A Low-Power Design for an Elliptic Curve Digital Signature Chip

Richard Schroeppel, Cheryl Beaver, Rita Gonzales, Russell Miller, and Timothy Draelos

Sandia National Laboratories** Albuquerque, NM 87185-0785
{rschroe, cbeaver, ragonza, rdmille, tjdrael}@sandia.gov

Abstract. We present a VHDL design that incorporates optimizations intended to provide digital signature generation with as little power, space, and time as possible. These three primary objectives of power, size, and speed must be balanced along with other important goals, including flexibility of the hardware and ease of use. The highest-level function offered by our hardware design is Elliptic Curve Optimal El Gamal digital signature generation. Our parameters are defined over the finite field $GF(2^{178})$, which gives security that is roughly equivalent to that provided by 1500-bit RSA signatures.
Our optimizations include using the point-halving algorithm for elliptic curves, field towers to speed up the finite field arithmetic in general, and further enhancements of basic finite field arithmetic operations. The result is a synthesized VHDL digital signature design (using a CMOS $0.5\mu m$, $5V$, $25°C$ library) of 191,000 gates that generates a signature in 4.4 ms at 20 MHz.

Keywords: Digital Signature, Elliptic Curve, ECDSA, Optimal El Gamal, Characteristic 2, Field Towers, Trinomial Basis, Quadratic Equation, Qsolve, Almost-Inverse Algorithm, Point Halving, Signed Sliding Window, GF(2^{89}), GF(2^{178}), Hardware, VHDL, Low Power

1 Introduction

While the value of elliptic curve arithmetic in enabling public-key cryptography to serve in resource-constrained environments is well accepted, efforts in creative implementations continue to bear fruit. A particularly active area is that of hardware implementations of elliptic curve operations, including hardware description language developments, programmable hardware realizations, and fabricated custom circuits. Kim, et al, [1] introduce a hardware architecture to take advantage of a nonconventional basis representation of finite field elements to make point multiplication more efficient. Moon, et al, [2] address field multiplication and division, proposing new methods for fast elliptic curve arithmetic

** Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

B.S. Kaliski Jr. et al. (Eds.): CHES 2002, LNCS 2523, pp. 366–380, 2003.

appropriate for hardware. Goodman and Chandrakasan [3] tackle the broader problem of providing energy-efficient public-key cryptography in hardware while supporting multiple algorithms, including elliptic curve-based algorithms. Moving closer to applications of elliptic curve cryptography (ECC), Aydos, et al, [4] have implemented an ECC-based wireless authentication protocol that utilizes the elliptic curve digital signature algorithm (ECDSA).

We present a VHDL[1] design that incorporates optimizations intended to provide elliptic curve-based digital signature generation with as little power, space, and time as possible. These three primary objectives of power, size, and speed must be balanced along with other important goals, including flexibility of the hardware (e.g., support of a class of elliptic curves) and ease of use (e.g., doesn't require the user to supply or interpret complex parameters). Currently, the highest-level function offered by our hardware design is digital signature generation. Our elliptic curve parameters are defined over the finite field $GF(2^{178})$, which gives security roughly equivalent to that provided by 1500-bit RSA signatures[2]. As we don't currently have hardware and therefore explicit power measurements, the emphasis of this paper is on a design that reflects many choices promoting a low-power outcome. In addition to the obvious goal of minimizing the number of gates, the speed of execution is critical to power consumption since power can be removed from circuits as soon as they complete their functions.

ECC solutions are well-known for their suitability in smart-card applications and wireless communications security. Our work was motivated by the need to reduce the resources required to provide strong public-key authentication for sensor-based monitoring systems and critical infrastructure protection. For these applications, signature generation is often performed in highly constrained, battery-operated environments, whereas signature verification is performed on desktop systems with only the typical constraint of purchasing power. Hence, our hardware design focused primarily on the signature generation, with signature verification to follow. Here, we present a chip design represented in VHDL of the best to date, in our minds and for our applications, digital signature generation solution for low-power, resource-constrained environments.

In Section 2, we start with the selection of an El Gamal digital signature variant that minimizes the number of operations necessary for signature generation. Section 3 presents algorithmic optimizations of the computational elements necessary to compute a digital signature. We note that many of these elements

[1] VHDL stands for VHSIC Hardware Description Language, where VHSIC stands for Very High Scale Integrated Circuit.

