An End-to-End Systems Approach to Elliptic Curve Cryptography

Nils Gura, Sheueling Chang Shantz, Hans Eberle, Sumit Gupta, Vipul Gupta,
Daniel Finchelstein, Edouard Goupy, Douglas Stebila

Sun Microsystems Laboratories
{Nils.Gura, Sheueling.Chang, Hans.Eberle, Gupta.Sumit, Vipul.Gupta,
Daniel.F.Finchelstein, Edouard.Goupy, Douglas.Stebila}@sun.com
http://www.research.sun.com

Abstract. Since its proposal by Victor Miller [17] and Neal Koblitz [15] in the mid 1980s, Elliptic Curve Cryptography (ECC) has evolved into a mature public-key cryptosystem. Offering the smallest key size and the highest strength per bit, its computational efficiency can benefit both client devices and server machines. We have designed a programmable hardware accelerator to speed up point multiplication for elliptic curves over binary polynomial fields $GF(2^m)$. The accelerator is based on a scalable architecture capable of handling curves of arbitrary field degrees up to $m = 255$. In addition, it delivers optimized performance for a set of commonly used curves through hard-wired reduction logic. A prototype implementation running in a Xilinx XCV2000E FPGA at 66.4 MHz shows a performance of 6987 point multiplications per second for $GF(2^{163})$. We have integrated ECC into OpenSSL, today's dominant implementation of the secure Internet protocol SSL, and tested it with the Apache web server and open-source web browsers.

1 Introduction

Since its proposal by Victor Miller [17] and Neal Koblitz [15] in the mid 1980s, Elliptic Curve Cryptography (ECC) has evolved into a mature public-key cryptosystem. Extensive research has been done on the underlying math, its security strength, and efficient implementations.

ECC offers the smallest key size and the highest strength per bit of any known public-key cryptosystem. This stems from the discrete logarithm problem in the group of points over an elliptic curve. Among the different fields that can underlie elliptic curves, integer fields $F(p)$ and binary polynomial fields $GF(2^m)$ have shown to be best suited for cryptographical applications. In particular, binary polynomial fields allow for fast computation in both software and hardware implementations.

Small key sizes and computational efficiency of both public- and private-key operations make ECC not only applicable to hosts executing secure protocols over wired networks, but also to small wireless devices such as cell phones, PDAs and SmartCards. To make ECC commercially viable, its integration into secure

protocols needs to be standardized. As an emerging alternative to RSA, the US government has adopted ECC for the Elliptic Curve Digital Signature Algorithm (ECDSA) and specified *named curves* for key sizes of 163, 233, 283, 409 and 571 bit [18]. Additional curves for commercial use were recommended by the Standards for Efficient Cryptography Group (SECG) [7]. However, only few ECC-enabled protocols have been deployed in commercial applications to date. Today's dominant secure Internet protocols such as SSL and IPsec rely on RSA and the Diffie-Hellman key exchange. Although standards for the integration of ECC have been proposed [4], they have not yet been finalized.

Our approach towards an end-to-end solution is driven by a scenario of a wireless and web-based environment where millions of client devices connect to a secure web server.

The aggregation of client-initiated connections/transactions leads to high computational demand on the server side, which is best handled by a hardware solution. While support for a limited number of curves is acceptable for client devices, server-side hardware needs to be able to operate on numerous curves. The reason is that clients may choose different key sizes and curves depending on vendor preferences, individual security requirements and processor capabilities. In addition, different types of transactions may require different security levels and thus, different key sizes.

We have developed a cryptographic hardware accelerator for elliptic curves over arbitrary binary polynomial fields $GF(2^m)$, $m \leq 255$. To support secure web transactions, we have fully integrated ECC into OpenSSL and tested it with the Apache web server and open source web browsers.

The paper is structured as follows: Section 2 summarizes related work and implementations of ECC. In Section 3, we outline the components of an ECC-enabled secure system. Section 4 describes the integration of ECC into OpenSSL. The architecture of the hardware accelerator and the implemented algorithms are presented in Section 5. We give implementation cost and performance numbers in Section 6. The conclusions and future directions are contained in Section 7.

2 Related Work

Hardware implementations of ECC have been reported in [20], [2], [1], [11], [10] and [9]. Orlando and Paar describe a programmable elliptic curve processor for reconfigurable logic in [20]. The prototype performs point multiplication based on Montgomery Scalar Multiplication in projective space [16] for $GF(2^{167})$. Their design uses polynomial basis coordinate representation. Multiplication is performed by a digit-serial multiplier proposed by Song and Parhi [22]. Field inversion is computed through Fermat's theorem as suggested by Itoh and Tsujii [13]. With a performance of 0.21 ms per point multiplication this is the fastest reported hardware implementation of ECC. Bednara et al. [2] designed an FPGA-based ECC processor architecture that allows for using multiple squarers, adders and multipliers in the data path. They researched hybrid coordinate representations in affine, projective, Jacobian and López-Dahab form.

