Security-Performance Trade-offs in Embedded Systems Using Flexible ECC Hardware

Hamad Alrimeih and Daler Rakhmatov
University of Victoria

Editor's note:
A popular approach to efficient implementation of cryptographic algorithms is to use coprocessors that implement selected computations in conjunction with an embedded processor. This article presents a detailed trade-off between security and performance in the design of such a coprocessor for elliptic-curve cryptography.
—Anand Raghunathan, NEC Laboratories America

ELLIPTIC-CURVE CRYPTOGRAPHY (ECC) offers an attractive mechanism for protecting information in resource-constrained embedded systems. In comparison with other public-key cryptosystems, ECC achieves equivalent security using fewer bits, which translates into smaller and faster hardware and software implementations.[1] The main ECC computation is the scalar multiplication $Q = kP$, where k is a positive integer, and P and Q are points on a properly chosen elliptic curve. The ''Example 1'' sidebar illustrates an ECC protocol. ECC protocols rely on the computational hardness of finding scalar k (private value), given points P and Q (public values). However, practical implementations of ECC can compromise its theoretical strength by ignoring countermeasures to side-channel attacks. In other words, an ECC application's mathematical security is necessary but not sufficient to achieve an actual realization's physical security.

In addition to being secure, ECC implementations must offer adequate performance, especially in deadline-sensitive embedded computing. For that purpose, an embedded system should include hardware accelerators for handling time-consuming ECC computations. These ECC accelerators should support multiple security and performance levels, allowing the system to adjust its security-performance settings to application-specific needs.

Increasing security might require extra computations that adversely affect performance. For example, a longer key provides extra mathematical security but might require more processing time. Also, masking computations (for example, redundant operations decorrelating the system's execution profile from the value of a key) provide extra physical security but might add to the system's latency. System software can include algorithms that explore these trade-offs to strike a balance between security and performance.

This article presents a study of our system prototype, which features a flexible hardware processor that accelerates ECC computations. The ECC processor is controlled by system software executed by the main processing element, called the supervisor. Our system supports ECC over prime fields $GF(p)$ recommended by the National Institute of Standards and Technology (NIST).[2] There are five $GF(p)$, whose prime sizes are 192, 224, 256, 384, and 521 bits. Our ECC processor is programmable: It can work with any of the five NIST primes, and it can automatically execute short sequences of modular operations. Thus, the system can vary both its mathematical security (by changing the prime size) and its physical security (by changing the number and type of masking operations). We propose several execution scenarios with different security-performance characteristics. These

0740-7475/07/$25.00 © 2007 IEEE Copublished by the IEEE CS and the IEEE CASS **IEEE Design & Test of Computers**

scenarios involve both the supervisor (software) and the ECC processor (hardware); that is, we rely on a mixed hardware-software approach. Although software-controlled hardware accelerators are a common tool for improving performance, previous research has not considered them for flexible balancing of security and performance.

Software-only ECC implementations are very flexible, but they are inefficient in terms of performance.[1] Specialized hardware accelerators are very efficient in terms of performance, but they are inflexible.[3–5] Several systems offer good performance while allowing a certain degree of flexibility during ECC computations.[6–10] However, they don't support all five NIST primes and don't address security-performance tradeoffs arising from hardware accelerator programmability. Our work attempts to fill this gap.

ECC background

Much of the material of this section is based on Hankerson, Menezes, and Vanstone's book.[1] We describe an elliptic curve over $GF(p)$ using equation E_p: $y^2 = x^3 + ax + b$. NIST recommends five such curves, with parameter $a = -3$ and different primes p. The NIST *Digital Signature Standard* provides the values of parameter b.[2] Point P belongs to E_p if its coordinates (x_P, y_P) satisfy the curve equation. If $P = (x_P, y_P) \in E_p$, negated point $-P = (x_P, -y_P)$ belongs to E_p as well. In addition to affine coordinates (x_P, y_P), point P can also be expressed using projective coordinates. For example:

- Standard projective point (x_P, y_P, z_P) corresponds to affine point $(x_P z_P^{-1}, y_P z_P^{-1})$, where $z_P \neq 0$. Converting from standard projective coordinates to affine coordinates requires one modular inversion z_P^{-1} and two modular multiplications, $x_P z_P^{-1}$ and $y_P z_P^{-1}$. Affine point (x_P, y_P) corresponds to standard projective point $(x_P z_P, y_P z_P, z_P)$. If we let $z_P = 1$, the conversion from affine to standard projective coordinates becomes trivial.
- Jacobian projective point (x_P, y_P, z_P) corresponds to affine point $(x_P z_P^{-2}, y_P z_P^{-3})$, where $z_P \neq 0$. Converting from Jacobian projective coordinates to affine coordinates requires one modular inversion z_P^{-1} and four modular multiplications, $(z_P^{-1})^2$, $x_P(z_P^{-1})^2$, $(z_P^{-1})^3$, and $y_P(z_P^{-1})^3$. Affine point (x_P, y_P) corresponds to Jacobian projective point $(x_P z_P^2, y_P z_P^3, z_P)$. If we let $z_P = 1$, the conversion from affine to Jacobian projective coordinates becomes trivial.
- Chudnovsky projective point $(x_P, y_P, z_P, u_P, v_P)$ corresponds to Jacobian projective point (x_P, y_P, z_P) with two redundant coordinates $u_P = z_P^2$ and $v_P = z_P^3$. Converting from Chudnovsky projective coordinates to affine coordinates is the same as converting from Jacobian projective coordinates to affine coordinates. Affine point (x_P, y_P) corresponds to Chudnovsky point $(x_P z_P^2, y_P z_P^3, z_P, z_P^2, z_P^3)$. If we let $z_P = 1$, the conversion from affine to Chudnovsky projective coordinates becomes trivial.

Scalar multiplication $Q = kP$ is the main ECC computation: We add point P to itself k times to obtain point Q. Figure 1 shows a standard algorithm for left-to-right scalar multiplication. It involves two types of point operations: point doubling $Q = 2Q$ and point addition or subtraction $Q = \pm P + Q$. In affine

Example 1: Elliptic-curve Diffie-Hellman (ECDH) key exchange protocol

Alice and Bob want to agree on a common private key to use for encrypted data exchange. The only communication means between them is insecure public channels. To achieve their goal in such a setting, Alice and Bob can follow the ECDH protocol:[1]

1. Alice and Bob agree on elliptic curve E (over a finite field) and point P on the curve. These parameters are public and must be chosen carefully.
2. Alice picks secret integer k_a, computes $Q_a = k_a P$, and sends Q_a to Bob.
3. Bob picks secret integer k_b, computes $Q_b = k_b P$, and sends Q_b to Alice.
4. Alice computes $Q_{ab} = k_a Q_b = k_a k_b P$, while Bob computes $Q_{ba} = k_b Q_a = k_b k_a P$. Points Q_{ab} and Q_{ba} are equivalent.
5. Alice and Bob extract the private key using an agreed method such as letting the x-coordinate of Q_{ab} be the key value.

Reference

1. D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004.

```
INPUT: Point P $\neq P_\infty$ scalar $k$ with no leading zeros
OUTPUT: Point $Q = kP$
   1. Let $Q = P$
   2. FOR $l = |k| - 2,\ldots, 1, 0$:
      2.1. Let $Q = 2Q$ //Point doubling
      2.2. IF $k_l = 1$ THEN let $Q = P + Q$ //Point addition
      2.3. IF $k_l = -1$ THEN let $Q = -P + Q$ //Point
           subtraction, only for nonadjacent-form (NAF) $k$
   3. RETURN $Q$
```

Figure 1. Left-to-right scalar multiplication algorithm.

coordinates, we define these operations as follows:

$$\begin{aligned} Q_A &= 2Q_A : \tilde{x}_Q = \lambda^2 - 2x_Q, \\ \tilde{y}_Q &= \lambda(x_Q - \tilde{x}_Q) - y_Q, \text{ where} \\ \lambda &= \left(3x_Q^2 - 3\right)(2y_Q)^{-1} \\ Q_A &= P_A + Q_A : \tilde{x}_Q = \lambda^2 - x_P - x_Q, \\ \tilde{y}_Q &= \lambda(x_P - \tilde{x}_Q) - y_P, \text{ where} \\ \lambda &= (y_Q - y_P)(x_Q - x_P)^{-1} \\ Q_A &= -P_A + Q_A : \tilde{x}_Q = \lambda^2 - x_P - x_Q, \\ \tilde{y}_Q &= \lambda(x_P - \tilde{x}_Q) + y_P, \text{ where} \\ \lambda &= (y_Q + y_P)(x_Q - x_P)^{-1} \end{aligned}$$

In these equations, we use the embellishment $\sim$ to differentiate the new coordinate values of Q (left-hand side) from the old coordinate values of Q (right-hand side). The "Example 2" sidebar illustrates an affine point operation.