[2] The number of computer instructions to factor a number, N, is estimated as $0.018e^{(1.923\sqrt[3]{logN(loglogN)^2})}$. The multiplier 0.018 is selected to give a figure of 10,000 MIPS-years to factor a 512-bit number. The number of elliptic curve operations (point addition or halving) to solve a discrete log problem (and discover a secret signing key, or forge a signature on a given message) is roughly the square root of the group size. Our group size is about 2^{177} so the breaking work is about $2^{88.5}$ curve operations. Assuming $1,000$ computer instructions per elliptic curve operation, this number of instructions would factor a 1570-bit number.

can be applied to other elliptic curve algorithms over $GF(2^m)$. The focus of this paper is on the implementation of highly-optimized versions of these core operations. Section 4 presents the VHDL implementation of the digital signature algorithm and elliptic curve arithmetic operations. In Section 5, we provide results of the number of gates and time required to generate a digital signature and perform many of the underlying primitive functions that might be used in an elliptic curve coprocessor.

2 The 'Optimal El Gamal' Authentication Algorithm

2.1 Optimal El Gamal Scheme

The signature algorithm is the Optimal El Gamal digital signature scheme adapted for use with elliptic curves (see [5], [6] for original description and security proofs). For an introduction to elliptic curves see [7]. This variant of the El Gamal algorithm was chosen because it avoids the computationally expensive modular reciprocal during signature generation and verification.

Parameters. The public parameters for the Optimal El Gamal Signature scheme are (E, G, W, r) where E is a choice of an elliptic curve, $G \in E$ is a point of large order, r, and $W = sG$ is the public key where s $(1 < s < r)$ is the long-term private key. We assume that the public key parameters and a common hash function are available to all relevant algorithms.

We denote by x_P (resp. y_P) the $x-$coordinate of a point $P \in E$ (resp. $y-$coordinate).

Alg. 1 Elliptic Curve Optimal El Gamal Signature Generation

Input: *Private Key, s; Message, M* **Output:** *Signature (c,d) of M*

1. *Generate a key pair* $(v, V = vG)$*, where* $v \neq 0$ *is a randomly chosen integer modulo r*
2. *Compute* $c \equiv x_V \pmod{r}$*; If* $c = 0$*, then go to Step 1*
3. *Let* $f = Hash(M)$*. Compute an integer* $d = (cfs + v) \pmod{r}$*; If* $d = 0$*, then go to Step 1*
4. *Output the pair* (c, d) *as the signature*

Although our current chip design does not include signature verification, we describe the algorithm for completeness. Most of the optimizations presented later in the paper will benefit signature verification as well as generation.

Alg. 2 Signature Verification
Input: *Signature (c,d) on Message M* **Output:** *Accept/Reject*

1. *If* $c \notin [1, ..., r-1]$, *or* $d \notin [1, ..., r-1]$, *output "Reject" and stop*
2. *Compute* $f = Hash(M)$
3. *Compute the integer* $h = cf \pmod{r}$
4. *Compute an elliptic curve point* $P = dG - hW$; *If* $P = \mathcal{O}$, *output "Reject" and stop*
5. *Compute* $c' \equiv x_P \pmod{r}$
6. *If* $c' = c$, *then output "Accept"*

We note that anyone can forge a signature on a message that hashes to 0. However, inverting the hash to find such a message is thought to be computationally infeasible.

3 Algorithmic Optimizations

The field of definition for the elliptic curve is important since it is the basis for all elliptic curve operations. Generally the curve is defined over either $GF(p)$ for some large prime p, or $GF(2^m)$. Since the arithmetic in the latter field is much faster, that was our choice. In particular, we use the field $GF(2^{178})$. One reason for choosing this field is to make use of optimizations that can be derived from the fact that it can be realized as a field tower: $GF(2^{178}) = GF((2^{89})^2)$. In the case of characteristic two fields, the equation for the elliptic curve can be given by $E : y^2 + xy = x^3 + ax^2 + b$. For simplicity and saving on storage, we assume that $a = (1, 0)$. This is useful since the point addition algorithms use a but not b so we don't need to store b (it is implied by the coordinates of the generating point). Further, we exploit properties of $GF(2^{89})$ to reduce some of the basic arithmetic operations (e.g. squaring, square root) to simple XOR gates which are very fast in hardware. The 'almost inverse' algorithm in [8] is especially fast for smaller degree fields. Finally, we modify our elliptic curve multiplication algorithm to use point halving [9,10] which offers a savings over the usual point doubling.

