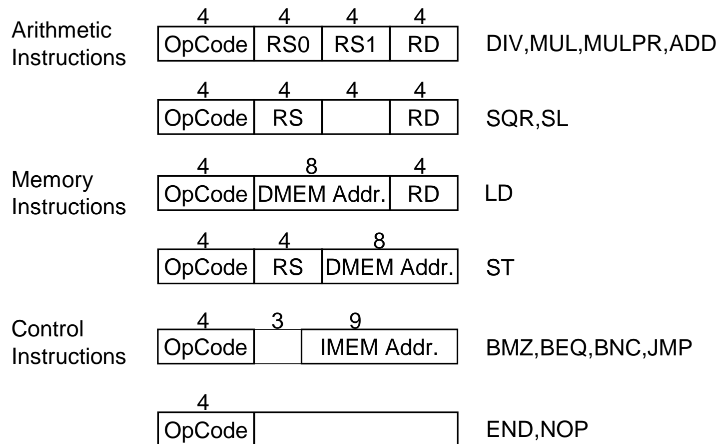

Figure 6. *Instruction Formats.*

6.2 Instruction Set

The cryptographic processor implements a load/store architecture. That is, memory can be accessed by load and store operations only, and all arithmetic instructions are limited to register operands. Instructions fall into three categories: memory instructions, arithmetic instructions, and control instructions. All instructions have a fixed length of 16 bits. Figure 6 gives the instruction formats and Table 1 contains the complete instruction set. The memory instructions include LD and ST instructions. Since memory is mainly used for passing parameters between the cryptographic processor and the host, only an absolute addressing mode is supported. The arithmetic instructions include DIV, MUL, ADD, SQR, and SL. DIV, MUL, and SQR implement modular reduction, whereby implementations differ significantly as will be shown later. The control instructions contain the three conditional branch instructions BMZ, BEQ, and BNC, the unconditional branch instruction JMP, the instruction NOP and the instruction END that terminates program execution.

Table 1. *Instruction Set.*

Opcode	Name	Semantics	Registers	Cycles
Memory Instr.				
LD DMEM,RD	Load	DMEM → RD	RD = {R0..R7,RM,RC}	3
ST RS,DMEM	Store	RS → DMEM	RS = {R0..R7}	3
Arithmetic Instr.				
DIV RS0,RS1,RD	Divide	(RS1/RS0) mod M → RD	RS0,RS1,RD = {R0..R7}	$\leq$ 2m+4 / $\leq$ 2n+4
MUL RS0,RS1,RD	Multiply	(RS0*RS1) mod M → RD	RS0,RS1,RD = {R0..R7}	7/8/10/12
ADD RS0,RS1,RD	Add	RS0+RS1 → RD (RD==0) → EQ	RS0,RS1,RD = {R0..R7}	3
SQR RS,RD	Square	(RS*RS) mod M → RD (RD==0) → EQ	RS,RD = {R0..R7}	3
SL RS,RD	Shift Left	{RS[254..0],0} → RD RS[255] → MZ (RD==0) → EQ	RS,RD = {R0..R7}	3
Control Instr.				
BMZ ADDR	Branch	branch if MZ == 0		2
BEQ ADDR	Branch	branch if EQ == 1		4
BNC ADDR	Branch	branch if NC is set		2
JMP ADDR	Jump	jump		2
NOP	No Operation	no operation		1
END	End	end program execution		

6.3 Control Unit

The control unit consists of the instruction memory IMEM that has a capacity of 1 kByte or 512 instructions and a finite state machine (FSM) that controls the data path according to the instructions fetched. The FSM uses a handshake protocol to coordinate with the function units. Handshake signals are pipelined to not delay instruction execution. This protocol allows for optimizing instruction execution times in that function units only take as many cycles as needed. Execution times vary for the execution of MUL and DIV instructions. For the multiplier, the cycle count varies with the field degree m, and for the divider, the cycle count depends on both the field degree m and the values of the operands.

Program execution times are further optimized by overlapping instruction execution and executing instructions in parallel. The control unit overlaps the execution of arithmetic instructions by prefetching the instruction as well as preloading the first source operand. This is illustrated in Figure 7. While instruction I_0 is being executed, the next instruction I_1 is prefetched, and register RS0 of I_1 is transferred over the SBUS from the register file to an arithmetic unit. Since $RS0$ of I_1 is loaded at the same time as RD of I_0 is stored, there must not be a data dependency between $RS0$ of I_1 and RD of I_0. Dependencies are detected by the assembler and are considered programming errors. Often, these dependencies can be resolved by swapping $RS0$ and $RS1$ of I_1. However, if I_0 is followed by SQR, SL, ST, or DIV, such a dependency cannot be removed as suggested and an NOP instruction needs to be inserted after I_0.