Affine point operations require expensive modular inversions. We can avoid them by using projective coordinates. For example, Figure 2 shows

- Jacobian point doubling $Q_J = 2Q_J$ (Figure 2a),
- affine-Jacobian point addition or subtraction $Q_J = \pm P_A + Q_J$ (Figure 2b), and
- Chudnovsky-Jacobian point addition or subtraction $Q_J = \pm P_C + Q_J$ (Figure 2c).

Compared with alternatives, these inversion-free choices require the fewest modular multiplications. Note that Q is always Jacobian, whereas P can be either affine or Chudnovsky. Using projective coordinates eliminates modular inversions inside the FOR loop in Figure 1 (step 2). However, before and after this loop, we must perform appropriate point conversions because input P and output Q are always affine points.

In addition to deciding on the point representation, we must also decide on the representation of scalar k. Here, we consider only binary and nonadjacent-form (NAF) representations among various alternatives.[1] Let k_B and k_N denote binary and NAF k, respectively. Any k_B corresponds to unique k_N, where each digit is either 0, 1, or -1. We can generate a sequence of NAF digits $r \in \{-1, 0, 1\}$ by repeatedly dividing k_B by 2. If k_B is even, $r = 0$; otherwise, we choose the value of r so that the quotient $(k_B - r)/2$ is even. For example, 8-bit binary $k_B = (11010111)_2$ corresponds to 9-digit NAF $k_N = (100\underline{1}0\underline{1}00\underline{1})_3$, where $\underline{1}$ denotes -1. The advantage of NAF is that no two consecutive digits are nonzero, which reduces the number of point additions or subtractions during scalar multiplication. Regardless of the scalar representation, we assume that all leading zeros (if any exist) are removed to avoid unnecessary computations.

As we add point P to itself k times, we obtain $(k - 1)$ new points $Q = 2P; 3P; \ldots; kP$ on curve E_p. For NIST curves, these points (including P) form a cyclic group generated by P. Hence, there exist k such that $(k + 1)P = P$; that is, we start repeating points once k becomes sufficiently large. Let curve order $\#E_p$ denote the smallest value of k such that $(k + 1)P = P$. Then $\#E_pP = (\#E_p + 1)P - P = P - P = 0P = -P + P = (\#E_p - 1)P + P$. We denote this special $0P$ point as P_∞. It serves as an additive identity: $P \pm P_\infty = P$. As long as k is between 2 and $(\#E_p - 2)$, we never encounter Q equal to P, $-P$, or P_∞ during scalar multiplication kP.

Security considerations

Figure 2 shows that the computational requirements of point doubling and point addition or subtraction are different, leading to different execution profiles of the system. An attacker can possibly identify a sequence of point doublings and point additions or subtractions by analyzing a system's power trace. Then the attacker can derive the private scalar value by correlating point additions or subtractions to nonzero bits or digits of k. Even if we take countermeasures against these single-trace simple attacks, the system might still be vulnerable to multitrace differential attacks. The attacker can manipulate input P, collect power traces of several scalar multiplications kP, and then use probabilistic techniques to correlate differences between power traces to the value of k. Blake, Seroussi, and Smart's book provides further details on various side-channel attacks.[11]

To protect against attacks, we must decorrelate the system's latency and power consumption from the scalar value during scalar multiplications. The following techniques are common countermeasures:[11]

- Masking bits or digits of scalar k by introducing redundant modular operations to make point doubling and point addition or subtraction indistinguishable. This protects k against simple attacks.
- Randomizing point P by letting affine point $P = (x_P, y_P)$ become randomized Chudnovsky point $P = (x_P z_P^2, y_P z_P^3, z_P, z_P^2, z_P^3)$, where z_P is a proper random integer between 1 and $(p - 1)$. This protects k against certain differential attacks. The exceptions are attacks that use zero-value points $(0, y_P)$ or $(x_P, 0)$, for which randomization does not work.
- Randomizing scalar k by letting $kP = iP + (k - i)P$, where i is a proper random integer between 1 and $(k - 1)$. This protects k against all differential attacks. Although we can randomize k without randomizing P, a general rule of thumb is to randomize both for better security.[11]

We assume that the system uses only these three techniques to increase its physical security. (Other advanced countermeasures exist,[11] but we don't consider them in this article.) The system can also increase its mathematical security by switching to a larger NIST prime p (if applicable) at the cost of more expensive modular operations. For a given prime p, we define four levels of physical security Γ_p, as shown in Table 1. Level 0 corresponds to an insecure implementation, and level 3 corresponds to the most secure implementation under our assumptions. For each Γ_p, there are also five levels of mathematical security associated with the five possible sizes of prime p.

We must mask scalar k first, before proceeding to more-advanced countermeasures. Hence, we express point operations using indistinguishable atomic blocks comprising several modular operations.[11] Figures 3a and 3b show such point operations with affine P and Jacobian Q. Introduced redundant operations are modular additions (shown in small boxes). Each atomic block includes one modular multiplication followed by two modular additions or subtractions. (Modular additions and subtractions are usually indistinguishable in latency and power consumption.) Using larger or smaller atomic blocks will introduce redundant modular multiplications, which are more expensive than modular additions. Figures 4a and 4b show atomic point operations with Chudnovsky P (after randomizing affine P) and Jacobian Q. Each atomic block is now double-sized, consisting of two modular multiplications and four modular additions or subtractions.

Example 2: Affine point addition over *GF*(*p*)

Let $p = 5$, and E_5: $y^2 = x^3 - 3x + 3$ over prime field $GF(5)$. Assume that $P_A = (1, 4)$ and $Q_A = (3, 1)$. To find new

$$Q_A = P_A + Q_A = (\tilde{x}_Q, \tilde{y}_Q)$$

we perform the following computations:

$$\lambda = (1 - 4)(3 - 1)^{-1} = (-3)2^{-1} = (-3)3$$
$$= -9 = 1 \bmod 5$$
$$\tilde{x}_Q = 1^2 - 1 - 3 = -3 = 2 \bmod 5$$
$$\tilde{y}_Q = 1(1 - 2) - 4 = -5 = 0 \bmod 5$$

The allowable values in $GF(5)$ are {0, 1, 2, 3, 4}. Hence, $\lambda = -9$ has been reduced to $\lambda = -9 + 5 + 5 = 1 \bmod 5$. We can perform modular reduction by repeatedly adding or subtracting prime p until we obtain an integer in $GF(p)$. Note that the inverse of 2 in $GF(5)$ is 3. By definition, the product of a number and its inverse must yield 1; indeed, the product of 2 and 3 is $6 = 1 \bmod 5$. Thus, new $Q_A = (2, 0)$. All three points—(1, 4), (3, 1), and (2, 0)—lie on the curve E_5: $y^2 = x^3 - 3x + 3$ over $GF(5)$.