Two prototypes were synthesized for $GF(2^{191})$ using an LFSR polynomial basis multiplier and a Massey-Omura normal basis multiplier, respectively. Agnew et al. [1] built an ECC ASIC for $GF(2^{155})$. The chip uses an optimal normal basis multiplier exploiting the composite field property of $GF(2^{155})$. Goodman and Chandrakasan [11] designed a generic public-key processor optimized for low power consumption that executes modular operations on different integer and binary polynomial fields. To our knowledge, this is the only implementation that supports $GF(2^m)$ for variable field degrees m. However, the architecture is based on bit-serial processing and its performance cannot be scaled to levels required by server-type applications.

3 System Overview

Figure 1 shows the implementation of a client/server system using a secure ECC-enhanced protocol. We integrated new cipher suites based on ECC into OpenSSL [19], 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. To accelerate public-key operations on the server side, we designed and built a hardware accelerator connected to the host machine through a PCI interface. The accelerator is accessed by a character device driver running under the Solaris™Operating Environment.
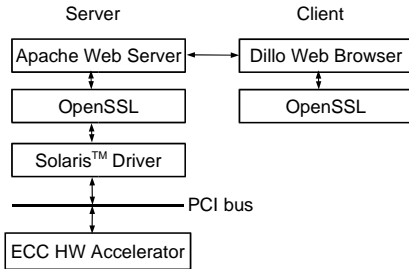

Fig. 1. *Secure Client/Server System.*

4 Secure Sockets Layer

Secure Sockets Layer (SSL aka TLS) [8] is the most widely deployed and used security protocol on the Internet today. The protocol has withstood years of scrutiny by the security community and, in the form of HTTPS[1], is now trusted to secure virtually all sensitive web-based applications ranging from banking to online trading to e-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 TCP and has the flexibility to accommodate different cryptographic algorithms for key agreement, encryption and hashing. However, the

---

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

specification does recommend particular combinations of these algorithms, called *cipher suites*, which have well-understood security properties.
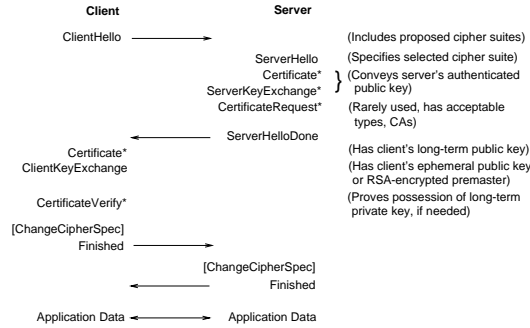
Client		Server	
Client		**Server**	
ClientHello ——————→			(Includes proposed cipher suites)
		ServerHello	(Specifies selected cipher suite)
		Certificate*	(Conveys server's authenticated
		ServerKeyExchange*	public key)
		CertificateRequest*	(Rarely used, has acceptable types, CAs)
	←——————	ServerHelloDone	
Certificate*			(Has client's long-term public key)
ClientKeyExchange			(Has client's ephemeral public key or RSA-encrypted premaster)
CertificateVerify*			(Proves possession of long-term private key, if needed)
[ChangeCipherSpec]			
Finished ——————→			
		[ChangeCipherSpec]	
	←——————	Finished	
Application Data ←——→		Application Data	

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

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[2], 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.

4.1 Public-key Cryptography in SSL

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 [4].

RSA-based Handshake The client and server exchange random nonces (used for replay protection) 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

---

[2] 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.

server uses its RSA private key to decrypt the pre-master secret. Both endpoints then use 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 MAC (Message Authentication Code) keys for bulk encryption by the Record Layer.

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.

5 ECC Hardware Acceleration

Point multiplication on elliptic curves is the fundamental and most expensive operation underlying both ECDH and ECDSA. For a point P in the group $(\{(x,y)|\, y^2 + xy = x^3 + ax^2 + b; x,y \in GF(2^m)\} \cup 0, +_P)$ defined by a non-supersingular elliptic curve with parameters $a, b \in GF(2^m)$ and for a positive integer k, the point multiplication kP is defined by adding P k-1 times to itself using $+_P$[3]. Computing kP is based on a sequence of modular additions, multiplications and divisions. To efficiently support ECC, these operations need to be implemented for large operands.

The design of our hardware accelerator was driven by the need to both provide high performance for named elliptic curves and support point multiplications for arbitrary, less frequently used curves. It is based on an architecture for binary polynomial fields $GF(2^m)$, $m \leq 255$. We believe that this maximal field degree offers adequate security strength for commercial web traffic for the foreseeable future. We chose to represent elements of $GF(2^m)$ in polynomial basis, i.e. polynomials $a = a_{m-1}t^{m-1} + a_{m-2}t^{m-2} + \cdots + a_1t + a_0$ are represented as bit strings $(a_{m-1}a_{m-2}\ldots a_1a_0)$.

---

[3] For a detailed mathematical background on ECC the reader is referred to [3].

5.1 Architectural Overview

We developed a programmable processor optimized to execute ECC point multiplication. The data path shown in Figure 3 implements a 256-bit architecture. Parameters and variables are stored in an 8kB data memory DMEM and program instructions are contained in a 1kB instruction memory IMEM. Both memories are dual-ported and accessible by the host machine through a 64-bit/66MHz PCI interface. The register file contains eight general purpose registers R0-R7, a register RM to hold the irreducible polynomial and a register RC for curve-specific configuration information.