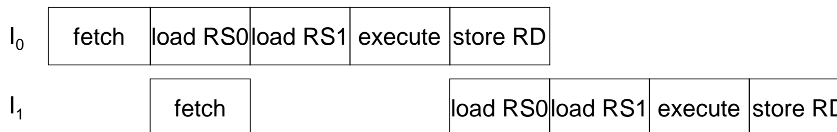

Figure 7. *Overlapped Instruction Execution.*

Parallel execution of instructions is implemented for the instruction sequence $I_0; I_1$ if I_0 is a MUL instruction and I_1 is either an ADD or SQR instruction and there are no data dependencies. The choice of these particular instructions is motivated by an analysis of the program code for point multiplication. As shown in Table 6 on page 21 the MUL instruction is the most frequently executed instruction and, in many instances, can be executed in parallel with either an ADD or an SQR instruction. Figure 8 illustrates the timing: I_1 is executed in parallel to I_0, and I_2 is prefetched while I_0 and I_1 are being executed. The following data dependencies need to be considered: I_0 and I_1 can be executed in parallel if both $RS0$ and $RS1$ of I_1 do not depend on RD of I_0; the execution of I_2 can be overlapped with the execution of I_0 and I_1 if $RS0$ of I_2 does not depend on RD of I_0.

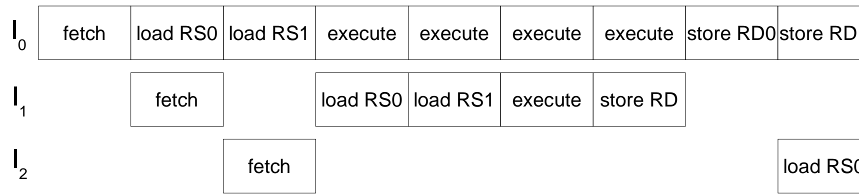

Figure 8. *Parallel Instruction Execution.*

6.4 Arithmetic and Logic Unit

The ALU shown in Figure 9 implements the two arithmetic instructions ADD and SQR and the logic instruction SL. ADD translates into a bit-wise XOR of the two source operands. SQR requires the insertion of zeroes between the bits of the source operand and the subsequent reduction of the so expanded source operand. A hardwired reduction circuit is used that can only handle named curves. SL shifts the source operand to the left by one bit. The ALU further sets the conditions codes EQ and MZ. EQ is set to 1 if the result of an ADD, SQR, or SL instruction is zero, and to 0 otherwise. MZ is set to the most significant bit shifted out of the source operand by the SL instruction.

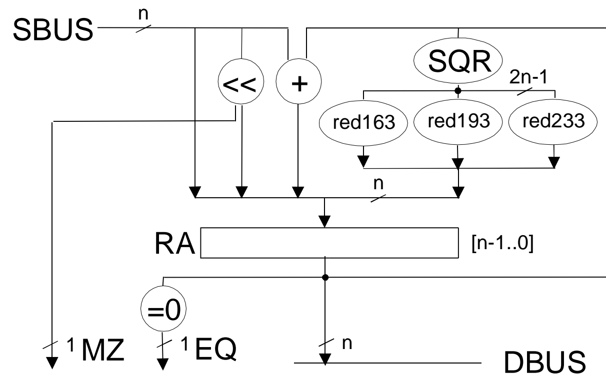

Figure 9. *ALU.*

6.5 Multiplier

The multiplier constitutes the core of the data path. As the performance analysis contained in Section 8 shows, more than half the number of cycles required to process a point multiplication are spent in the multiplier. For this reason, we optimized its performance as much as possible and spent a significant part of the chip resources on it.

We have implemented a number of digit-serial modular multiplier designs based on algorithms described by Song and Parhi in [20]. Our first design described in [12] used an LSD first multiplier that could perform modular multiplication in hardware for named curves only. In addition, the multiplier could generate an unreduced product so that reduction for generic curves could be performed by microcode. Generic curves were implemented in microcode and processed at a tenth of the throughput achieved for named curves. Here, we describe a novel multiplier design that performs modular multiplication for both types of curves in hardware thereby significantly improving the performance for generic curves. The new design is based on a MSD first multiplier. We also considered the LSD first multiplier but, as we will explain later, found that pipelining for the MSD multiplier can be done more efficiently.

We will first describe an MSD first multiplier that works for named curves only, before we describe the final design that can handle both named and generic curves. The pseudo code looks as follows:

```
X[n-1..0] := x*t^(d*⌊(n-m)/d⌋); Y[n-1..0] := y*t^(d*⌊(n-m)/d⌋);
P[n+d-1..0] := 0; Z[n-1..0] := 0;
for i := 0 to ⌈m/d⌉-1 do
  P[n+d-1..0] := X[n-1..n-d] * Y[n-1..0];
  X[n-1..0] := shift_left(X[n-d-1..0],d);
  Z[n-1..0] := (shift_left(Z[n-1..0],d) + P[n+d-1..0]) mod M*t^(d*⌊(n-m)/d⌋);
end;
```

Figure 10 shows a block diagram of an MSD first multiplier for named curves of field degrees 163, 193, and 233. The following three computation steps are performed in parallel: (i) the MSD of X is multiplied with Y; (ii) X is shifted to the left by d bits; (iii) Z is shifted to the left by d bits, added to P, and subsequently reduced. It takes again $\lceil m/d \rceil + 1$ clock cycles to perform the modular multiplication, that is, the number of multiplication steps executed depends on m. This optimization requires that the registers X and Y are loaded with the operands shifted to the left by $d * \lfloor (n-m)/d \rfloor$ bits. In our implementation, we only support a shift by d bits. That is, for $n = 256$ and $d = 64$, the modular multiplication takes five clock cycles for $m > 192$ and four clock cycles for $m \leq 192$.