Execution environment

Figure 5a shows our target system, which performs ECC-related computations using both software (executed by the supervisor) and hardware (based on the ECC processor). The supervisor dynamically programs the processor by loading appropriate instructions into the processor memory. Each instruction encodes a modular operation that can use any of the five NIST primes shown in Figure 5b. The supervisor groups these instructions into simple programs for performing complete point doubling and point addition or subtraction operations and schedules these programs for execution on the ECC processor during scalar

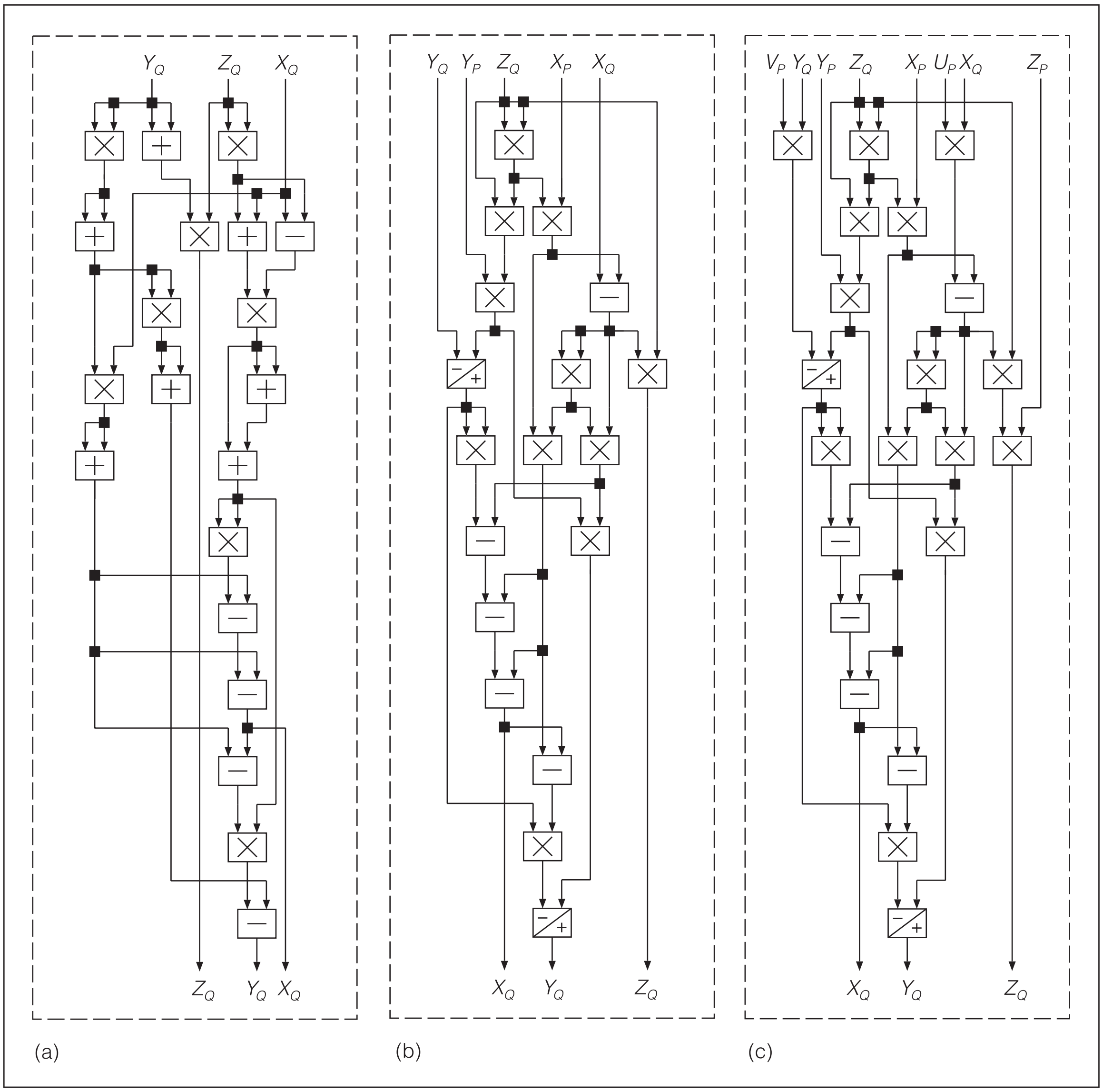


Figure 2. Point operations with affine or Chudnovsky *P* and Jacobian *Q*: $Q_J = 2Q_J$ (a); $Q_J = \pm P_A + Q_J$ (b); $Q_J = \pm P_C + Q_J$ (c). Boxes with [−/+] represent [−] for +*P* and [+] for −*P*.

multiplications. The supervisor can change the system's physical and mathematical security by changing the processor programming.

Table 1. Four levels of physical security Γ_p.

Security level Γ_p	Masked scalar *k*	Randomized point *P*	Randomized scalar *k*	Scalar protection
0	No	No	No	None
1	Yes	No	No	Poor
2	Yes	Yes	No	Good
3	Yes	Yes	Yes	Excellent

Figure 5c specifies the processor instructions. Table 2 shows the number of clock cycles required by each instruction. The table also shows the number of clock cycles the supervisor requires to write an instruction or access a point coordinate in the processor memory. The time it takes to perform a specific instruction depends on the prime used. Inversion is the most expensive operation, and its latency is data dependent (Table 2 shows the upper bound).

We achieve processor flexibility by supporting programmable point operations (doubling, addition, and subtraction) and by supporting the five NIST prime fields. We can change a point operation by downloading a different sequence of processor instructions. We can change a prime by modifying the opcode field of the instructions to be executed. When the prime changes, the internal data path controllers adjust their control-signal sequences accordingly. Ananyi and Rakhmatov present further details on the processor design.[12]

Figure 5d shows an example of how the supervisor maps instructions and data to processor memory locations. Let Memory[T] denote the Tth location of the processor memory, where $T = 0, 1, \ldots, 511$. Also, let L denote the total number of programs residing in processor memory, and let Program[l] denote the (l)th program, where $l = 1, 2, \ldots, L$. Memory[0] contains instruction **`jump(?)`**, where the jump target **`(?)`** can be any program pointer among $T_1, T_2, \ldots, T_L$. The processor memory contains not only instructions but also data: coordinates of P and Q, as well as temporary variables and coordinates of intermediate point R (if needed). We specify temporary variables' addresses by using appropriate offsets from data pointers T_P, T_Q, and/or T_R. The memory requirements for data depend on the size of the NIST prime used; for example, for prime $p256$, each coordinate of P requires 8 words.

To read or write to a processor memory location, the supervisor must generate *supervisor* = 1. Data wires *d_in* and *d_out* carry the 32-bit memory word addressed by *addr* during read/write (WE = 0/1). While *supervisor* = 1, the processor must not access its memory. Once *supervisor* = 0 and *globalstart* = 1, the processor asserts *busy* = 1 and starts executing instructions stored in its memory. The *globalstart* signal becomes zero in response to the assertion of *busy*. As long as *busy* = 1, the supervisor must not access processor memory. When the processor encounters a **`stop`** instruction, it deasserts *busy* and enters an idle mode. The processor's program counter is reset to 0 when it receives *globalreset* = 1.

In the memory mapping shown in Figure 5d, the processor executes only one program at a time before encountering a **`stop`** instruction. The supervisor must restart the processor to continue. If the supervisor doesn't reset the processor before restarting it, the next executed program will be the one following the last executed **`stop`** instruction. If the supervisor resets the processor before restarting it (thus clearing its program counter), the processor jumps to the program targeted by **`jump(?)`**. As an option, we can let the last instruction following the Lth stop be **`jump(0)`**, which can serve as a self-reset.

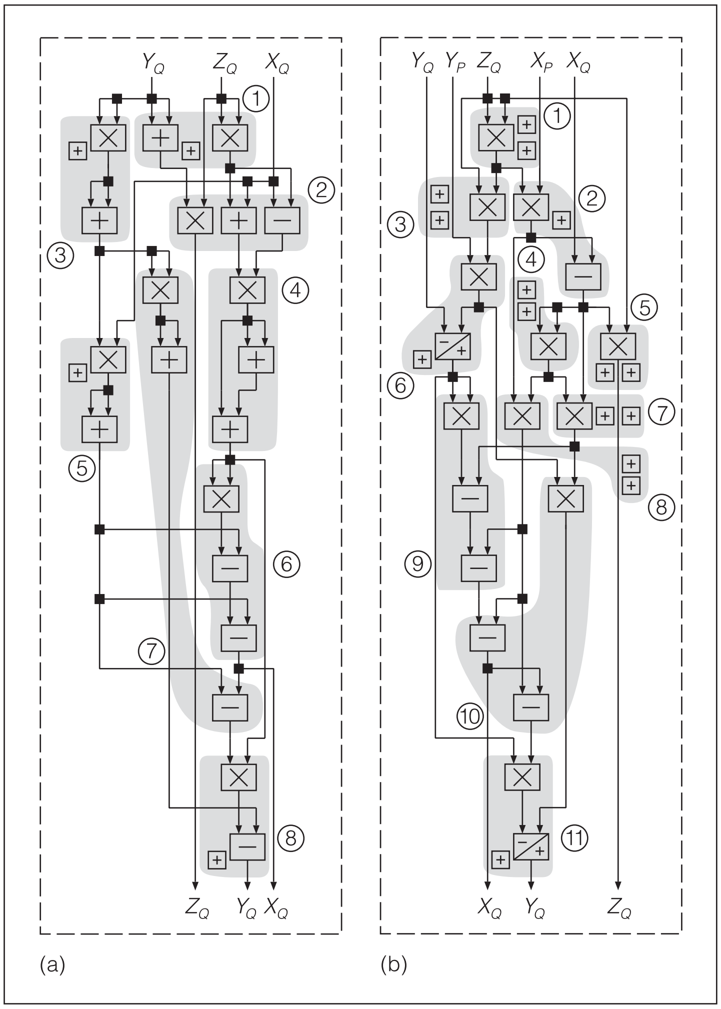


Figure 3. Affine-Jacobian atomic point operations: $Q_J = 2Q_J$ (a) and $Q_J = \pm P_A + Q_J$ (b).