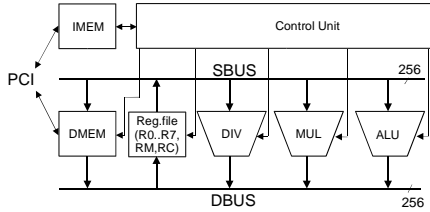The arithmetic units implement division (DIV), multiplication (MUL) and

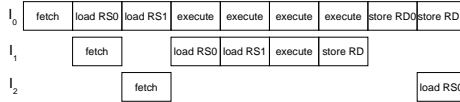**Fig. 3.** *Data Path and Control Unit.*

Fig. 4. *Parallel Instruction Execution.*

squaring/addition/shift left (ALU). Source operands are transferred over the source bus SBUS and results are written back into the register file over the destination bus DBUS.

Program execution is orchestrated by the Control Unit, which fetches instructions from the IMEM and controls the DMEM, the register file and the arithmetic units. As shown in Table 1, the instruction set is composed of memory instructions, arithmetic/logic instructions and control instructions. Memory instructions LD and ST transfer operands between the DMEM and the register file. The arithmetic and logic instructions include MUL, MULNR, DIV, ADD, SQR and SL. We implemented a load/store architecture. That is, arithmetic and logic instructions can only access operands in the register file. The execution of arithmetic instructions can take multiple cycles and, in case of division and multiplication, the execution time may even be data-dependent. To control the flow of the program execution, conditional branches BMZ and BEQ, unconditional branch JMP and program termination END can be used.

The data path allows instructions to be executed in parallel or overlapped. The Control Unit examines subsequent instructions and decides on the execution model based on the type of instruction and data dependencies. An example for parallel and overlapped execution of an instruction sequence $I_0; I_1; I_2$ is given in Figure 4. Parallel execution of $I_0; I_1$ is possible if I_0 is a MUL or MULNR instruction and I_1 is an ADD or SQR instruction and no data dependencies exist between the destination register/s of I_0 and the source and destination register/s of I_1. Execution of I_1 and I_2 can be overlapped if source register $RS0$

Instruction Type / 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}	≤ 2m + 4
MUL	RS0,RS1,RD	Multiply	(RS0*RS1) mod M → RD	RS0,RS1,RD={R0..R7}	8 (7)
MULNR	RS0,RS1,RD	Multiply w/o Reduction	RS0*RS1 → RD0,RD1	RS0,RS1,RD0,RD1= {R0..R7}	8
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
JMP	ADDR	Jump	jump		2
END		End	end program execution		

Table 1. *Instruction Set.*

of I_2 is different from destination register $RD1$ of I_0, i.e. $RS0$ can be read over the SBUS while $RD1$ is written over the DBUS.

5.2 ALU

The ALU incorporates two arithmetic and one logic operation: Addition, squaring and shift left. 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 $mod\, 2$. This can be efficiently computed as the bitwise XOR of the corresponding bit strings.

Squaring is a special case of multiplication and is defined in two steps. First, the operand $a \in GF(2^m)$ is multiplied by itself resulting in a polynomial $c_0 = a^2$ of degree less than $2m - 1$, i.e. $deg(c_0) < 2m - 1$. c_0 may not be an element of the underlying field since its degree may be greater than $m - 1$. Second, c_0 is reduced to a congruent polynomial $c \equiv c_0 \, mod\, M$, whereby $c \in GF(2^m)$ is defined as the residue of the polynomial divison of c_0 and the irreducible polynomial M. Squaring a does not require a full multiplication since all mixed terms $a_i a_j t^k, k = 1..2(m - 1), k = i + j, i \neq j$ occur twice cancelling each other out. Therefore, $a^2 = a_{m-1}t^{2(m-1)} + a_{m-2}t^{2(m-2)} + \cdots + a_1 t^2 + a_0$ can be easily computed by inserting zeros into the corresponding bit string. For example, squaring $(t^3 + t^2 + t + 1)$ results in $(1111)^2 = 1010101$.

Reduction is based on the congruency

$$u \equiv u + vM \, mod\, M \qquad (1)$$

for an irreducible polynomial M and arbitrary polynomials u and v. 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

$$c_0 = a^2 = c_{0,h} * t^m + c_{0,l} \qquad (2)$$

Using $t^m \equiv M - t^m \, mod \, M$ as a special case of (1), the congruency $c_1 = c_{0,h} * (M - t^m) + c_{0,l} \equiv c_0 \, mod \, M$ is obvious. Given that $deg(c_{0,h}) < m - 1$ and $deg(M - t^m) < m$, it follows that $deg(c_1) < 2m - 2$. By iteratively splitting up c_j into polynomials $c_{j,h}$ and $c_{j,l}$ such that

$$c_{j+1} = c_{j,h} * (M - t^m) + c_{j,l} \quad \text{until} \quad c_{j,h} = 0 \Leftrightarrow c_j \in GF(2^m) \qquad (3)$$

the reduced result $c = c_i$ can be computed in a maximum of $i \leq m - 1$ reduction iterations. The minimum number of iterations depends on the second highest term in the irreducible polynomial M [22], [12]. For

$$M = t^m + t^k + \sum_{j=1}^{k-1} M_j t^j + 1, \, 1 \leq k < m \qquad (4)$$

it follows that a better upper bound for $deg(c_1)$ is $deg(c_1) < m + k - 1$. Applying (3), $deg(c_j)$ gradually decreases such that