3.1 Finite Field Arithmetic and Field Towers

Our first optimization involves field towers, which simplifies all underlying operations. The finite field is

$$GF(2^m) \cong GF(2)[x]/f(x) = \{a_0 + a_1 x + \cdots + a_{m-1}x^{m-1} \pmod{f(x)} \mid a_i \in GF(2)\}$$

where $f(x)$ is an irreducible binary polynomial of degree m. An element $a \in GF(2^m)$ can therefore be represented as an $m-$tuple $a = (a_0, a_1, ..., a_{m-1})$ of zeros and ones. Addition of two elements is a bitwise exclusive-OR (XOR) operation:

$$a, b \in GF(2^m),\ a + b = (a_0 \oplus b_0, a_1 \oplus b_1, ..., a_{m-1} \oplus b_{m-1})$$

and multiplication is like a plain multiplication without any carries but with the XOR accumulation only. The result of the multiplication must, however, be reduced by the field polynomial $f(x)$. As the degree m of the field gets large, the multiplication can become time-consuming and the representation of the numbers can become cumbersome. For a general reference on finite field arithmetic, see [11].

If m is a composite number, we can use field towers to speed-up the computations. Suppose $m = ns$. Then we can think of $GF(2^m) = GF((2^n)^s)$ as a degree s extension of $GF(2^n)$. The elements are $a \in GF(2^m), a = (\alpha_0, \alpha_1, ..., \alpha_{s-1})$, where $\alpha_i \in GF(2^n)$.

For this work, we use the finite field $GF(2^{178})$ and the corresponding field tower $GF((2^{89})^2)$. Our choice of irreducible polynomial for $GF(2^{89})$ over $GF(2)$ is $f(u) = u^{89} + u^{38} + 1$, and the irreducible polynomial we use for $GF((2^{89})^2)$ over $GF(2^{89})$ is $g(V) = V^2 + V + 1$. We note that this field is not susceptible to known attacks on elliptic curves over composite degree fields (see [12]). Using a trinomial for the field polynomial over $GF(2^{89})$ makes the modular reduction easy and also helps with squaring, square root, Qsolve, and the finishing step in the almost inverse algorithm.

3.2 Finite Field Algorithms

As elements of $GF(2^{178})$ are represented as pairs of $GF(2^{89})$ elements, all algorithms can be described using the arithmetic over $GF(2^{89})$. While some of our optimizations are for general fields, some are specific to our chosen field. We first describe any optimizations over $GF(2^{89})$ and then give algorithms for the extension to $GF(2^{178})$.

Algorithms over $GF(2^{89})$. Unlike the situation with real numbers, squaring and square-rooting are one-to-one operations in characteristic 2 finite fields. Every field element has a single unique square root and square. The following algorithms are specific to $GF(2^{89})$ with field polynomial $f(u) = u^{89} + u^{38} + 1$. In the case of squaring, square root, and solving the equation $a = z^2 + z$ for z (which we call "QSolve"), we note that the algorithmic descriptions can be reduced to simple XORs of the input bits.

Alg. 3 *Squaring*
Input: $a = (a_{00}, ..., a_{88}) \in GF(2^{89})$
Output: $z = (z_{00}, ..., z_{88}) \in GF(2^{89})$ *where* $z = a^2$

z even bits:

$$z_{00} - z_{36} : z_{2n} = a_n \oplus a_{n+70}$$
$$z_{38} - z_{74} : z_{2n} = a_n \oplus a_{n+51}$$
$$z_{76} - z_{88} : z_{2n} = a_n$$

z odd bits:

$$z_{01} - z_{37} : z_{2n+1} = a_{n+45}$$
$$z_{39} - z_{87} : z_{2n+1} = a_{n+45} \oplus a_{n+26}$$

Alg. 4 *Square Root*
Input: $a = (a_{00}, ..., a_{88}) \in GF(2^{89})$
Output: $z = (z_{00}, ..., z_{88}) \in GF(2^{89})$ *where* $z = \sqrt{a}$

$$
\begin{aligned}
&z_{00} - z_{12} : z_n = a_{2n} \oplus a_{2n+51} \oplus a_{2n+13} \\
&z_{13} - z_{18} : z_n = a_{2n} \oplus a_{2n+51} \oplus a_{2n+13} \oplus a_{2n-25} \\
&z_{19} - z_{31} : z_n = a_{2n} \oplus a_{2n+13} \oplus a_{2n-25} \\
&z_{32} - z_{37} : z_n = a_{2n} \oplus a_{2n-63} \oplus a_{2n+13} \oplus a_{2n-25} \\
&z_{38} - z_{44} : z_n = a_{2n} \\
&z_{45} - z_{63} : z_n = a_{2n-89} \\
&z_{64} - z_{82} : z_n = a_{2n-89} \oplus a_{2n-127} \\
&z_{83} - z_{88} : z_n = a_{2n-89} \oplus a_{2n-127} \oplus a_{2n-165}
\end{aligned}
$$