We further developed a generic MSD first multiplier shown in Figure 11 that can handle both named and generic curves. It uses a hardwired reducer for named curves and it reuses the multiplier circuit to perform reduction for generic curves. Reduction for generic curves is based on the partial reduction algorithm explained in Section 5.2. This technique reduces to the data path width n rather than to the field size m, with $m \leq n$. This avoids costly shift and mask operations to extract field-sized operands smaller than the data path width. In comparison with Montgomery modular multiplication, our scheme uses fewer multiplications; this is particularly true when a large multiplier is used. The pseudo code

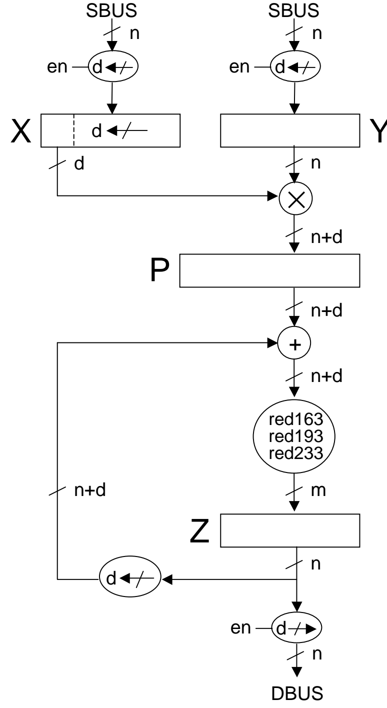

Figure 10. *MSD First Multiplier for Named Curves.*

for performing modular multiplication on generic curves looks as follows:

```
X[n-1..0] := x*t^{d*⌊(n-m)/d⌋}; Y[n-1..0] := y*t^{d*⌊(n-m)/d⌋};
P[n+d-1..0] := 0; Z[n-1..0] := 0;
for i := 0 to ⌈m/d⌉-1 do
  P[n+d-1..0] := X[n-1..n-d] * Y[n-1..0];
  X[n-1..0] := shift_left(X[n-1..0],d);
  r[n+d-1..0] := shift_left(Z[n-1..0]) + P[n+d-1..0];
  Z[n-1..0] := r[n-1..0] + r[n+d-1..n] * (M - t^m) * t^{n-m};
end;
```

There is one partial product generator $\bigotimes$ that is alternately used to perform a multiplication step and a reduction step. Rather than strictly interleaving these two steps, the computation begins with executing two multiplication steps before the first reduction step is executed. That is, P and Z are computed in the order $\{P_0, P_1, Z_0, P_2, Z_1, ..\}$ such that P_i is only needed two cycles later when Z_{i+1} is calculated.

Table 2 shows the state diagram for the generic MSD first multiplier of Figure 11. Separate control flows are given for named and generic curves. The state diagram for named curves looks as follows: The source operands are loaded from the SBUS in states $S0$ and $S1$; the partial products are computed in states $S2$, $S3$, $S4$ and $S5$ - $S3$, $S4$ and $S5$ also accumulate and reduce the partial results; $S6$ performs a final accumulation and reduction; finally, the result is transferred over the DBUS into the register file in state $S7$ (not shown). The shown states are executed for curves with field degree $192 < m \leq 255$. For $m \leq 192$,

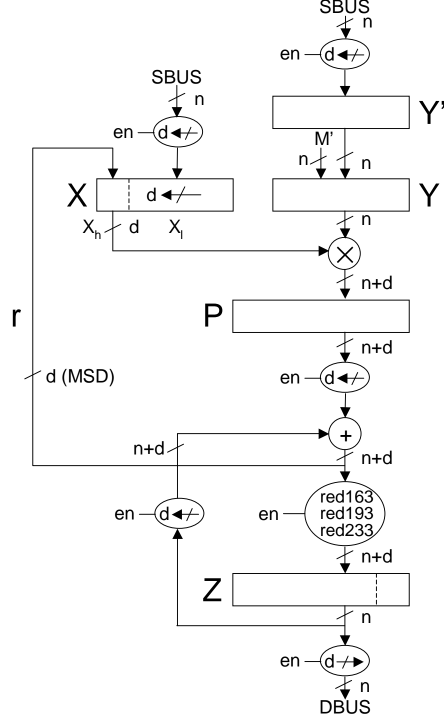

Figure 11. *Generic MSD First Multiplier for Generic and Named Curves.*

state $S4$ is skipped. Looking at generic curves, the state diagram is specified as follows: The source operands are loaded from the SBUS in states $S0$ and $S1$; the partial products are computed in states $S2$, $S3$, $S5$ and $S7$; the reduction of the accumulated multiplication results happens in states $S4$, $S6$, $S8$ and $S9$; $S10$ performs a final accumulation and reduction; finally, the result is transferred over the DBUS into the register file in state $S11$ (not shown). Since the multiplier is alternately used for a multiplication step and a reduction step, register X alternately supplies the MSD of x and the MSD of the accumulated result, and register Y alternately supplies y and M' where $M' = (M - t^m) * t^{n-m}$. The state machine for generic curves is again optimized such that states are skipped for smaller field degrees: States $S5$ and $S6$ are skipped for $m \leq 192$.