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

The minimum number of iterations i is given by

$$m - 1 - i(m - k) \leq 0 \Leftrightarrow i \geqslant \lceil \frac{m-1}{m-k} \rceil \qquad (5)$$

To enable efficient implementations, M is often chosen to be either a trinomial M_t or pentanomial M_p:

$$M_t = t^m + t^{k_3} + 1, \quad M_p = t^m + t^{k_3} + t^{k_2} + t^{k_1} + 1, \quad m > k_3 > k_2 > k_1 > 1$$

Choosing M such that $k_3 \leq \frac{m-1}{2}$ apparently limits the number of reduction iterations to 2, which is the case for all irreducible polynomials recommended by NIST [18] and SECG [7]. The multiplications $c_{j,h} * (M - t^m)$ can be optimized if $(M - t^m)$ is a constant sparse polynomial.

In this case, the two steps of a squaring operation can be hard-wired and executed in a single clock cycle. As shown in Figure 5, the ALU implements hard-wired reduction for the irreducible polynomials $t^{163} + t^7 + t^6 + t^3 + 1, t^{193} + t^{15} + 1$ and $t^{233} + t^{74} + 1$, respectively. Moreover, the ALU can compute addition (XOR) and execute a shift left. It further computes the flags EQ and MZ used by the branch instructions BEQ and BMZ as specified in Table 1.
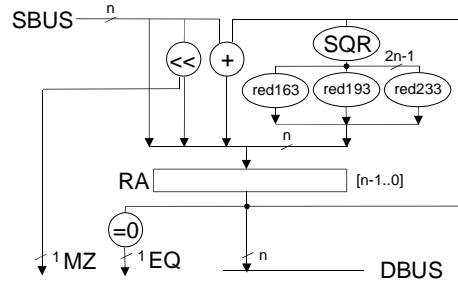

Fig. 5. *ALU.*

5.3 Multiplier

We studied and implemented several different architectures and, finally, settled on a digit-serial shift-and-add multiplier. Figure 6 gives a block diagram of the multiplier.

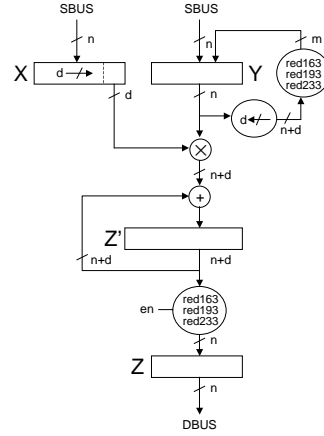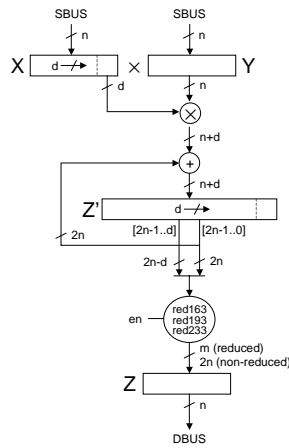The result is computed in two steps. First, the product is computed by iteratively

Fig. 6. *Shift-and-Add Multiplier.*   **Fig. 7.** *Least-Significant-Digit-First Multiplier.*

multiplying a digit of operand X with Y, and accumulating the partial products in Z'. Next, the product Z' is reduced by the irreducible polynomial. In our implementation, the input operands X and Y can have a size of up to $n = 256$ bits, and the reduced result Z has a size of $m = 163, 193, 233$ bits according to the specified named curve. The digit size d is 64. We optimized the number of iterations needed to compute the product Z' such that the four iterations it takes to perform a full 256-bit multiplication are only executed for $m = 193, 233$ whereas three iterations are executed for $m = 163$. To compensate for the missing shift operation in the latter case, a multiplexer was added to select the bits of Z' to be reduced. The reduction is hard-wired and takes another clock cycle.

The alternative designs we studied were based on the Karatsuba algorithm [14] and the LSD multiplier [22]. Applying the Karatsuba algorithm to Figure 6, we first split the 64-bit by 256-bit multiplication $X[63..0] * Y[255..0]$ into four 64-bit by 64-bit multiplications $X[63..0] * Y[63..0]$, $X[63..0] * Y[127..64]$, $X[63..0] * Y[191..128]$, $X[63..0] * Y[255..192]$ and then use the Karatsuba algorithm to calculate the four partial products. Compared with the shift-and-add algorithm the Karatsuba algorithm is attractive since it lowers the bit complexity from $O(n^2)$ to $O(n^{lg3})$ [6]. It does, however, introduce irregularities into the wiring and, as a result, additional wire delays. As we will show in Table 3, this design did not meet our timing goal.