Quadratic Solve. We developed a special circuit for computing QSolve with a relatively small number of XOR gates (387) and depth (35). The full circuit and detailed derivation are in [13].

Alg. 5 *Qsolve*
Input: $a = (a_{00}, ..., a_{88}) \in GF(2^{89})$
Output: $z = (z_{00}, ..., z_{88}) \in GF(2^{89})$ *where* $a = z^2 + z$

Except for odd z in the range $z_{01} - z_{19}$ *(which are computed directly), the bits of z are computed from the following equations:*

a even bits:

$$
\begin{aligned}
&a_{00} - a_{36} : a_{2n} = z_{2n} \oplus z_n \oplus z_{n+70} \\
&a_{38} - a_{74} : a_{2n} = z_{2n} \oplus z_n \oplus z_{n+51} \\
&a_{76} - a_{88} : a_{2n} = z_{2n} \oplus z_n
\end{aligned}
$$

a odd bits:

$$
\begin{aligned}
&a_{01} - a_{37} : a_{2n+1} = z_{2n+1} \oplus z_{n+45} \\
&a_{39} - a_{87} : a_{2n+1} = z_{2n+1} \oplus z_{n+45} \oplus z_{n+26}
\end{aligned}
$$

This derivation uses several observations to reduce the number of gates.

1. QSolve is linear, so we could precompute QSolve(u^N) for each N. The runtime circuit XORs together the appropriate subset for a general polynomial (see [14] for one method of doing the precomputation). This is fast, but uses a lot of gates. We traded speed for size, getting a slower but smaller circuit.
2. We reduced the number of required QSolve(u^N) values by removing some powers of u from the problem. For example, the substitution QSolve(u^{2N}) $\Rightarrow u^N + QSolve(u^N)$ eliminates even powers of u. The substitution $u^N \Rightarrow u^{N-38} + u^{N+51}$ removes some odd powers of u. After repeated substitutions like these, QSolve(u^N) is only needed for odd N in the range 1...19.
3. Only some of the answer bits are required: z_{odd} in the range $z_{01}...z_{19}$. This reduces the number of gates considerably. The remaining bits can be recovered by solving the bit equations for QSolve. For example, we compute z_{45} from the equation $a_{01} = z_{01} \oplus z_{45}$.

4. We assume that a_{00} is equal to a_{51}. The actual value of a_{00} is ignored. Furthermore, z_{00} is irrelevant, and is set equal to 0.

Our minimal size QSolve circuit used only 287 XOR gates, but had depth 65. We moved back from this extreme point on the speed-size tradeoff curve to a circuit with 387 XOR gates and depth 35.

Division. Inversion over $GF(2^{89})$ is performed with an "almost inverse" algorithm [8]. Division is a reciprocal followed by a multiply.

Algorithms over $GF(2^{178})$. We consider $GF(2^{178})$ as a degree two extension of $GF(2^{89})$ with field polynomial $V^2 + V + 1$. Elements of $\omega \in GF(2^{178})$ are pairs of elements from $GF(2^{89})$. So $\omega = (u_1, v_1)$ where $u_1, v_1 \in GF(2^{89})$; i.e. $\omega = u_1 V + v_1$ where $V^2 = V + 1$. The algorithms from $GF(2^{89})$ are extended to $GF(2^{178})$ in the obvious way. We give here some examples where some optimizations have been made.