We also implemented a similar multiplier using the LSD first method. Comparing the two implementations we found that the MSD multiplier can be pipelined more efficiently saving one state for generic curves. The reason is that dependencies between partial products and reduction results are less stringent for the MSD first multiplier.

Table 3 gives the cycle counts for the generic MSD first multiplier. The cycle counts include the time needed to load and store the operands. It takes seven cycles to perform a modular multiplication for named curves over fields $GF(2^m), m \leq 192$, eight cycles for named curves over fields $GF(2^m), 193 \leq m \leq 255$, 10 cycles for generic curves over fields $GF(2^m), m \leq 192$, and 12 cycles for generic curves over fields $GF(2^m), 193 \leq m \leq 255$.

Note that we have assumed that it takes a single multiplication to execute a reduction step, which poses some restrictions on the irreducible polynomial. When reducing $r[n+d-1..0] = shift_left(Z[n-1..0]) + P[n+d-1..0]$, we assume that the partially reduced result of the multiplication $r[n-1..0] + r[n+d-1..n] * ((M - t^m) * t^{n-m})$ can be stored in an

Table 2. *State Diagram of the Generic MSD First Multiplier.*

State	X	Y	Y'	P	Z
Named Curves					
S0	-	-	y	-	-
S1	x	Y'	Y'	-	-
S2	$shift_left(X)$	Y'	Y'	$X_h * Y$	0
S3	$shift_left(X)$	Y'	Y'	$X_h * Y$	$(P + shift_left(Z))modM$
S4	$shift_left(X)$	Y'	-	$X_h * Y$	$(P + shift_left(Z))modM$
S5	-	-	-	$X_h * Y$	$(P + shift_left(Z))modM$
S6	-	-	-	-	$(P + shift_left(Z))modM$
Generic Curves					
S0	-	-	y	-	-
S1	x	Y'	Y'	-	-
S2	$shift_left(X)$	Y'	Y'	$X_h * Y$	0
S3	$\{(P + shift_left(Z))_h, X_l\}$	M'	Y'	$X_h * Y$	$P + shift_left(Z)$
S4	$shift_left(X)$	Y'	Y'	$X_h * Y$	$P + shift_left(Z)$
S5	$\{(shift_left(P) + Z)_h, X_l\}$	M'	Y'	$X_h * Y$	$shift_left(P) + Z$
S6	$shift_left(X)$	Y'	Y'	$X_h * Y$	$P + shift_left(Z)$
S7	$\{(shift_left(P) + Z)_h, X_l\}$	M'	Y'	$X_h * Y$	$shift_left(P) + Z$
S8	$\{(P + shift_left(Z))_h, X_l\}$	M'	Y'	$X_h * Y$	$P + shift_left(Z)$
S9	-	-	-	$X_h * Y$	$shift_left(P) + Z$
S10	-	-	-	-	$shift_left(P) + shift_left(Z)$

n-bit register. This requirement is equivalent to the partial reduction being executable in a single iteration. Employing Equations (10) and (8) and given a partial product generator that multiplies $d \times n$ bits, the number of reduction iterations i is

$$d - i(m - k) \leqslant 0 \Leftrightarrow i \geqslant \lceil \frac{d}{m - k} \rceil$$

For limiting partial reduction to a single iteration it follows that $d \leq m - k$. For $d = 64$ this limits irreducible polynomials P to those with $m - k \geq 64$. All polynomials recommended by NIST and SECG satisfy this condition. Polynomials with $m - k < 64$ could be accommodated by allowing for multiple reduction iterations. This, however, would significantly reduce the performance of the multiplier.

Table 3. *Cycle Counts for the Generic MSD First Multiplier.*

	Named Curves	Generic Curves
$m > 192$	8	12
$m \leq 192$	7	10

6.6 Divider

The cryptographic processor implements a modular divider based on an algorithm described by Chang-Shantz [7] that has similarities to Euclid's greatest common divisor (GCD) algorithm. The divider consists of four 256-bit registers - A, B, U and V - and a fifth register holding the irreducible polynomial M. It can compute division for arbitrary irreducible polynomials M and field degrees up to $m = 255$.

Initially, A is loaded with the divisor X, B with the irreducible polynomial M, U with the dividend Y, and V with 0. Throughout the division, the following invariants are

maintained:

$$A * Y \equiv U * X \, mod \, M \tag{12}$$

$$B * Y \equiv V * X \, mod \, M \tag{13}$$

Through repeated additions and divisions by t, A and B are gradually reduced to 1 such that U (respectively V) contains the quotient $\frac{Y}{X} \, mod \, M$. One should note that a polynomial is divisible by t if it is even, i.e., the least significant bit of the corresponding bit string is 0. Division by t can be efficiently implemented as a shift right operation. We use two counters CA and CB to test for termination of the algorithm. For named curves, CB is initialized with the field degree m and CA with $m-1$. And for generic curves, CB is initialized with the register size n and CA with $n-1$. CA and CB represent the upper bound for the order of A and B. This is due to the fact that the order of $A+B$ is never greater than the order of A if $CA > CB$ and never greater than the order of B if $CA \leq CB$. The following pseudo code describes the operation of the divider:

```
A:=X; B:=M; U:=Y; V:=0;
if named_curve then {CA:=m-1; CB:=m}
               else {CA:=n-1; CB:=n};
while (even(A) and CA>=0) do {
  A:=shiftr(A); CA:=CA-1;
  if even(U) then U:=shiftr(U)
             else U:=shiftr(U+M);}
while (CA>=0 and CB>=0) do {
  if (CA>CB) then {
    A:=A+B; U:=U+V;
    while (even(A) and CA>=0) do {
      A:=shiftr(A); CA:=CA-1;
      if even(U) then U:=shiftr(U)
                 else U:=shiftr(U+M);}
    }
    else {
    B=A+B; V:=U+V;
    while (even(B) and CB>=0) do {
      B:=shiftr(B); CB:=CB-1;
      if even(V) then V:=shiftr(V)
                 else V:=shiftr(V+M);}
    }
  }
if (CA<0) then return V
          else return U;
```