We also implemented the LSD multiplier shown in Figure 7. When compared with the shift-and-add multiplier of Figure 6 the LSD multiplier is attractive since it reduces the size of the register used for accumulating the partial results from $2n$ bits to $n+d$ bits. This is accomplished by shifting the Y operand rather than the product Z' and reducing Y every time it is shifted. The implementation cost is an additional reduction circuit. Since the two reduction operations of Y and Z' do not take place in the same clock cycle, it is possible to share one reduction circuit. However, considering the additional placement and routing constraints imposed by a shared circuit, two separate circuits are, nevertheless, preferred. An analysis of our FPGA implementation shows no advantage in terms of size or performance. The size of the multiplier is dominated by the amount of combinational logic resources and, more specifically, the number of look-up tables (LUTs) needed. Thus, there is no advantage in reducing the size of the register holding Z'. Note, that as the digit size d is reduced, the ratio of registers and LUTs changes; given the fixed ratio of registers and LUTs available on an FPGA device, the LSD multiplier, therefore, can be attractive for small digit sizes.

As it is our goal to process arbitrary curve types, we can rely on the hard-wired reducers only for the named curves. All other curve types need to be handled in a more general way, for example, with the algorithm presented in Section 5.5. We, therefore, need a multiplier architecture that either provides a way to reduce by an arbitrary irreducible polynomial or offers the option to calculate a non-reduced product. We opted for the latter option and added a path to bypass the reducer in Figure 6. Note that with the LSD multiplier a non-reduced product can not be offered thus requiring full multipliers to replace the reduction circuits.

5.4 Divider

The hardware accelerator implements dedicated circuitry for modular division based on an algorithm described by Shantz [21]. A block diagram of the divider is shown in Figure 8. It 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:

$$A * Y \equiv U * X \, mod \, M \qquad (6) \qquad \qquad B * Y \equiv V * X \, mod \, M \qquad (7)$$

Through repeated additions and divisions by t, A and B are gradually reduced to 1 such that U (respectively V) contains the quotient $\frac{Y}{X} \, mod \, M$. 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. In contrast to the original algorithm, which included magnitude comparisons of registers A and B, we use two counters CA and CB to test for termination of

the algorithm. CB is initialized with the field degree m and CA with $m-1$. The division algorithm consists of the following operations:

1. **Division by t**
   (a) If even(A) and even(U): $A := \frac{A}{t}$, $CA := CA - 1$
   (b) If even(B) and even(V): $B := \frac{B}{t}$, $CB := CB - 1$
2. **Congruent addition of M**
   (a) If odd(U): $U := U + M$
   (b) If odd(V): $V := V + M$
3. **Addition of A and B**
   (a) If odd(A) and odd(B) and $CA > CB$: $A := A + B$, $U := U + V$
   (b) If odd(A) and odd(B) and $CA \leq CB$: $B := A + B$, $V := U + V$

The preconditions ensure that for any configuration of A, B, U and V at least one of the operations can be executed. It is interesting to note that operations, whose preconditions are satisfied, can be executed in any order without violating invariants (6) and (7). The control logic of the divider chooses operations as preconditions permit starting with 1a and 2a. To ensure termination, 3a is executed if $CA > CB$ and 3b is executed if $CA \leq CB$. 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 $CA \leq CB$. Postconditions of 2a, 2b, 3a and 3b guarantee that either 1a or 1b can be executed to further decrease the order of A and B towards 1.

The division circuit shown in Figure 8 was designed to execute sequences of operations per clock cycle, e.g. 3a,2a and 1a could be executed in the same cycle. In particular, it is possible to always execute either 1a or 1b once per clock cycle. Therefore, a modular division can be computed in a maximum of $2m$ clock cycles.
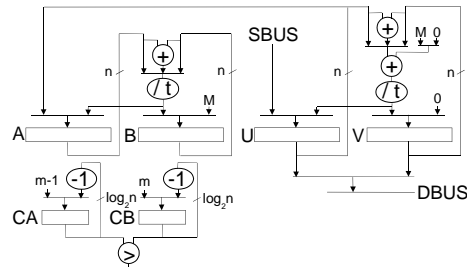

Fig. 8. *Divider.*

5.5 Point Multiplication Algorithms

We experimented with different point multiplication algorithms and settled on Montgomery Scalar Multiplication using projective coordinates as proposed by López and Dahab [16]. This choice is motivated by the fact that, for our implementation, multiplications can be executed much faster than divisions. Expensive divisions are avoided by representing affine point coordinates (x, y) as projective triples (X, Y, Z) with $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$. In addition, this algorithm is attractive since it provides protection against timing and power analysis attacks as each point doubling is paired with a point addition such that the sequence of instructions is independent of the bits in k.