Alg. 6 *Multiplication* $GF(2^{178})$
Input: $x = (u_1, v_1)$, $y = (u_2, v_2)$; **Output:** $z = x * y = (u_3, v_3)$

1. $u_3 = (u_1 + v_1)(u_2 + v_2) + v_1 v_2$
2. $v_3 = u_1 u_2 + v_1 v_2$

Alg. 7 *Inversion* $GF(2^{178})$
Input: $x = (u_1, v_1)$; **Output:** $x^{-1} = (u_2, v_2)$

1. $u_2 = \frac{u_1}{(u_1+v_1)^2+u_1 v_1}$
2. $v_2 = \frac{u_1+v_1}{(u_1+v_1)^2+u_1 v_1}$

Alg. 8 *Squaring* $GF(2^{178})$
Input: $x = (u_1, v_1)$; **Output:** $x^2 = (u_2, v_2)$

1. $u_2 = u_1^2$
2. $v_2 = u_1^2 + v_1^2$

Alg. 9 *Square Root* $GF(2^{178})$
Input: $x = (u_1, v_1)$; **Output:** $\sqrt{x} = (u_2, v_2)$

1. $u_2 = \sqrt{u_1}$
2. $v_2 = \sqrt{u_1} + \sqrt{v_1}$

Alg. 10 *Qsolve* $GF(2^{178})$
Input: $a = (u_1, v_1)$; **Output:** $z = (u_2, v_2)$ *such that* $a = z^2 + z$

1. $u_2 = Qsolve(u_1)$ *(per Alg. 5)*
2. *Set* $t = u_1 + v_1 + u_2 = t_0 t_1 ... t_{88}$
3. *If* $t_0 \oplus t_{51} = 1$, *then* $u_2 = u_2 + 1$ *and* $t = t + 1$
4. $v_2 = Qsolve(t)$

In both $GF(2^{89})$ and $GF(2^{178})$, only half of the field elements, a, have a corresponding solution, z. Moreover, when z is a solution, so is $z + 1$. In step 3 of Algorithm 10, we choose the Qsolve solution u_2 so that t can be Qsolved in step 4.

3.3 Point Halving Algorithm

The slowest part of the signature algorithm is the multiplication of points. We modified the point multiplication algorithm to use a point halving algorithm in place of a doubling algorithm. The idea of "halving" a point $P = (x_P, y_P)$ is to find a point $Q = (x_Q, y_Q)$ such that $2Q = P$. Note this is the inverse of the point doubling problem. The point halving can nevertheless be used in our algorithm by a simple adjustment on the base point of the elliptic curve used. The algorithm offers a speed-up in software of a factor of about two to three over the point doubling algorithm. We follow the algorithm of [9].

For this algorithm we sometimes write the coordinates of the points $P \in E$ as (x_P, r_P) where $r_P = x_P / y_P$. In fact, we use the (x_P, r_P) form whenever possible, but the input and output of the point addition algorithm need the Y coordinate, so the halving algorithm must handle Y outputs and inputs. When the y_Q output is not required, the point halving algorithm needs only one field multiplication. It is most efficient when point halvings are consecutive. Our signed sliding window multiplication method uses about five halvings between additions.

Alg. 11 *Point Halving over* $GF(2^m)$
Input: $P \in E$ **Output:** $Q = \frac{1}{2}P \in E$

1. $M_h = Qsolve(x_P + a)$, *where* a *is the curve parameter*
2. $T = x_P * (M_h + r_P)$ *or* $T = x_P * M_h + y_P$
3. *If parity*$(T$ *and* $t_m) = 0$, *then* $M_h = M_h + 1$; $T = T + x_P$
   Here t_m *is a mask that depends upon the modulus polynomial. In our case,* $t_m = (u^{51} + 1, 0)$.
4. $x_Q = \sqrt{T}$
5. $r_Q = M_h + x_Q + 1$
6. *If needed,* $y_Q = x_Q * r_Q$

3.4 Sliding Window Multiplication with Precomputation

The computation of elliptic curve multiplication, nP, is performed using a 4-bit signed sliding window algorithm [8]. The table of precomputed values $\{1, 3, 5, 7\}G$ are stored and the circuit automatically computes the negatives $\{-1, -3, -5, -7\}G$ as needed on the fly. On average there are 5 halvings per addition.

4 Hardware Architecture and Design

The hardware design is a full VHDL implementation of the Elliptic Curve Optimal El Gamal Signature algorithm that can be targeted to a Field Programmable Gate Array (FPGA) or an Application Specific Integrated Circuit (ASIC). The implementation is a VHDL Register-Transfer-Level design. The goal is to maximize speed and functionality while conserving area and therefore power.

The overall strategy was first to develop a set of basic $GF(2^m)$ arithmetic blocks (in VHDL) that would be used throughout the design. Basic building blocks include addition, multiplication, reciprocal, squaring, etc. The design was then built with these blocks to create the full algorithm implementation for point addition, point halving, point multiplication, and signature generation. The VHDL implementation was created using a bottom-up approach. This allowed a great deal of flexibility throughout the development. As algorithms were improved and/or optimized, the design was easily adaptable.