A modular division can be computed in a maximum of $2m$ clock cycles for named curves and in a maximum of $2n$ clock cycles for generic curves. The corresponding block diagram of the hardware implementation of the divider is shown in Figure 12.

Note that the divider fully reduces the result to the field degree. In particular, divisions by 1 can be used to reduce a polynomial of degree less than n to a polynomial of degree less than m.

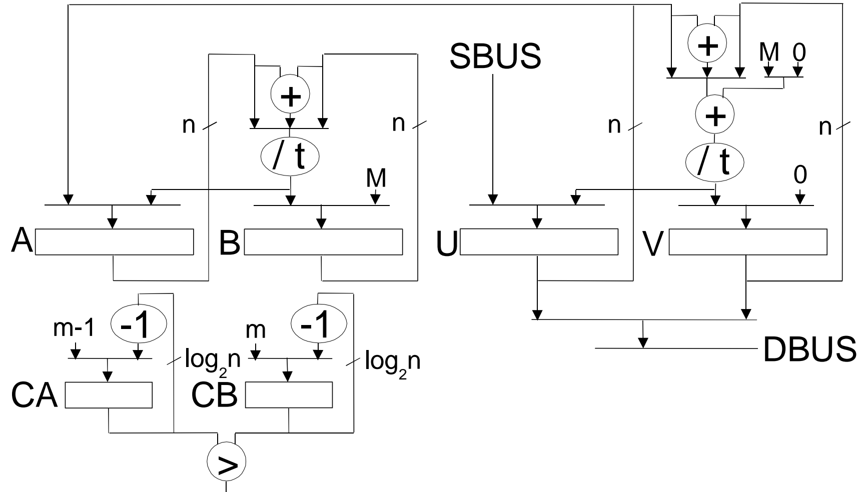

Figure 12. *Divider.*

6.7 Implementation

We prototyped the cryptographic processor in a Xilinx Virtex-II XCV2000E-7 FPGA. The floorplan of the synthesized design is shown in Figure 13. Area constraints were provided for the ALU, the divider and the register file, whereas the multiplier was left unconstrained. This way, these blocks do not interfere with each other when resources are allocated, while at the same time as many resources as needed can be allocated to the multiplier which constitutes the critical path. No other constraints and, in particular, no manual placement was required to obtain a synthesized design that runs at the targeted frequency of 66.4 MHz which is derived from the PCI clock.

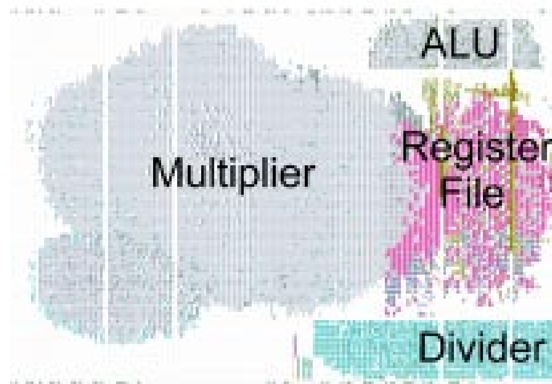

Figure 13. *FPGA Floorplan.*

Table 4 quantifies the resources used by the function units. The listed resources are 4-input look-up tables (LUTs) and flip-flops (FFs). The multiplier clearly dominates the

size of the design as it uses 74% of the LUTs and 47% of the FFs. Since multiplication is the single most time-critical operation, its dominance, nevertheless, seems justified.

Table 4. *Usage of Chip Resources.*

Unit	LUTs	FFs
Generic MSD first Multiplier	14797	2948
ALU	1345	279
Divider	2678	1316
Full Design	20068	6321

7 Point Multiplication Code

We have implemented Montgomery's point multiplication algorithm that we outlined in Section 4. A fragment of the assembly code implementing Equations (1) and (2) is shown in Table 5. The computation of the four equations is interleaved to achieve a higher degree of instruction-level parallelism. We use a single code base for named curves and generic curves. This is accomplished by executing MUL and SQR instructions according to the curve type. For named curves, MUL denotes a multiplication with hardwired reduction and, for generic curves, it is executed as a multiplication with partial reduction. The execution of an SQR instruction is slightly more complicated. For named curves, SQR is executed by the ALU. And for generic curves, the SQR instruction is translated into a MUL instruction that is executed as a multiplication with partial reduction. We use the BNC (branch if named curve) instruction in the few places where the program code differs for the two curve types. BNC tests the NC flag which is initialized by the host before a point multiplication computation is started.