A point multiplication kP can be computed with $\lfloor log_2(k) \rfloor$ point additions and doublings. Throughout the computation, only the X- and Z-coordinates

of two points $P_{1,i}$ and $P_{2,i}$ are stored. Montgomery's algorithm exploits the fact that for a fixed point $P = (X, Y, 1)$ and points $P_1 = (X_1, Y_1, Z_1)$ and $P_2 = (X_2, Y_2, Z_2)$ the sum $P_1 + P_2$ can be expressed through only the X- and Z-coordinates of P, P_1 and P_2 if $P_2 = P_1 + P$. P_1 and P_2 are initialized with $P_{1,\lfloor log_2(k) \rfloor} = P$ and $P_{2,\lfloor log_2(k) \rfloor} = 2P$. To compute kP, the bits of k are examined from left $(k_{\lfloor log_2(k) \rfloor})$ to right (k_0). For $k_i = 0$, $P_{1,i}$ is set to $2P_{1,i+1}$ (8) and $P_{2,i}$ is set to $P_{1,i+1} + P_{2,i+1}$ (9).

$$X_{1,i} = X_{1,i+1}^4 + bZ_{1,i+1}^4 \qquad Z_{2,i} = (X_{1,i+1} * Z_{2,i+1} + X_{2,i+1} * Z_{1,i+1})^2$$
$$Z_{1,i} = Z_{1,i+1}^2 * X_{1,i+1}^2 \quad (8) \quad X_{2,i} = XZ_{2,i} + (X_{1,i+1}Z_{2,i+1})(X_{2,i+1}Z_{1,i+1}) \quad (9)$$

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. In projective coordinates, Montgomery Scalar 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.

Named Curves An implementation of Equations (8) and (9) for named curves over $GF(2^{163})$, $GF(2^{193})$ and $GF(2^{233})$ is shown in Table 2. The computation

```
// register        R0, R1, R2, R3                          Code Execution
// value            X1, Z1, X2, Z2
MUL(R1, R2, R2)     R2 = Z1 * X2                            MUL(R1, R2, R2); SQR(R1, R1)
SQR(R1, R1)         R1 = Z1^2
MUL(R0, R3, R4)     R4 = X1 * Z2                            MUL(R0, R3, R4); SQR(R0, R0)
SQR(R0, R0)         R0 = X1^2
ADD(R2, R4, R3)     R3 = Z1 * X2 + X1 * Z2                  ADD(R2, R4, R3)
MUL(R2, R4, R2)     R2 = Z1 * X2 * X1 * Z2                  MUL(R2, R4, R2); SQR(R1, R4)
SQR(R1, R4)         R4 = Z1^4
MUL(R0, R1, R1)     R1 = Z1^2 * X1^2                        MUL(R0, R1, R1); SQR(R3, R3)
SQR(R3, R3)         R3 = Z3 = (Z1 * X2 + X1 * Z2)^2
LD(data_mem_b, R5)  R5 = b                                  LD(data_mem_b, R5)
MUL(R4, R5, R4)     R4 = b * Z1^4                           MUL(R4, R5, R4); SQR(R0, R0)
SQR(R0, R0)         R0 = X1^4
LD(data_mem_Px, R5) R5 = X                                  LD(data_mem_Px, R5)
MUL(R3, R5, R5)     R4 = X * (Z1 * X2 + X1 * Z2)^2          MUL(R3, R5, R5); ADD(R4, R0, R0)
ADD(R4, R0, R0)     R0 = X1^4 + b * Z1^4
ADD(R2, R5, R2)     R2 = X * Z3 + (Z1 * X2) * (X1 * Z2)    ADD(R2, R5, R2)
```
Table 2. *Implementation and Execution of Projective Point Doubling and Addition.*

of the two equations is interleaved such that there are no data dependencies for any MUL/SQR or MUL/ADD instruction sequences. Hence, all MUL/SQR and MUL/ADD sequences can be executed in parallel. Furthermore, there are no data dependencies between subsequent arithmetic instructions allowing for overlapped execution.

Generic Curves Squaring and multiplication require reduction, which can either be hard-wired or implemented for arbitrary field degrees through an instruction sequence of polynomial multiplications (MULNR) and additions (ADD) as

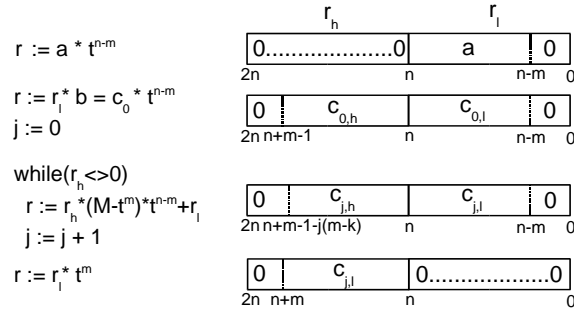shown in Section 5.2. Figure 9 shows a multiplication including reduction. Note

Fig. 9. *Non-Hard-Wired Reduction through Multiplication and Addition.*

that the multiplier includes registers r_l and r_h, which have a width $n = 256$ not equal to the field degree m. Therefore, the constant factor t^{n-m} is used to align multiplication results to the boundary between r_l and r_h. Computing $c_{j+1,h}$ and $c_{j+1,l}$ from $c_{j,h}$ and $c_{j,l}$ based on Equation (3) requires one MULNR and one ADD instruction. Hence, multiplication and squaring operations with reduction for arbitrary field degrees can be computed with $3 + i$ MULNR and i ADD instructions with i as in Equation (5). Looking at the code sequence of a point multiplication, optimization can be done by storing some multiplication results multiplied by t^{n-m} omitting the first and last step.