4.1 Hardware Implementation

The VHDL implementation consists of mapping algorithms discussed in the previous sections to hardware functions and optimizing area and speed, where possible, while allowing user flexibility.

The hardware was organized into four functional design blocks (Fig. 1). The control block contains all the I/O interface circuitry and controls the flow of the digital signature algorithm. The remainder is used for modular reduction in the signature as well as in the pseudo-random number generation process. The SHA-1 hash function serves a dual purpose in hashing the input message and creating the pseudo-random number required for signature generation. The signature algorithm block controls and performs the actual signing of the message.

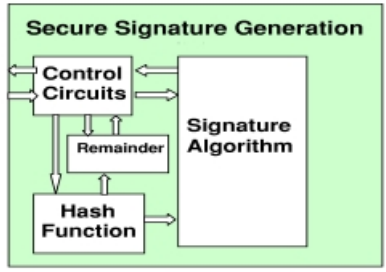

Fig. 1. Top Level Architecture.

4.2 Command, Configuration, and Control

The command, configuration, and control circuitry is responsible for all the high-level control and configuration of the device. It controls the external interface

to the chip, message input and signature output, random number generation control, power management, and algorithm flow control.

The external I/O interface to this chip is intended to hang off of a microprocessor bus. There is a 16-bit address bus, an 8-bit data bus, and control signals. The device is intended to be used as a memory-mapped device in which communication to the device is via a read and write interface similar to that of random access memory (RAM). In addition, there are interrupt signals that are used to indicate to the host system signature status, error status, and signature completion (Fig. 2).

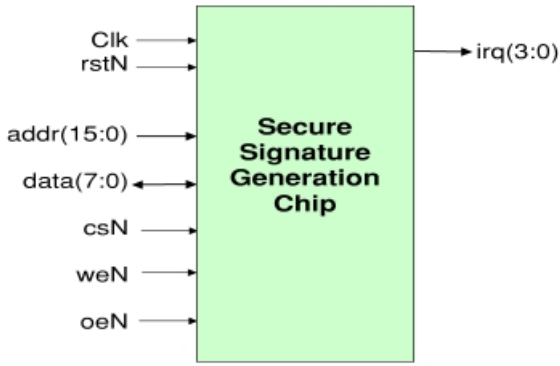

Fig. 2. Secure Signature Generation Chip Interface

The architecture gives the end user a great deal of flexibility. The device can be used in conjunction with any microprocessor that contains a memory-mapped interface. Within the chip, there is a memory map for a full suite of initialization, configuration, result, and status registers. In this respect, the aspects of the signature algorithm are programmable. The following parameters can be programmed (i.e. written) into the Secure Signature Generation Chip.

- Message (up to 512 bits at a time) or Message Digest (based on configuration)
- Generating Point on the Elliptic Curve, $G = (x_G, r_G)$
- Order, r, of the Point G
- Private Key, s
- Random Number (178 bits) or Random Seed (320 bits) (based on configuration), used to generate the per-message nonce
- Configuration Variables (message format, random number format, sleep mode, 1st Message, etc.)

In addition, the signature algorithm generates a set of variables that can be polled (i.e. read).

- Public Key Output
- Output Signature (c, d)

The power management circuitry (in the control section) is essentially a clock-gating circuit that controls when a certain functional area is receiving a clock. The power management is used on a function-by-function basis. That is, the clock-gating follows the circuit function, and, when a circuit is not calculating a value (i.e. idling), the clock to that respective circuit is disabled. This logic is used to reduce overall power consumption by controlling the switching capacitance of respective functional blocks.

4.3 Random Number Generation

There are two methods for generating the per-message nonce needed for the El Gamal signature generation. The first is to simply input the random number via the I/O memory mapped interface. This allows the user to use a true-random number if so desired, but has the obvious overhead of needing to input that random number for each message to be signed.

The second approach is to use the on-chip pseudo-random number generator. This method follows the updated pseudo-random number generation algorithm of the Digital Signature Standard [15]. This circuit uses the remainder circuit and the hash function to create the pseudo-random number. The methodology is to use two 160-bit seeds to create two independent 160-bit hash values. These values are fed back into the random seed registers for further creation of pseudo-random numbers. They are then concatenated together to produce a 320-bit value, from which the remainder (mod r) is extracted. This value is then used as a 178-bit random number.