While waiting for the processor to finish executing its program, the supervisor can switch to other tasks, including non-ECC applications (if present). Once the processor reaches a **`stop`** instruction, it signals the completion of program execution by deasserting *busy*. This event can serve as an interrupt request for the supervisor. The supervisor responds to such an interrupt by resuming its control over the current ECC application.

Proposed execution scenarios

Given scalar k and affine point P on elliptic curve E_p over prime field $GF(p)$, we want to find affine point $Q = kP$. NIST specifies five different curves E_p and their orders $\#E_p$. We assume that $2 \leq k \leq (\#E_p - 2)$ and consider four representations of k. Two of them, binary k_B and NAF k_N, were described earlier. We propose the

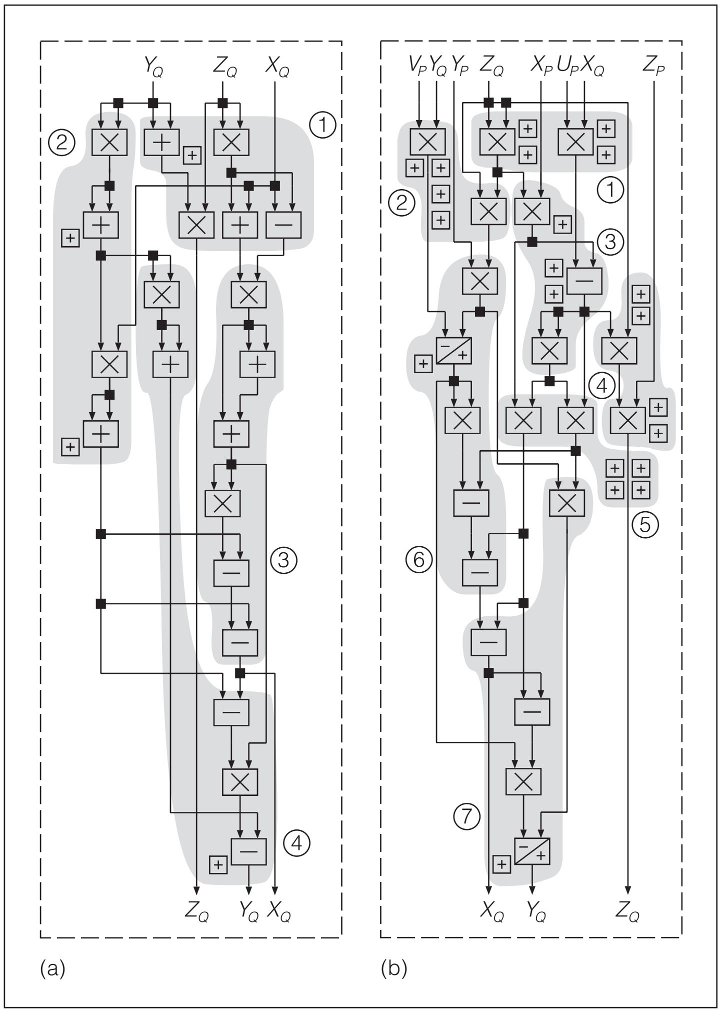


Figure 4. Affine-Chudnovsky atomic point operations: $Q_J = 2Q_J$ (a) and $Q_J = \pm P_C + Q_J$ (b).

other two representations of k based on the following idea:

$$\begin{aligned} kP &= kP - P_\infty = kP - \#E_pP \\ &= -\#E_pP + kP \\ &= -\left(\#E_p - k\right)P = -\widehat{k}P \end{aligned}$$

where $\widehat{k}$ denotes $(\#E_p - k)$. Thus, in addition to k_B and k_N, we also consider the binary and NAF representations of $\widehat{k}$, with the leading zeros removed (if any). Computing $-\widehat{k}P$ may be faster than computing kP in some cases. To find $Q = -\widehat{k}P$, we first find $(x_Q, y_Q) = \widehat{k}P$ and then negate y_Q by subtracting it from prime p.

For different security levels Γ_p, we must execute different programs. Table 3 summarizes our 15-program library for given prime p. For example, if $\Gamma_p = 0$, we use Library$_p[1:3]$ to perform the point operations shown in Figure 2 and then either Library$_p[14]$ or Library$_p[15]$, depending on whether $Q = kP$ or $Q = -\widehat{k}P$, to convert Jacobian Q to affine Q. If $\Gamma_p = 1$, we use Library$_p[4:6]$ to perform the point operations shown in Figures 3a and 3b. If P is randomized (that is, $\Gamma_p > 1$), we use Library$_p[7]$ to convert affine P with random integer z_P to Chudnovsky P, and we use Library$_p[8:10]$ to perform the point operations shown in Figures 4a and 4b.

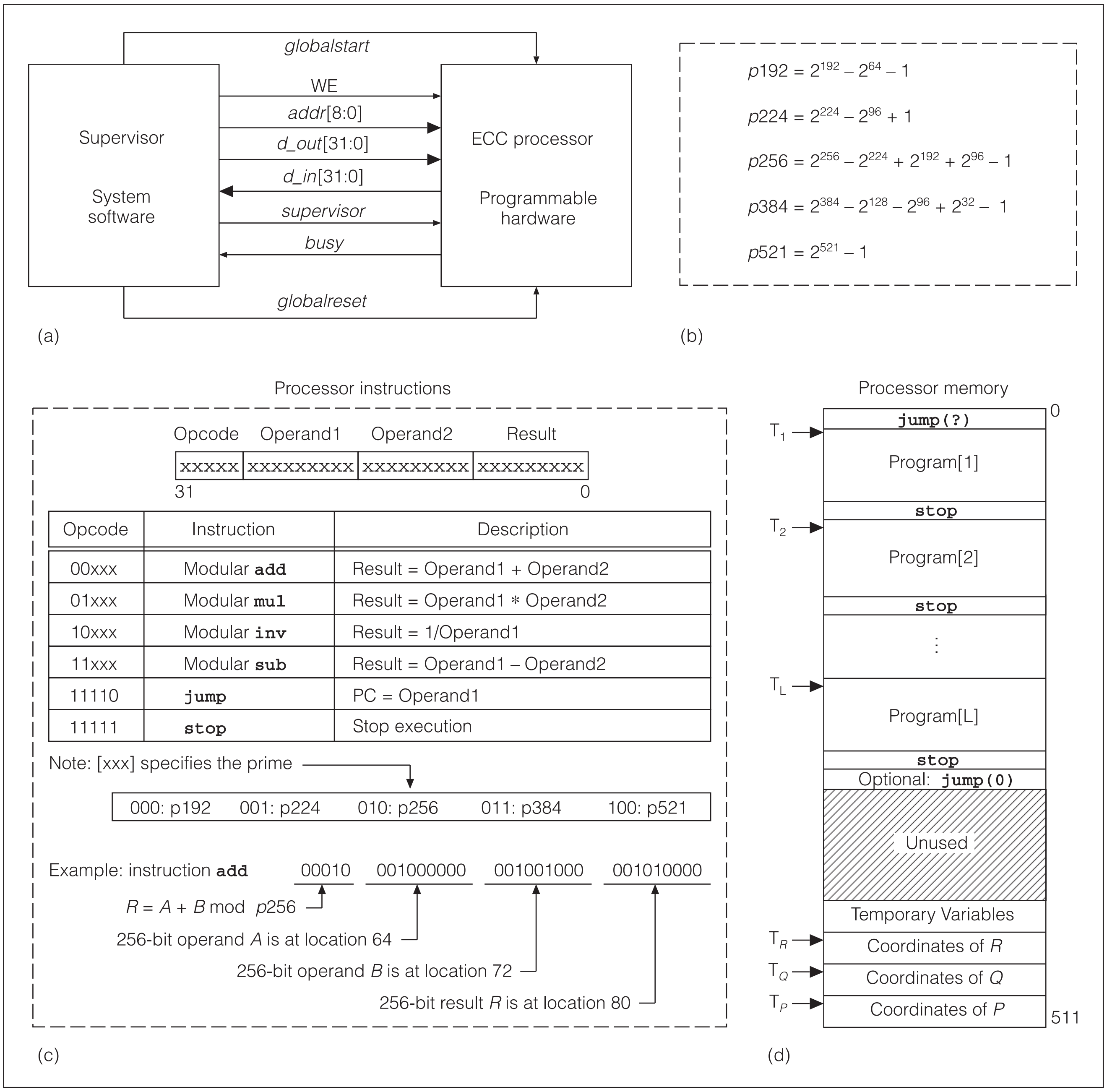
If k is also randomized ($\Gamma_p = 3$), we must perform two scalar multiplications $R = iP$ and $Q = (k - i)P$, followed by point addition $Q = R + Q$, where R is an intermediate point, and i is a random integer. First, we compute $Q = iP$, using Library$_p[8:10]$. Second, we convert Jacobian Q to Chudnovsky R (for later point addition), using either Library$_p[11]$ or Library$_p[12]$. Third, we compute $Q = (k - i)P$, again using Library$_p[8:10]$. Finally, we add Jacobian Q and Chudnovsky R, using Library$_p[13]$, as shown in Figure 4b, where P should be replaced with R.