6 Implementation and Performance

We specified the hardware in Verilog and prototyped it in a Xilinx Virtex XCV2000E-FG680-7 FPGA using the design tools Synplify 7.0.2 and Xilinx Design Manager 3.3.08i. Area constraints were given for the ALU, the divider and the register file, but no manual placement had to be done. The prototype runs off the PCI clock at a frequency of 66.4 MHz.

Table 3 summarizes the cost and performance of the ALU, the divider and three multiplier design alternatives. The cost is given as the number of used 4-input look-up tables (LUTs) and flip-flops (FFs). The multiplier clearly dominates the design size with 73% of the LUTs and 46% of the FFs. However, multiplication is the single most time-critical operation as shown in Table 4. For point multiplication over $GF(2^{163})$, field multiplications account for almost 62% 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 more than 27% time compared to sequential execution. There is still room for improvements since instructions BMZ, BEQ, SL, JMP and END responsible for the flow control consume almost 21% of the execution time. This time could be saved by separating control flow and data flow.

Instruction	#Instr.	Cycles	ms
DIV	1	329	0.00495
ADD	333	666	0.01003
SQR	3	6	0.00009
MUL	10	60	0.00090
MULNR	1	7	0.00011
MUL + ADD	162	972	0.01464
MUL + SQR	810	4860	0.07319
ST	11	33	0.00050
LD	344	688	0.01036
BMZ	326	652	0.00982
BEQ	2	8	0.00012
JMP	162	324	0.00488
SL	326	978	0.01473
END	1	5	0.00008
total		9588	0.14440

Table 4. *Decomposition of the Execution Time for a $GF(2^{163})$ Point Multiplication.*

Unit	LUTs	FFs	f[MHz]
Karatsuba Multiplier	9870	2688	52.2
LSD Multiplier	14347	2592	66.6
Shift-and-Add Multiplier	14241	2990	66.5
ALU	1345	279	80.4 (est.)
Divider	2678	1316	79.6 (est.)
Full Design	19508	6442	66.5

Table 3. *Cost and Performance of Arithmetic Units.*

To evaluate the performance of the divider, we implemented an inversion algorithm proposed by Itoh and Tsujii [13] based on Fermat's theorem. With this algorithm, an inversion optimized for $GF(2^{163})$ takes 938 cycles (0.01413 ms), while the divider is almost three times faster speeding up point multiplication by about 6.4%.

Table 5 shows hardware and software performance numbers for point multiplication on named and generic curves as well as execution times for ECDH and ECDSA with and without hardware support. The hardware numbers were obtained on a 360MHz Sun Ultra™60 workstation and all software numbers represent a generic 64-bit implementation measured on a 900MHz Sun Fire™280R server. For generic curves, the execution time for point multiplications depends on the irreducible polynomial as described in Sections 5.5 and 5.2. The obtained numbers assume irreducible polynomials with $k_3 \leq \frac{m-1}{2}$. Hard-wired reduction for named curves improves the execution time by a factor of approximately 10 compared to generic curves.

For ECDH-163, the hardware accelerator offers a 12.5-fold improvement in execution time over the software implementation for named curves. Overhead is created by OpenSSL and accesses to the hardware accelerator leading to a lower speedup than measured for raw point multiplication. A disproportionally larger drop in speedup can be observed for ECDSA-163 since it includes integer operations executed in software. In addition, ECDSA signature verification requires two point multiplications. All numbers were measured using a single process on one CPU. The hardware numbers for ECDH and ECDSA could be improved by having multiple processes share the hardware accelerator such that while one processes waits for a point multiplication to finish, another process can use the CPU.

	Hardware		Software		Speedup
	ops/s	ms/op	ops/s	ms/op	
Named Curves					
$GF(2^{163})$	6987	0.143	322	3.110	21.7
$GF(2^{193})$	5359	0.187	294	3.400	18.2
$GF(2^{233})$	4438	0.225	223	4.480	19.9
Generic Curves					
$GF(2^{163})$	644	1.554	322	3.110	2.0
$GF(2^{193})$	544	1.838	294	3.400	1.9
$GF(2^{233})$	451	2.218	223	4.480	2.0
ECDH					
$GF(2^{163})$	3813	0.235	304	3.289	12.5
ECDSA (sign)					
$GF(2^{163})$	1576	0.635	292	3.425	5.4
ECDSA (verify)					
$GF(2^{163})$	1224	0.817	151	6.623	8.1

Table 5. *Hardware and Software Performance.*

7 Conclusions

We have demonstrated a secure client/server system that employs elliptic curve cryptography for the public-key operations in OpenSSL. We have further presented a hybrid hardware accelerator architecture providing optimized performance for named elliptic curves and support for generic curves over arbitrary fields $GF(2^m)$, $m \leq 255$. Previous approaches such as presented in [11] and [20] focused on only one of these aspects.