As we had explained in Section 6.3, we make use of the fact that the multiplier and the ALU can operate in parallel. That is, if there are no data dependencies, the MUL instruction can be executed in parallel with either an ADD or an SQR instruction. Since the SQR instruction is executed by the ALU for named curves and by the multiplier for generic curves, the order in which instructions are executed differs depending on the curve type even though the code is the same.

Data dependencies are detected in different ways. The assembler checks for dependencies that would prevent overlapped instruction execution. In these cases, the programmer needs to resolve the dependencies by reordering operands or inserting NOP instructions. With respect to parallel instruction execution, the control unit examines dependencies and decides whether instructions can be executed in parallel or not.

The code fragment in Table 5 shows no data dependencies for any MUL/SQR or MUL/ADD instruction sequence. Hence, for named curves, all MUL/SQR and MUL/ADD sequences are executed in parallel.

Furthermore, since there are no data dependencies between subsequent arithmetic instructions, instruction execution can be overlapped, thus, saving one cycle per instruction.

Code execution looks different for generic curves as illustrated. In this case, all MUL/SQR sequences have to be executed sequentially as SQR instructions are now executed as MUL instructions. However, there still is one SQR/ADD sequence and one MUL/ADD sequence left that can be executed in parallel.

Table 5. *Assembly Code for Projective Point Doubling and Point Addition and its Execution for Named and Generic Curves.*

Instructions		Execution for Named Curves	Execution for Generic Curves
	$R0 = X1, R1 = Z_1, R2 = X_2, R3 = Z_2$		
MUL(R1,R2,R2)	$R2 = Z_1 * X_2$	MUL(R1,R2,R2);SQR(R1,R1)	MUL(R1,R2,R2)
SQR(R1,R1)	$R1 = Z_1^2$		SQR(R1,R1)
MUL(R0,R3,R4)	$R4 = X_1 * Z_2$	MUL(R0,R3,R4);SQR(R0,R0)	MUL(R0,R3,R4)
SQR(R0,R0)	$R0 = X_1^2$		SQR(R0,R0);ADD(R2,R4,R3)
ADD(R2,R4,R3)	$R3 = Z_1 * X_2 + X_1 * Z_2$	ADD(R2,R4,R3)	
MUL(R2,R4,R2)	$R2 = Z_1 * X_2 * X_1 * Z_2$	MUL(R2,R4,R2);SQR(R1,R4)	MUL(R2,R4,R2)
SQR(R1,R4)	$R4 = Z_1^4$		SQR(R1,R4)
MUL(R0,R1,R1)	$R1 = Z_1^2 * X_1^2$	MUL(R0,R1,R1);SQR(R3,R3)	MUL(R0,R1,R1)
SQR(R3,R3)	$R3 = Z_3 = (Z_1 * X_2 + X_1 * Z_2)^2$		SQR(R3,R3)
LD(dmem_b,R5)	$R5 = b$	LD(dmem_b,R5)	LD(dmem_b,R5)
MUL(R4,R5,R4)	$R4 = b * Z_1^4$	MUL(R4,R5,R4);SQR(R0,R0)	MUL(R4,R5,R4)
SQR(R0,R0)	$R0 = X_1^4$		SQR(R0,R0)
LD(dmem_Px,R5)	$R5 = X$	LD(dmem_Px,R5)	LD(dmem_Px,R5)
MUL(R3,R5,R5)	$R4 = X * (Z_1 * X_2 + X_1 * Z_2)^2$	MUL(R3,R5,R5);ADD(R4,R0,R0)	MUL(R3,R5,R5);ADD(R4,R0,R0)
ADD(R4,R0,R0)	$R0 = X_1^4 + b * Z_1^4$		
ADD(R2,R5,R2)	$R2 = X * Z_3 + (Z_1 * X_2) * (X_1 * Z_2)$	ADD(R2,R5,R2)	ADD(R2,R5,R2)

8 Evaluation

This section contains two parts. First, we look at the distribution of instructions executed by a point multiplication. Next, we compare performance numbers for point multiplication executed in hardware and software.

8.1 Instruction Distribution

Table 6 gives the distribution of instructions executed by the point multiplication operation for named and generic curves, respectively. For point multiplication on named curves over $GF(2^{163})$, field multiplications account for almost 62% of the execution time. In the case of generic curves, field multiplications even constitute 81% of the execution time. It is, therefore, justified to allocate a significant portion of the available hardware resources to the multiplier. Parallel and overlapped execution save 36% of the execution time for named curves and 20% for generic curves when compared to sequential execution. There is still room for further improvements since the control flow instructions BMZ, BEQ, SL, JMP and END consume almost 21% of the execution time when processing named curves and 10% when processing generic curves. This time could be saved by separating control flow and data flow.

To evaluate the performance of the divider, we implemented an inversion algorithm that replaces the division with a sequence of multiplications and squarings based on Fermat's little theorem. A hard-coded field inversion for named curves over binary polynomials fields $GF(2^{163})$ took 938 cycles (0.01413 ms). Thus, the divider is almost three times as fast as a soft-coded implementation. The speedup is even higher when generic curves are used. Also note that the divider provides a reduced canonical result of less than the field degree m, which is a useful property when dealing with generic curves.