Performance metric

We define our performance metric Ω_p as the total number of clock cycles the ECC processor uses during scalar multiplication. The value of Ω_p can change, depending on the following choices:

- *Scalar value*. Using a different scalar k can result in a different number of point operations per scalar multiplication.
- *Scalar representation*. Using different representations of k (k_B, k_N, $\widehat{k}_\text{B}$, or $\widehat{k}_\text{N}$) can result in a different number of point operations per scalar multiplication.
- *Security level*. Using different security levels Γ_p can result in a different number of modular operations per point operation.
- *NIST prime*. Using a different prime p (corresponding to a different curve E_p) can result in a different number of clock cycles per modular operation, as well as a different number of point operations per scalar multiplication (if $\widehat{k}$ is used).

We express our performance metric as function $\Omega_p(n, m, \Gamma_p)$, where n and m denote respectively the number of all bits or digits and the number of nonzero bits or digits of scalar k. Changing scalar value and/or scalar representation will change n and/or m, whereas Γ_p will reflect changes in the security level and/or NIST



Opcode	Instruction	Description
00xxx	Modular **add**	Result = Operand1 + Operand2
01xxx	Modular **mul**	Result = Operand1 * Operand2
10xxx	Modular **inv**	Result = 1/Operand1
11xxx	Modular **sub**	Result = Operand1 – Operand2
11110	**jump**	PC = Operand1
11111	**stop**	Stop execution

Figure 5. Embedded system with flexible elliptic-curve cryptography (ECC) processor: system setup (a), supported National Institute of Standards and Technology (NIST) primes (b), supported processor instructions (c), and processor memory map (d).

prime. Table 4 shows how we compute $\Omega_p(n, m, \Gamma_p)$, given our processor-programming algorithm (discussed later). (The "Example 3" sidebar shows the derivation of performance metric $\Omega_p(n, m, \Gamma_p)$ for NIST prime $p256$ and $\Gamma_p = 2$.)

If k is not randomized ($\Gamma_p < 3$), we must perform $(n - 1)$ point doublings and $(m - 1)$ point additions or subtractions during scalar multiplication (Figure 1). Randomized k can be expressed as $(i + j)$, where i is random and $j = k - i$. Let $n_{\{i\}}$ and $m_{\{i\}}$ denote respectively the number of all bits or digits and the number of nonzero bits or digits of scalar i. Also, let $n_{\{j\}}$ and $m_{\{j\}}$ denote respectively the number of all bits or digits and the number of nonzero bits or digits of scalar j. Since $kP = iP + jP$, we need $[(n_{\{i\}} - 1) + (n_{\{j\}} - 1)]$ point doublings and $[(m_{\{i\}} - 1) + (m_{\{j\}} - 1) + 1]$ point

Table 2. Number of clock cycles required by each instruction.

Processor instruction	No. of cycles for NIST prime				
(fetch, decode, execute)	*p*192	*p*224	*p*256	*p*384	*p*521
`add`	29	31	33	43	53
`sub`	29	31	33	43	53
`mul`	74	76	78	178	325
`inv`	3,090	3,604	4,118	6,174	8,376
`jump`	7	7	7	7	7
`stop`	7	7	7	7	7
Instruction write (supervisor)	2	2	2	2	2
Coordinate write or read (supervisor)	12	14	16	24	34

additions or subtractions. For $\Gamma_p = 3$ (last column of Table 4), we let $n = n_{\{i\}} + n_{\{j\}} - 1$ and $m = m_{\{i\}} + m_{\{j\}}$. Thus, randomized scalar multiplication becomes computationally similar to nonrandomized scalar multiplication with $(n - 1)$ point doublings and $(m - 1)$ point additions or subtractions.

Performance optimization under security constraints

Given Γ_p as our security constraint, we want to minimize our performance metric $\Omega_p(n, m, \Gamma_p)$. Figure 6 shows the corresponding algorithm for finding an optimal solution. For a given $\Gamma_p < 3$, we must consider only four choices of the scalar representation. Performance optimization is trivial—we simply examine all four possibilities and select the best one. We use $|X_Y|$ and $|X_Y{}^*|$ to denote respectively the number of all bits or digits and the number of nonzero bits or digits of X_Y, where X is either k or $\widehat{k}$, and Y is either binary (B) or NAF (N). Four possible $n = |X_Y|$ and $m = |X_Y{}^*|$ correspond to four possible $\Omega_p(n, m, \Gamma_p)$ for a given $\Gamma_p < 3$. The best X_Y becomes our scalar representation s, used to compute $Q = kP$. If we use $s = \widehat{k}_\mathrm{B}$ or $s = \widehat{k}_\mathrm{N}$, we must negate Q, which is indicated by flag $neg = 1$.

Given $\Gamma_p = 3$, we have two scalars i and j after randomizing k. Again, we examine all scalar representation choices and select the best one. We can have binary or NAF i with binary or NAF j. Alternatively, we can have binary or NAF $\widehat{i}$ with binary or NAF $\widehat{j}$. The best i or $\widehat{i}$ becomes our scalar representation s. The best j or $\widehat{j}$ becomes our scalar representation r. We let $\widehat{i} = \#E_p - \max(i, j)$ and $\widehat{j} = \min(i, j)$. This reduces the values of $\widehat{i}$ and $\widehat{j}$. If we use $\widehat{i}$ and $\widehat{j}$, we no longer negate Q. Instead, we negate the intermediate point $\widehat{i}P$ before adding it to the other intermediate point $\widehat{j}P$. In other words, $Q = -\widehat{i}P + \widehat{j}P = -[\#E_p - \max(i, j) - \min(i, j)]P = kP$.

Once the supervisor selects the best scalar representation, it programs the processor according to Table 5 and Figure 5d. Then the supervisor performs scalar multiplication using the algorithm shown in Figure 7. It scans scalar bits or digits and schedules the

Table 3. Program library.