The biggest performance gain was achieved by optimizing field multiplication. However, as the number of cycles per multiplication decreases, the relative cost of all other operations increases. In particular, squarings can no longer be considered cheap. Data transport delays become more critical and contribute to a large portion of the execution time. To make optimal use of arithmetic units connected through shared data paths, overlapped and parallel execution of instructions can be employed.

For generic curves, reduction has shown to be the most expensive operation. As a result, squarings become almost as expensive as multiplications. This significantly impacts the cost analysis of point multiplication algorithms. In particular, the Itoh-Tsujii method becomes much less attractive since it involves a large number of squaring operations.

Dedicated division circuitry leads to a performance gain over soft-coded inversion algorithms for both named and generic curves. However, the tradeoff between chip area and performance needs to be taken into account.

Although prototyped in reconfigurable logic, the architecture does not make use of reconfigurability. It is thus well-suited for an implementation in ASIC technology. For commercial applications this means lower cost at high volumes, less power consumption, higher clock frequencies and tamper resistance.

As for future work, we are in the process of setting up a testbed that will allow us to empirically study the performance of ECC-based cipher suites and compare it to conventional cipher suites. This includes measurements and analysis of the system performance at the web server level. As for the hardware accelerator,

we intend to improve the performance of point multiplication on generic curves. Furthermore, we want to optimize the hardware-software interface to achieve higher performance at the OpenSSL level. We plan to discuss the results in a follow-on publication.

References

[1] Agnew, G.B., Mullin, R.C., Vanstone, S.A.: An Implementation of Elliptic Curve Cryptosystems Over $F_{2^{155}}$. In IEEE Journal on Selected Areas in Communications, 11(5):804-813, June 1993.

[2] Bednara, M., Daldrup, M., von zur Gathen, J., Shokrollahi, J., Teich, J.: Reconfigurable Implementation of Elliptic Curve Crypto Algorithms. Reconfigurable Architectures Workshop, 16th International Parallel and Distributed Processing Symposium, April 2002.

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

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

[5] Blum, T., Paar, C.: High Radix Montgomery Modular Exponentiation on Reconfigurable Hardware. To Appear in IEEE Transactions on Computers.

[6] Borodin, A., Munro, I.: The Computational Complexity of Algebraic and Numeric Problems. American Elsevier, New York, 1975.

[7] Certicom Research: SEC 2: Recommended Elliptic Curve Domain Parameters. Standards for Efficient Cryptography, Version 1.0, September 2000.

[8] Dierks, T., Allen, C.: The TLS Protocol - Version 1.0. IETF RFC 2246, January 1999.

[9] Ernst, M., Klupsch, S., Hauck, O., Huss, S.A.: Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems. 12th IEEE Workshop on Rapid System Prototyping, Monterey, CA, June 2001.

[10] Gao, L., Shrivastava, S., Lee, H., Sobelman, G.: A Compact Fast Variable Key Size Elliptic Curve Cryptosystem Coprocessor. Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 1998.

[11] Goodman, J., Chandrakasan, A.P.: An Energy-Efficient Reconfigurable Public-Key Cryptography Processor. In IEEE Journal of Solid-State Circuits, Vol. 36, No. 11, 1808-1820, November 2001.

[12] Halbutoğulları, A., Koç, Ç.K.: Mastrovito Multiplier for General Irreducible Polynomials. In IEEE Transactions on Computers, Vol. 49, No. 5, 503-518, May 2000.

[13] Itoh, T., Tsujii, S.: A Fast Algorithm for Computing Multiplicative Inverses in $GF(2^m)$ Using Normal Bases. In Information and Computation, 78:171-177, 1988.

[14] Karatsuba, A., Ofman, Y.: Multiplication of Many-Digital Numbers by Automatic Computers. Doklady Akad. Nauk, SSSR 145, 293-294. Translation in Physics-Doklady 7, 595-596, 1963.

[15] Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation, 48:203-209, 1987.

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

[17] Miller, V.: Uses of elliptic curves in cryptography. In Lecture Notes in Computer Science 218: Advances in Crytology - CRYPTO '85, pages 417-426, Springer-Verlag, Berlin, 1986.

[18] U.S. Department of Commerce / National Institute of Standards and Technology: Digital Signature Standard (DSS), Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[19] See `http://www.openssl.org/`.

[20] Orlando, G., Paar, C.: A High-Performance Reconfigurable Elliptic Curve Processor for $GF(2^m)$. In CHES '2000 Workshop on Cryptographic Hardware and Embedded Systems, Springer-Verlag, Lecture Notes in Computer Science 1965, August 2000.

[21] Chang Shantz, S.: From Euclid's GCD to Montgomery Multiplication to the Great Divide. Sun Microsystems Laboratories Technical Report TR-2001-95, `http://research.sun.com/`, June 2001.

[22] Song, L., Parhi, K.K.: Low-Energy Digit-Serial/Parallel Finite Field Multipliers. In IEEE Journal of VLSI Signal Processing Systems 19, 149-166, 1998.