4.4 Message Input

There are two methods for message input. The user can configure the device to accept a 160-bit message digest. This allows the user to generate the hash of the message and input the message digest via the memory-mapped interface. The hashing overhead would be under user control.

Alternatively, the user can have the on-chip circuitry hash a raw input message using SHA-1. The SHA-1 VHDL was implemented per FIPS Standard [16] and computes a 160-bit message digest from the incoming message.

4.5 Signature Algorithm

The VHDL implementation of the Elliptic Curve Optimal El Gamal Signature Algorithm is a direct implementation of the algorithm described in Section 2. As with the full-chip implementation, the control circuit is responsible for operation of the algorithm and data flow between the various blocks. The multiply and remainder functions exist to compute the products and modular reductions needed in the signature. They are both simple ripple/shift implementations of the mathematical operations. The block diagram is shown in Fig. 3.

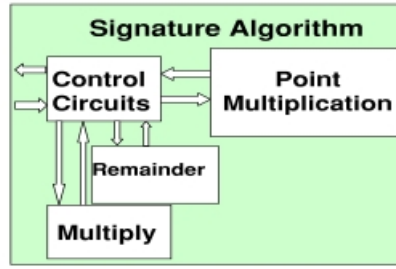

Fig. 3. Signature Algorithm Architecture

4.6 Hardware Optimizations

There were several design optimizations that were used to improve area and performance. Some of the more prominent and significant improvements are discussed below.

For multiplication in a finite field (Section 3.1), which operates with a simple shift and add, the radix of the multiply was increased to 16. This allowed us to perform the multiply in 4-bit fragments, which provided a dramatic speed increase with a slight area penalty. This operation was a bottleneck in the design, thus this improvement provided a speed-up of about a factor of two.

In the Almost Inverse Algorithm ([8], p.50), there were three optimizations that were implemented. The 1st is a parallel degree comparator circuit, which was optimized for both area and speed. The 2nd optimizes the search for a 1 in the LSB of a variable by using a "look ahead" technique with 4-bit blocks before defaulting to operating on the data 1 bit at a time. This increased speed with a very small area penalty. In a similar manner, the 3rd optimization is applied to the last step in the algorithm, which divides and shifts the result a variable number of times. This too performs a "look ahead" using 8-bit data blocks before defaulting to the single-bit implementation.

The Qsolve Algorithm 5 was parallelized and the depth of the XOR tree has been reduced to increase the speed as described in Section 3.2.

The implementation of the SHA-1 algorithm has been optimized to use a shift register for the main data storage, which reduced the area used, with a corresponding increase in speed.

5 Hardware Design Results

This design has not been realized in silicon. However, the design has been synthesized to a target CMOS $0.5\mu m$, $5V$ library. It has also undergone static timing analysis, timing simulations, and power analysis. The following is a summary of results for this target library.

Signature Generation Time Using a 20 Mhz System Clock:

- Initialization: 0.25 ms
  (Necessary any time the Generating Point is initialized and/or changed)
- Signature Generation: 4.4 ms

Synthesized Gate Count Approximations (target library $0.5\mu m$, $5V$, $25°C$) of major sub-blocks, where a gate is equivalent to a standard library NAND Cell.

- Chip: 191,000 Gates
  - Control: 27,000 Gates
  - SHA-1: 13,000 Gates
  - Remainder: 6,700 Gates
  - Signature Algorithm: 143,000 Gates
    - Control: 15,000 Gates
    - Multiply: 6,200 Gates
    - Remainder: 6,800 Gates
    - Point Multiplication: 112,000 Gates
      - Register & Control: 30,000 Gates
      - Point Addition: 52,000 Gates
      - Point Halving: 29,000 Gates

Critical Timing Path (Setup Timing) (target library $0.5\mu m$, $5V$, $25°C$):

- Critical Setup Timing Path (register to register): 48 ns

The critical timing path is located at the Signature Algorithm Level in the computation of $cfs + u \pmod r$. Specifically, it is located in the subtract circuit within the Remainder that computes the modulo r value for the signature. Optimizations are still being performed to improve timing critical paths that affect the overall performance of the device.

Power Analysis and Estimation Using 20 Mhz System Clock (target library $0.5\mu m$, $5V$, $25°C$):

- Dynamic Power Consumption Estimation: 150 mW
- Static (Idle) Power Consumption Estimation: 6 μW

The above numbers were generated using the Synopsys Power Compiler (power analysis tool) which uses gate switching data (based on typical simulation results) to generate power estimates. These estimates are library dependent and are based on the accuracy of the library models provided.