Library$_p$[...]	Action	[No. of `mul`, `add`/`sub` instructions]	Description
[1]	$Q_J = 2Q_J$	[8, 12]	Jacobian point doubling
[2]	$Q_J = P_A + Q_J$	[11, 7]	Affine-Jacobian point addition
[3]	$Q_J = -P_A + Q_J$	[11, 7]	Affine-Jacobian point subtraction
[4]	$Q_J = 2Q_J$	[1, 2] × 8	8 atomic blocks for Jacobian point doubling
[5]	$Q_J = P_A + Q_J$	[1, 2] × 11	11 atomic blocks for affine-Jacobian point addition
[6]	$Q_J = -P_A + Q_J$	[1, 2] × 11	11 atomic blocks for affine-Jacobian point subtraction
[7]	$P_A \rightarrow P_C$	[2, 4] × 2	2 atomic blocks for affine-to-Chudnovsky point conversion
[8]	$Q_J = 2Q_J$	[2, 4] × 4	4 atomic blocks for Jacobian point doubling
[9]	$Q_J = P_C + Q_J$	[2, 4] × 7	7 atomic blocks for Chudnovsky-Jacobian point addition
[10]	$Q_J = -P_C + Q_J$	[2, 4] × 7	7 atomic blocks for Chudnovsky-Jacobian point subtraction
[11]	$Q_J \rightarrow R_C$	[2, 4] × 1	1 atomic block for Jacobian-to-Chudnovsky point conversion
[12]	$Q_J \rightarrow -R_C$	[2, 4] × 1	1 atomic block for negated Jacobian-to-Chudnovsky point conversion
[13]	$Q_J = R_C + Q_J$	[2, 4] × 7	7 atomic blocks for Chudnovsky-Jacobian point addition
[14]	$Q_J \rightarrow Q_A$	[4, (1)]	Jacobian-to-affine point conversion (dummy `sub`), also requires `inv`
[15]	$Q_J \rightarrow -Q_A$	[4, 1]	Negated Jacobian-to-affine point conversion, also requires `inv`

Table 4. Performance metric $\Omega_p(n, m, \Gamma_p)$.

NIST prime	No. of cycles for security level			
	$\Gamma_p = 0$	$\Gamma_p = 1$	$\Gamma_p = 2$	$\Gamma_p = 3$
*p*192	$963n + 1{,}040m + 1{,}665$	$1{,}079n + 1{,}475m + 1{,}182$	$1{,}079n + 1{,}871m + 1{,}447$	$1{,}079n + 1{,}871m + 1{,}870$
*p*224	$1{,}003n + 1{,}076m + 2{,}127$	$1{,}127n + 1{,}541m + 1{,}606$	$1{,}127n + 1{,}955m + 1{,}885$	$1{,}127n + 1{,}955m + 2{,}326$
*p*256	$1{,}043n + 1{,}112m + 2{,}589$	$1{,}175n + 1{,}607m + 2{,}030$	$1{,}175n + 2{,}039m + 2{,}323$	$1{,}175n + 2{,}039m + 2{,}782$
*p*384	$1{,}963n + 2{,}282m + 3{,}021$	$2{,}135n + 2{,}927m + 2{,}272$	$2{,}135n + 3{,}719m + 2{,}717$	$2{,}135n + 3{,}719m + 3{,}440$
*p*521	$3{,}259n + 3{,}969m + 2{,}908$	$3{,}471n + 4{,}764m + 1{,}969$	$3{,}471n + 6{,}057m + 2{,}621$	$3{,}471n + 6{,}057m + 3{,}708$

appropriate program by calling the scheduling subroutine shown in Table 6. During step 7 of the scheduling subroutine, the supervisor can switch to a different task while waiting for an interrupt by the deasserted *busy* signal. Finally, the supervisor reads the output data from processor memory.

Quantitative results

Changing security level Γ_p changes processor programming, which affects our performance metric Ω_p. In this section, we provide examples illustrating the quantitative relationship between Γ_p and Ω_p. We assume binary scalar k_B and let $n = |p|$ and $m = \lceil |p|/2 \rceil$, where $|p|$ is the prime size. This is a common assumption in the literature. In randomizing k, we let the sizes of the resulting two scalars be $(|p| - 1)$ each, with $\lceil (|p| - 1)/2 \rceil$ nonzero bits. Equivalently, $n = 2|p| - 3$ and $m = 2\lceil (|p| - 1)/2 \rceil$.

Table 7 shows our performance metric values Ω_p for four different security levels Γ_p and five different primes p. Traversing a column shows the impact of increasing physical security. Traversing a row shows the impact of increasing mathematical security. Comparing the table's first and last elements indicates our system's performance range (286,401 cycles versus 6,759,717 cycles). These results demonstrate a wide range of possible security-performance tradeoffs. Table 8 shows the delay incurred by control

Example 3: Calculating performance metric $\Omega_p(n, m, \Gamma_p)$

Let the scalar representation be n-bit binary $\widehat{k}_B$ with m nonzero bits, and let $\Gamma_p = 2$. We want to find Ω_p for NIST prime *p*256. According to Figure 1, we must execute $(n - 1)$ point doublings and $(m - 1)$ point additions or subtractions. Each point doubling requires four atomic blocks, and each point addition or subtraction requires seven atomic blocks. Each block involves two **`mul`** and four **`add`** or **`sub`** instructions, which translates into 288 clock cycles per block (see Table 2). Each point operation also requires one **`jump`** and one **`stop`**, adding 14 more clock cycles. Thus, the number of clock cycles over all point operations is $[(288 \times 4 + 14) \times (n - 1) + (288 \times 7 + 14) \times (m - 1)]$.

We must also perform two point conversions. Converting affine P_A to randomized Chudnovsky P_C requires two atomic blocks, which translates into an additional 288×2 clock cycles. Converting Jacobian Q_J to affine Q_A involves one **`inv`**, four **`mul`**, and one **`sub`** instruction (to negate y_Q). This adds 4,463 more clock cycles. By including two **`jump`** and two **`stop`** instructions (two programs must be executed), we have a total of $(1{,}166n + 2{,}030m + 1{,}871)$ clock cycles.