Table 6. *Decomposition of the Execution Time for $GF(2^{163})$ Point Multiplication.*

	Named Curves			Generic Curves		
Instruction	**#Instr.**	**Cycles**	**ms**	**#Instr.**	**Cycles**	**ms**
MUL	6	41	0.00062	818	7366	0.11093
MUL + ADD	166	996	0.01500	327	2943	0.04432
MUL + SQR	811	4866	0.07328	0	0	0.00000
SQR	2	4	0.00006	651	5859	0.08824
DIV	1	329	0.00495	2	902	0.01358
ADD	330	660	0.00994	170	340	0.00512
SL	326	978	0.01473	326	978	0.01473
ST	6	18	0.00027	8	24	0.00036
LD	334	668	0.01006	337	674	0.01015
BMZ	326	652	0.00982	326	652	0.00982
BEQ	1	4	0.00006	1	4	0.00006
BNC	2	4	0.00006	2	4	0.00006
JMP	162	324	0.00488	162	324	0.00488
END	1	2	0.00003	1	2	0.00003
total		9549	0.14381		20072	0.30229

8.2 Point Multiplication Performance

Table 7 shows performance numbers for implementations of point multiplication in hardware and software. The hardware implementation uses the prototype system described in Section 6.7. The software implementation considers generic curves and does not contain any curve-specific optimizations. It is executed on a 900 MHz Sun Fire™280R server. The execution time of a point multiplication kP depends on the execution times of the arithmetic operations in $GF(2^m)$ and the size of the integer k, which is mostly in the order of the field degree m. The time needed for a point multiplication on the cryptographic processor grows almost linearly with the size of k between $1 \leq m \leq 192$ and $193 \leq m \leq 256$. The non-linear increase at $m = 192$ is caused by the multiplier that exhibits different execution times based on the field degree.

Hardwired reduction improves the execution time for named curves by more than a factor of two in comparison to generic curves. The speedup of the hardware implementation over the software implementation is a factor of about 20 for named curves which is significant considering that the FPGA prototype runs at less than $\frac{1}{13}$ the 900 MHz clock frequency. The poor software performance is mainly due to the lack of support for $GF(2^m)$ arithmetic in general-purpose CPUs. While the software implementation is optimized for irreducible polynomials that are either pentanomials or trinomials, the hardware implementation is more generic in that it can operate on arbitrary irreducible polynomials.

Table 7. *Hardware and Software Performance.*

	Hardware		Software		Speedup
	ops/s	**ms/op**	**ops/s**	**ms/op**	
Named Curves					
$GF(2^{163})$	6955	0.14	322	3.11	21.6
$GF(2^{193})$	5333	0.19	294	3.40	18.1
$GF(2^{233})$	4423	0.23	223	4.48	19.8
Generic Curves					
$GF(2^{163})$	3308	0.30			
$GF(2^{193})$	2375	0.42			
$GF(2^{233})$	1980	0.51			

9 Conclusions

We presented a cryptographic processor that provides optimized performance for a number of named curves and support for generic curves over arbitrary fields $GF(2^m), m \leq 255$. This flexibility is needed by server applications that have to perform large numbers of point multiplications on different curves.

We described two novel elements of a cryptographic processor: A modular divider and a modular multiplier both capable of handling arbitrary polynomial fields $GF(2^m)$. The divider leads to a performance gain of a factor of three over a soft-coded implementation based on Fermat's little theorem. The divider, furthermore, comes in handy when the intermediate results computed by the partial reduction algorithm have to be reduced to the actual field degree.

We described a novel modular multiplier that is capable of handling named curves as well as generic curves. Processing the two types of curves differs in that hardwired reduction logic is used for named curves and the multiplier logic is reused to perform reduction for generic curves. Since reduction for generic curves reuses existing logic, the additional resources needed to support generic curves are minimal.

Our cryptographic processor uses a common code base to implement point multiplication for both named curves and generic curves. To make this possible, squaring instructions are dynamically translated into multiplication instructions in the case of generic curves - this translation is necessary since modular squaring is implemented in hardware for named curves only.

We increased performance by exploiting the parallelism found in Montgomery's point multiplication algorithm. More specifically, we allow multiplication instructions, which are the most frequently executed instructions, to be executed in parallel with either add or square instructions. Together with overlapped instruction execution, parallel execution reduces the execution time for a point multiplication operation by 36% for named curves and 20% for generic curves.

We are currently working on a cryptographic processor that uses a common architecture capable of executing point multiplication for both $GF(p)$ and $GF(2^m)$.

Acknowledgments

Daniel Finchelstein helped with the architecture and the implementation of the initial design. Jo Ebergen suggested a divider design that uses counters rather than comparators. Edouard Goupy implemented the inversion algorithm. Sumit Gupta helped with the integration of the ECC cipher suites into OpenSSL.

References

[1] G. B. Agnew, R. C. Mullin, and S. A. Vanstone. An implementation of elliptic curve cryptosystems over $\mathbb{F}_{2^{155}}$. *In IEEE Journal on Selected Areas in Communications*, 11(5):804–813, June 1993.

[2] M. Bednara, M. Daldrup, J. von zur Gathen, and J. Shokrollahi. Reconfigurable implementation of elliptic curve crypto algorithms. *Reconfigurable Architectures Workshop, 16th International Parallel and Distributed Processing Symposium*, April 2002.