Please note that these design results (performance/speed, gate count, and power estimation) are only applicable to the target hardware process technology, which is not the most advanced technology available today, but was the

most accessible and complete for this analysis. If one were to target a more advanced technology, the design would certainly improve in performance (speed), area (gate count), and power consumption. Specifically, the Critical Setup Timing Path could significantly improve, thus improving the overall speed of the chip. Using power $P = V^2/R$, where V=operating voltage and R=operating resistance, which is fixed, lowering V from $5V$ to $3.3V(1.8V)$ would result in a 56%(87%) reduction in power consumption. At $1.8V$, the estimated dynamic power consumption is $19mW$.

6 Conclusions

Low-power hardware implementations of public-key cryptography continue to enable its use in resource-constrained environments. Wireless applications alone will further drive this market. In this paper, our VHDL design takes advantage of several optimizations of both finite field and elliptic curve arithmetic for the specific function of digital signature generation. We use hardware techniques to reduce the overall power consumption by switching the clock off to areas that are not currently being used. This reduces the power by reducing the effective switching capacitance of the clock. Our design has been successful in achieving performance attributes that are attractive to low-power applications requiring strong public-key authentication. Opportunities to further develop optimized implementations of elliptic curve-based signature algorithms include the following.

1. Further utilization of extension fields.
2. Additional improvements to point multiplication.
3. Improvement of the worst case setup timing path.

Finally, since our main focus was minimizing power consumption, we note that we have ignored the problem of side channel attacks. Countermeasures against such attacks are important and should be the subject of future work. Under the auspices of technology transfer, anyone interested in employing our current and future developments in their application is encouraged to contact the authors.

Acknowledgements. The authors would like to thank Mark Torgerson for many useful discussions and comments.

References

1. Kim, C., Oh S., and Lim, Jongin, "A new hardware architecture for operations in GF(2^n)", IEEE Transactions on Computers v 51 n 1 January 2002. p. 90–92.
2. S. Moon, J. Park, and Y. Lee, "Fast VLSI arithmetic algorithms for high-security elliptic curve cryptographic applications", Proceedings of ICCE. International Conference on Consumer Electronics, 19-21 June 2001, Los Angeles, CA.

3. J. Goodman and A. Chandrakasan, "An energy-efficient reconfigurable public-key cryptography processor", IEEE Journal of Solid-State Circuits, vol.36, no.11, p. 1808–20. Feb. 2001, San Francisco, CA.
4. M. Aydos, T. Yanik, and C. Koc, "High-speed implementation of an ECC-based wireless authentication protocol on an ARM microprocessor", in IEEE Proceedings-Communications, vol.148, no.5, p. 273–9, Oct. 2001.
5. Nyberg, K. and R. A. Rueppel, "Message recovery for signature schemes based on the discrete logarithm problem", Advances in Cryptography – Eurocrypt '94, Springer LNCS 950, 1994, p. 182–193.
6. L. Harn and Y. Xu, "Design of Generalised El Gamal Type Digital Signature Schemes Based on Discrete Logarithm", in Electronics Letters Online, No.19941398, September 30, 1994. IEEE.
7. J. Silverman, The Arithmetic of Elliptic Curves. Springer-Verlag, 1986.
8. R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems", in Advances in Cryptology – Crypto '95, Springer LNCS 963, 1995, p. 43–56.
9. R. Schroeppel, "Elliptic Curves – Twice as Fast", Midwest Algebraic Geometry Conference, Urbana, IL, November 2000.
10. E. Knudsen, "Elliptic Scalar Multiplication Using Point Halving", Advances in Cryptology – Asiacrypt '99, Springer LNCS 1716, 1999, p. 135–149.
11. IEEE P1363, Standard Specifications for Public Key Cryptography. Appendix A, 1997.
12. N. Smart, "How Secure Are Elliptic Curves over Composite Extension Fields?", in Eurocrypt 2001, LNCS 2045, May 2001, p. 30–39.
13. Schroeppel, R. "Circuits for Solving a Quadratic Equation in $GF[2^N]$", in preparation, 2002.
14. M. Rosing, Implementing Elliptic Curve Cryptography, Manning Publications, 1999.
15. FIPSPUB 186-2 + change notice 1.
16. FIPSPUB 180-1.