We also want to account for the clock cycles required to program the processor. First, we must write 138 instructions (including six **`stop`** instructions) into processor memory, which takes 138×2 clock cycles (Table 2). As for data, we must write six coordinates and read five coordinates, which takes 11×16 clock cycles (Table 2). During execution, we also must write $[(n - 1) + (m - 1) + 2]$ **`jump`** instructions. Each **`jump`** also involves a seven-cycle synchronization. This translates into a penalty of $(2 + 7)$ clock cycles per jump. Hence, the overall programming overhead is $(9n + 9m + 452)$ clock cycles. This overhead is far less than the number of clock cycles needed to execute all processor instructions. Hence, $\Omega_{p256}(n, m, 2) = 1{,}175n + 2{,}039m + 2{,}323$.

```
INPUT: Point $P \neq P_\infty$, NIST prime $p$, curve order $\#E_p$, scalar $k$ with no leading zeros,
       security level $\Gamma_p$
OUTPUT: Point $Q = kP$
  1. IF $\Gamma_p > 1$ THEN generate proper random integer $z_p \in [1, p - 1]$ //Randomize P
  2. IF $\Gamma_p < 3$ THEN:
     2.1. Let $\hat{k} = \#E_p - k$
     2.2. Let $n_1 = |k_B|$, $n_2 = |k_N|$, $n_3 = |\hat{k}_B|$, $n_4 = |\hat{k}_N|$
     2.3. Let $m_1 = |k_B^*|$, $m_2 = |k_N^*|$, $m_3 = |\hat{k}_B^*|$, $m_4 = |\hat{k}_N^*|$
     2.4. FOR $l$ = 1, 2, 3, 4: compute metrics $\Omega_p(n_l, m_l\ \Gamma_p)$ and find the minimum $\Omega_p^{\min}$
     2.5. IF $\Omega_p^{\min} = \Omega_p(n_1, m_1, \Gamma_p)$ THEN let $neg$ = 0 and $s = k_B$  //Select binary k
     2.6. IF $\Omega_p^{\min} = \Omega_p(n_2, m_2, \Gamma_p)$ THEN let $neg$ = 0 and $s = k_N$  //Select NAF k
     2.7. IF $\Omega_p^{\min} = \Omega_p(n_3, m_3, \Gamma_p)$ THEN let $neg$ = 1 and $s = \hat{k}_B$  //Select binary $\hat{k}$, negate Q
     2.8. IF $\Omega_p^{\min} = \Omega_p(n_4, m_4, \Gamma_p)$ THEN let $neg$ = 1 and $s = \hat{k}_N$  //Select NAF $\hat{k}$, negate Q
  3. IF $\Gamma_p = 3$ THEN:
     3.1. Generate proper random integer $i \in [1, k - 1]$ and let $j = k - i$ //Randomize k
     3.2. Let $\hat{i} = \#E_p - \max(i, j)$ and $\hat{j} = \min(i, j)$ //Randomize $\hat{k}$
     3.3. Let $n_1 = |i_B| + |j_B| - 1$, $n_2 = |i_B| + |j_N| - 1$, $n_3 = |i_N| + |j_B| - 1$, $n_4 = |i_N| + |j_N| - 1$
     3.4. Let $n_5 = |\hat{i}_B| + |\hat{j}_B| - 1$, $n_6 = |\hat{i}_B| + |\hat{j}_N| - 1$, $n_7 = |\hat{i}_N| + |\hat{j}_B| - 1$, $n_8 = |\hat{i}_N| + |\hat{j}_N| - 1$
     3.5. Let $m_1 = |i_B^*| + |j_B^*|$, $m_2 = |i_B^*| + |j_N^*|$, $m_3 = |i_N^*| + |j_B^*|$, $m_4 = |i_N^*| + |j_N^*|$
     3.6. Let $m_5 = |\hat{i}_B^*| + |\hat{j}_B^*|$, $m_6 = |\hat{i}_B^*| + |\hat{j}_N^*|$, $m_7 = |\hat{i}_N^*| + |\hat{j}_B^*|$, $m_8 = |\hat{i}_N^*| + |\hat{j}_N^*|$
     3.7. FOR $l$ = 1, 2, 3, 4, 5, 6, 7, 8: compute metrics $\Omega_p(n_l, m_l, \Gamma_p)$ and find the minimum $\Omega_p^{\min}$
     3.8. IF $\Omega_p^{\min} = \Omega_p(n_1, m_1, \Gamma_p)$ THEN let $neg$ = 0, $s = i_B$, $r = j_B$  //Select binary i and
                                                                      binary j
     3.9.  IF $\Omega_p^{\min} = \Omega_p(n_2, m_2, \Gamma_p)$ THEN let $neg$ = 0, $s = i_B$, $r = j_N$ //Select binary i and NAF j
     3.10. IF $\Omega_p^{\min} = \Omega_p(n_3, m_3, \Gamma_p)$ THEN let $neg$ = 0, $s = i_N$, $r = j_B$ //Select NAF i and binary j
     3.11. IF $\Omega_p^{\min} = \Omega_p(n_4, m_4, \Gamma_p)$ THEN let $neg$ = 0, $s = i_N$, $r = j_N$ //Select NAF i and NAF j
     3.12. IF $\Omega_p^{\min} = \Omega_p(n_5, m_5, \Gamma_p)$ THEN let $neg$ = = 1, $s = \hat{i}_B$, $r = \hat{j}_B$  //Select binary $\hat{i}$ and
                                                                        binary $\hat{j}$, negate Q
     3.13. IF $\Omega_p^{\min} = \Omega_p(n_6, m_6, \Gamma_p)$ THEN let $neg$ = = 1, $s = \hat{i}_B$, $r = \hat{j}_N$  //Select binary $\hat{i}$ and
                                                                        NAF $\hat{j}$, negate Q
     3.14. IF $\Omega_p^{\min} = \Omega_p(n_7, m_7, \Gamma_p)$ THEN let $neg$ = 1, $s = \hat{i}_N$, r = $\hat{j}_B$   //Select NAF $\hat{i}$ and
                                                                        binary $\hat{j}$, negate Q
     3.15. IF $\Omega_p^{\min} = \Omega_p(n_8, m_8, \Gamma_p)$ THEN let $neg$ = 1, s = $\hat{i}_N$, $r = \hat{j}_N$ //Select NAF $\hat{i}$ and
                                                                        NAF $\hat{j}$, negate Q
  4. Execute procedure shown in Figure 7
  5. RETURN Q
```

Figure 6. Performance optimization algorithm for given security level Γ_p.

instructions **jump** and **stop**. These instructions' worst-case contribution doesn't exceed 1.4%, which is negligible. Table 9 shows delay due to processor programming. This overhead includes writing instructions and data to and reading data from processor memory, as well as supervisor processor synchronization. The programming penalty is also negligible: less than 1.0%.

Assuming a 60-MHz clock, we can calculate the scalar multiplication delay in milliseconds (60 MHz is the maximum frequency of our ECC processor prototyped on a Xilinx Virtex-4 FPGA). The slowest

Table 5. Processor programming using Library$_p$[1 : 15].

Security level	No. of programs	Library mapping
$\Gamma_p = 0$	5	Program[1 : 5] ← Library$_p$[1 : 3] ∪ Library$_p$[14 : 15]
$\Gamma_p = 1$	5	Program[1 : 5] ← Library$_p$[4 : 6] ∪ Library$_p$[14 : 15]
$\Gamma_p = 2$	6	Program[1 : 6] ← Library$_p$[7 : 10] ∪ Library$_p$[14 : 15]
$\Gamma_p = 3$	8	Program[1 : 8] ← Library$_p$[7 : 14]

```
INPUT: Point $P \neq P_\infty$, NIST prime $p$, scalar $k$ represented by $\{s, r\}$, flag $neg$, security level $\Gamma_p$
OUTPUT: Point $Q = kP$
  1. Assert signal supervisor
  2. Program the processor according to Table 5 and Figure 5d
  3. IF $\Gamma_p$ = 0 THEN:
     3.1. Write coordinates $(x_P, y_P)$ to Memory[$T_P$] and $(x_P, y_P, 1)$ to Memory[$T_Q$] //Initialize $P_A$ and $Q_J$
     3.2. FOR $l = |s| - 2, \ldots, 1, 0$:
          3.2.1. Schedule(1) //$Q_J = 2Q_J$
          3.2.2. IF $s_l = 1$ THEN Schedule(2) ELSE IF $s_l = -1$ THEN Schedule(3) //$Q_J = \pm P_A + Q_J$
     3.3. IF $neg = 0$ THEN Schedule(4) ELSE Schedule(5) //$Q_J \rightarrow Q_A$ or $Q_J \rightarrow -Q_A$
  4. IF $\Gamma_p$ = 1 THEN:
     4.1. Write coordinates $(x_P, y_P)$ to Memory[$T_P$] and $(x_P, y_P, 1)$ to Memory[$T_Q$] //Initialize $P_A$ and $Q_J$
     4.2. FOR $l = |s| - 2, \ldots, 1, 0$:
          4.2.1. Schedule(1) //Atomic $Q_J = 2Q_J$
          4.2.2. IF $s_l = 1$ THEN Schedule(2) ELSE IF $s_l = -1$ THEN Schedule(3) //Atomic $Q_J =$
                                                                                   $\pm P_A + Q_J$
     4.3. IF $neg = 0$ THEN Schedule(4) ELSE Schedule(5) //$Q_J \rightarrow Q_A$ or $Q_J \rightarrow -Q_A$
  5. IF $\Gamma_p$ = 2 THEN:
     5.1.  Write coordinates $(x_P, y_P, z_P)$ to Memory[$T_P$] //Initialize $P_A$ with $z$-coordinate
     5.2.  Schedule(1) //Atomic $P_A \rightarrow P_C$
     5.3.  Read coordinates $(x_P, y_P, z_P)$ from Memory[$T_P$] //Result $P_C$ without $u_P$ and $v_P$
     5.4.  Write coordinates $(x_P, y_P, z_P)$ to Memory[$T_Q$] //Initialize $Q_J$
     5.5.  FOR $l = |s| - 2, \ldots, 1, 0$:
           5.5.1. Schedule(2) //Atomic $Q_J = 2Q_J$
           5.5.2. IF $s_l = 1$ THEN Schedule(3) ELSE IF $s_l = -1$ THEN Schedule(4) //Atomic $Q_J =$
                                                                                   $\pm P_C + Q_J$
     5.6.  IF $neg = 0$ THEN Schedule(5) ELSE Schedule(6) //$Q_J \rightarrow Q_A$ or $Q_J \rightarrow -Q_A$
  6. IF $\Gamma_p$ = 3 THEN:
     6.1.  Write coordinates $(x_P, y_P, z_P)$ to Memory[$T_P$] //Initialize $P_A$ with $z$-coordinate
     6.2.  Schedule(1) //Atomic $P_A \rightarrow P_C$
     6.3.  Read coordinates $(x_P, y_P, z_P)$ from Memory[$T_P$] //Result $P_C$ without $u_P$ and $v_P$
     6.4.  Write coordinates $(x_P, y_P, z_P)$ to Memory[$T_Q$] //Initialize $Q_J$
     6.5.  FOR $l = |s| - 2, \ldots, 1, 0$:
           6.5.1. Schedule(2) //Atomic $Q_J = 2Q_J$
           6.5.2. IF $s_l = 1$ THEN Schedule(3) ELSE IF $s_l = -1$ THEN Schedule(4) //Atomic $Q_J =$
                                                                                   $\pm P_C + Q_J$
     6.6.  IF $neg = 0$ THEN Schedule(5) ELSE Schedule(6) //Atomic $Q_J \rightarrow R_C$ or $Q_J \rightarrow -R_C$
     6.7.  Write coordinates $(x_P, y_P, z_P)$ to Memory[$T_Q$] //Reinitialize $Q_J$
     6.8.  FOR $l = |r| - 2, \ldots, 1, 0$:
           6.8.1. Schedule(2) //Atomic $Q_J = 2Q_J$
           6.8.2. IF $r_l = 1$ THEN Schedule(3) ELSE IF $r_l = -1$ THEN Schedule(4) //Atomic $Q_J =$
                                                                                   $\pm P_C + Q_J$
     6.9.  Schedule(7) //Atomic $Q_J = R_C + Q_J$
     6.10. Schedule(8) //$Q_J \rightarrow Q_A$
  7. Read coordinates $(x_Q, y_Q)$ from Memory[$T_Q$] //Result $Q_A$
  8. Deassert signal supervisor
  9. RETURN $Q$
```