[3] Berlekamp. *Algebraic Coding Theory.* Aegan Park Press, 1984.

[4] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography.* Cambridge University Press, 1999. London Mathematical Society Lecture Note Series 265.

[5] S. Blake-Wilson, T. Dierks, and C. Hawk. *ECC Cipher Suites for TLS.* IETF Internet Draft, March 2001. http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-01.txt.

[6] Certicom Research. Sec 2: Recommended elliptic curve domain parameters. Standards for Efficient Cryptography Version 1.0, September 2000.

[7] S. Chang-Shantz. From euclid's gcd to montgomery multiplication to the great divide. Technical report, Sun Microsystems Laboratories TR-2001-95, June 2001. http://research.sun.com/.

[8] T. Dierks and C. Allen. The tls protocol - version 1.0. ietf rfc 2246. January 1999.

[9] J. Goodman and A. P. Chandrakasan. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, 36(11):1808–1820, November 2001.

[10] V. Gupta, S. Blake-Wilson, B. Möller, and C. Hawk. *ECC Cipher Suites for TLS.* IETF Internet Draft, August 2002. http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-02.txt.

[11] V. Gupta, S. Gupta, S. Chang, and D. Stebila. Performance analysis of elliptic curve cryptography for ssl. In *ACM Workshop on Wireless Security*, September 2002. Atlanta, Georgia.

[12] N. Gura, H. Eberle, and S. Chang-Shantz. An end-to-end systems approach to elliptic curve cryptography. In *CHES '2002 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science. Springer-Verlag, August 2002. Redwood City, California.

[13] N. Gura, H. Eberle, and S. Chang-Shantz. Generic implementations of elliptic curve cryptography using partial reduction. In *9th ACM Conference on Computers and Communications Security*, November 2002. Washington, DC.

[14] A. Halbutoğulları and Ç. K. Koç. Mastrovito multiplier for general irreducible polynomials. *IEEE Transactions on Computers*, 49(5):503–518, May 2000.

[15] D. Hankerson, J. L. Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1965. Springer-Verlag, August 2000.

[16] A. Karatsuba and Y. Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akad. Nauk*, (145):293–294, 1963. Translation in Physics-Doklady 7, 595-596.

[17] J. López and R. Dahab. Fast multiplication on elliptic curves over $\mathbb{GF}(2^m)$ without precomputation. In *CHES '99 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1717. Springer-Verlag, August 1999.

[18] OpenSSL Project. http://www.openssl.org/.

[19] G. Orlando and C. Paar. A high-performance reconfigurable elliptic curve processor for $\mathbb{GF}(2^m)$. In *CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science 1965. Springer-Verlag, August 2000.

[20] L. Song and K. K. Parhi. Low-energy digit-serial/parallel finite field multipliers. *IEEE Journal of VLSI Signal Processing Systems*, (19):149–166, 1998.

[21] U.S. Department of Commerce and National Institute of Standards and Technology. Digital signature standard (dss). Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[22] A. D. Woodbury, D. V. Bailey, and C. Paar. Elliptic curve cryptography on smart cards without coprocessors. In *The Fourth Smart Card Research and Advanced Applications (CARDIS2000) Conference*, September 2000. Bristol, UK.

About the Authors

Hans Eberle is a Senior Staff Engineer at Sun Microsystems Laboratories in Mountain View, California, where he works in the areas of networks and security. From 1993 until 1998 he was an Assistant Professor of Computer Science at ETH Zurich, Switzerland, teaching classes for EE and CS students and conducting research on switched system area networks. From 1988 until 1993 he was a Principal Engineer at the Systems Research Center of the Digital Equipment Corporation in Palo Alto, California, working on multiprocessors, ATM networks, and high-speed DES encryption. He received a diploma in Electrical Engineering and Ph.D. in Technical Sciences from ETH Zurich in 1984 and 1987, respectively.

Nils Gura joined Sun Microsystems Laboratories, California, in January 2001 and has been working on cluster interconnect technologies and network scheduling algorithms. His current interest lies in efficient algorithms and hardware acceleration for next generation cryptographic technologies, primarily Elliptic Curve Cryptography. Nils holds a M.Sc. in Computer Science from the University of Ulm, Germany.

Sheueling Chang Shantz is a Distinguished Engineer in Sun Microsystems Laboratories. She joined Sun in 1984 with a Master degree and a Ph.D. degree in Computer Science from California Institute of Technology. She also received a Master in Business Management from Stanford Business School in 1998. Sheueling's background includes network security, RSA and Elliptic Curve public key cryptography (ECC), electronic commerce, and Internet banking and payment systems.

Her current interest is in the area of efficient algorithms for implementing Elliptic Curve Cryptosystems, and the development of hardware cryptographic accelerators for RSA and ECC.

Vipul Gupta is a Senior Staff Engineer at Sun Microsystems Laboratories, where his research interests include secure networking protocols and mobile computing. Prior to joining Sun, he was an Assistant Professor at the State University of New York where he taught courses in computer networking, parallel processing, and operating systems and conducted research funded by the National Science Foundation and industry sponsors that included IBM and NEC. He has a Ph.D. in Computer Science from Rutgers University.