Figure 7. Processor-programming algorithm.

and most secure scalar multiplication takes 112.7 ms, whereas the fastest and least-secure scalar multiplication takes 4.8 ms. The latter 5-ms delay is competitive with the performance of other ECC systems reported in the literature.

Our performance figures don't include execution delays of the supervisor itself running the algorithms shown in Figures 6 and 7. Our future work will study the supervisor's contribution to the overall system latency. Another challenge is related to performing point operations using atomic blocks. Atomic blocks for point doubling are indistinguishable from atomic blocks for point addition or subtraction. Once the supervisor calls the scheduling subroutine (Table 6), these atomic blocks execute in a single program. The delay between two consecutive atomic blocks in the same program is not the same as the delay between

Table 6. Scheduling subroutine *Schedule*(*l*).

Step	Action
1	Write instruction `jump`(T_l) to Memory[0]
2	Deassert signal *supervisor*
3	Assert signal *globalreset*
4	Deassert signal *globalreset* and assert signal *globalstart*
5	Wait for signal *busy* to be asserted
6	Deassert signal *globalstart*
7	Wait for signal *busy* to be deasserted
8	Assert signal *supervisor*

Table 7. Processor performance metric Ω_p (in clock cycles) for different security levels and NIST primes.

Security	NIST prime				
level Γ_p	*p*192	*p*224	*p*256	*p*384	*p*521
0	286,401	347,311	411,933	1,194,957	2,736,756
1	349,950	426,646	508,526	1,384,096	3,053,764
2	388,231	473,293	564,115	1,536,605	3,391,889
3	772,201	941,761	1,122,841	3,064,811	6,759,717

Table 8. Overhead of instructions `jump` and `stop` (in clock cycles) for different security levels and NIST primes.

Security	NIST prime				
level Γ_p	*p*192	*p*224	*p*256	*p*384	*p*521
0	4,018	4,690	5,362	8,050	10,934
1	4,018	4,690	5,362	8,050	10,934
2	4,032	4,704	5,376	8,064	10,948
3	8,036	9,380	10,724	16,100	21,840

Table 9. Processor programming overhead (in clock cycles) for different security levels and NIST primes.

Security	NIST prime				
level Γ_p	*p*192	*p*224	*p*256	*p*384	*p*521
0	2,813	3,259	3,705	5,489	7,413
1	2,881	3,327	3,773	5,557	7,481
2	3,000	3,454	3,908	5,724	7,688
3	5,710	6,602	7,494	11,062	14,892

the current program's last atomic block and the next program's first atomic block. The latter is greater than the former and is determined by the delay of the next call for the scheduling subroutine. This difference could identify the type of point operation, thus leaking the private scalar value. Our future work will address this problem. ■

Acknowledgments

We thank the Canadian Microelectronics Corp., Canadian Foundation for Innovation, British Columbia Knowledge Development Fund, and Xilinx University Program for their support.

References

1. D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, Springer, 2004.
2. *Federal Information Processing (FIPS) 186-2, Digital Signature Standard (DSS)*, National Inst. of Standards and Technology, 2000.
3. S. Ors et al., "Hardware Implementation of an Elliptic Curve Processor over *GF*(*p*)," *Proc. 14th IEEE Int'l Conf. Application-Specific Systems, Architecture and Processors* (ASAP 03), IEEE CS Press, 2003, pp. 433-443.
4. W. Shuhua and Z. Yuefei, "A Timing-and-Area Tradeoff *GF*(*p*) Elliptic Curve Processor Architecture for FPGA," *Proc. Int'l Conf. Communications, Circuits and Systems*, IEEE Press, 2005, pp. 1308-1312.
5. A. Satoh and K. Takano, "A Scalable Dual-Field Elliptic Curve Cryptographic Processor," *IEEE Trans. Computers*, vol. 52, no. 4, Apr. 2003, pp. 449-460.
6. G. Orlando and C. Paar, "A Scalable *GF*(*p*) Elliptic Curve Processor Architecture for Programmable Hardware," *Proc. 3rd Int'l Workshop Cryptographic Hardware and Embedded Systems* (CHES 01), LNCS 2162, Springer, 2001, pp. 348-363.
7. S. Xu and L. Batina, "Efficient Implementation of Elliptic Curve Cryptosystems on an ARM7 with Hardware Accelerator," *Proc. 4th Int'l Information Security Conf.* (ISC 01), LNCS 2200, Springer, 2001, pp. 266-279.
8. J. Wolkerstorfer, "Dual-Field Arithmetic Unit for *GF*(*p*) and *GF*(2^m)," *Proc. 4th Int'l Workshop Cryptographic Hardware and Embedded Systems* (CHES 02), LNCS 2523, Springer, 2002, pp. 500-514.
9. A. Daly et al., "An FPGA Implementation of a *GF*(*p*) ALU for Encryption Processors," *Microprocessors and Microsystems*, vol. 28, no. 5-6, 2004, pp. 253-260.
10. K. Sakiyama et al., "Reconfigurable Modular Arithmetic Logic Unit for High-Performance Public-Key Cryptosystems," *Proc. Int'l Workshop Applied Reconfigurable Computing* (ARC 06), LNCS 3985, Springer, 2006, pp. 347-357.
11. I. Blake, G. Seroussi, and N. Smart (Eds.) (2005). *Advances in Elliptic Curve Cryptography*, Cambridge Univ. Press, 2005.
12. K. Ananyi and D. Rakhmatov, "Design of a Reconfigurable Processor for NIST Prime Field ECC," *Proc. 14th Ann. IEEE Symp. Field-Programmable Custom Computing Machines* (FCCM 06), IEEE CS Press, 2006, pp. 333-334.

Hamad Alrimeih is pursuing a PhD in the Electrical and Computer Engineering Department of the University of Victoria, British Columbia, Canada. His research interests include adaptive architectures for elliptic-curve cryptography. Alrimeih has a BS in computer engineering from King Saud University, Saudi Arabia, an MS in computer engineering from Essex University, UK, and an MS in electrical engineering from Edinburgh University, UK.

Daler Rakhmatov is an assistant professor in the Electrical and Computer Engineering Department of the University of Victoria, British Columbia, Canada. His research interests include energy-efficient computing and dynamically reconfigurable systems. Rakhmatov has a BS in electrical engineering from the Rochester Institute of Technology, and an MS and a PhD in electrical engineering from the University of Arizona, Tucson. He is a member of the IEEE.

■ Direct questions and comments about this article to Daler Rakhmatov, Dept. of Electrical and Computer Engineering, University of Victoria, PO Box 3055, STN CSC Victoria, British Columbia, V8W 3P6 Canada; daler@ece.uvic.ca.

For further information about this or any other computing topic, please visit our Digital Library at http://www.computer.org/csdl.