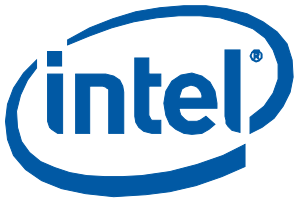


White Paper

Krzysztof Jankowski
Pierre Laurent
Aidan O'Mahony
Firmware Engineers at
Intel Corporation

Intel Polynomial Multiplication Instruction and its Usage for Elliptic Curve Cryptography

April, 2012

326779-002

Executive Summary

Elliptic Curve Cryptography (ECC) is an algorithm for public-key cryptography based on elliptic curves over finite fields and is an alternative to commonly-used methods, such as RSA, DSA and Diffie-Hellman. This paper details an experimental study that uses new Intel instructions to increase the performance of ECC by up to 600 percent.

Using the PCLMULQDQ instruction, the performance of ECC can be boosted significantly on a range of IA processor cores, making it an attractive option over other public key algorithms.

Note: Intel does not have a position with respect to the relative strength/weakness of the cryptographic algorithms mentioned in this paper and does not promote or discourage the use of any specific cryptographic algorithm type.

Contents

PCLMULQDQ Instruction Definition

The PCLMULQDQ instruction was added to the 2010 Intel® Core™ processor family based on the 32 nm Intel® microarchitecture, codenamed Westmere. The instruction performs carry-less multiplication of two 64-bit operands, which can be used for CRC computation and block cipher encryption (that is, GCM), among many other algorithms. As demonstrated in the following, accelerating carry-less multiplication can play a significant role in achieving high-speed secure computing and communication.

Elliptic Curve Cryptography

In 1985, Elliptic Curve Cryptography (ECC) was proposed independently by cryptographers Victor Miller (IBM) and Neal Koblitz (University of Washington). The security attributes of ECC are based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP). Like the prime factorization problem, which RSA is based upon, ECDLP is another "hard" problem that is deceptively simple to state:

> "Given two points, P and Q, on an elliptic curve, and an additive group operation on the group of points on the curve, find the integer n, if it exists, such that P = nQ."

Elliptic curves combine number theory and algebraic geometry. These curves can be defined over any field of numbers (that is, real, integer, complex); although they are commonly used over finite fields for applications in cryptography. A (simplified) elliptic curve consists of the set of real numbers (x,y) that satisfies the equation:

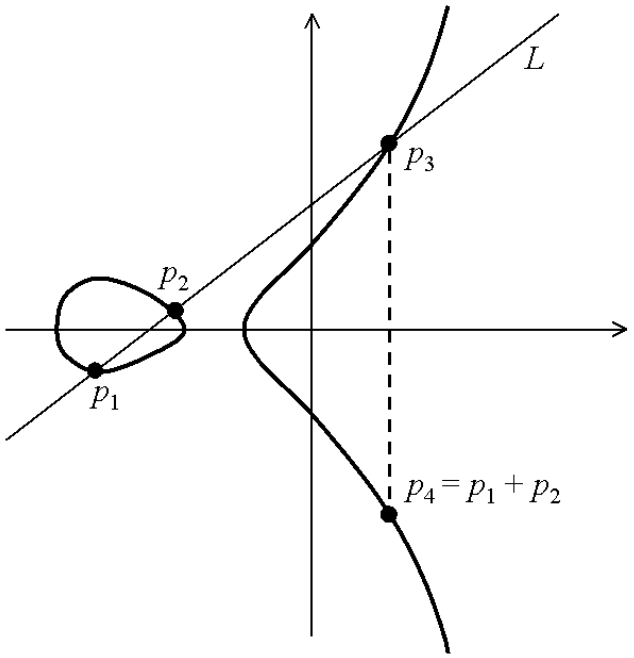
$$y^2 = x^3 + ax + b$$

The set of all the solutions to the equation forms the elliptic curve. Changing a and b will impact the shape of the curve and small variations in these parameters can result in major changes in the set of (x,y) solutions.

ECC Point Multiplication

Elliptic curve point multiplication is the operation of successively adding a point along an elliptic curve to itself repeatedly. Further defining point addition, it is taking the line through two points along a curve E and computing its intersection with the curve in another point P3, followed by negating of the y coordinate to give P4 (see the example in the following figure).

Figure 1. Elliptic Curve Point Addition



There are a variety of formulas to compute a point addition. One such example (taken from [4]) is:

$$x4 = \frac{(y2 - y1)^2}{(x2 - x1)^2} - x1 - x2$$

$$y4 = \frac{(2 * x1 + x2) * (y2 - y1)}{(x2 - x1)} - \frac{(y2 - y1)^3}{(x2 - x1)^3} - y1$$

where, P1 = (x1,y1) and P2 = (x2,y2).

Polynomial Multiplication

The schoolboy multiplication of two polynomials $A = \sum_{i<\deg(A)} a_i x^i$ and $B = \sum_{j<\deg(B)} a_j x^j$ is $R = A.B = \sum_{i<\deg(A)} \sum_{j<\deg(B)} (a_i \otimes b_j) x^{(i+j)}$ and the PCLMULQDQ instruction implements this double loop in a few cycles for polynomials up to degree 64. The multiplication of larger polynomials requires additional steps that can be implemented efficiently with Karatsuba multiplication techniques as described in [5]. For example, the multiplication of polynomials of degree 192 can be implemented by interleaving the steps of the following formula, where the degree(64) multiplier is used six times:

$$
\begin{aligned}
R &= A.B \\
&= (a_0 b_0)(x^0 \oplus x^{2n}) \\
&\oplus (a_1 b_1)(x^{2n} \oplus x^{3n}) \\
&\oplus (a_2 b_2)(x^n \oplus x^{2n} \oplus x^{3n} \oplus x^{4n}) \\
&\oplus (a_2 \oplus a_0)(b_2 \oplus b_0)(x^n \oplus x^{2n}) \\
&\oplus (a_2 \oplus a_1)(b_2 \oplus b_1)(x^n \oplus x^{3n}) \\
&\oplus (a_2 \oplus a_1 \oplus a_0)(b_2 \oplus b_1 \oplus b_0) x^n
\end{aligned}
$$

Squaring a degree 192 polynomial is based on a linear squaring property:

$$R = A.A = a_0^{\,2} x^0 \oplus a_1^{\,2} x^{2n} \oplus a_2^{\,2} x^{4n}$$

Field multiplications require an additional reduction step. Aside from the two standard textbook methods of modular multiplication (Montgomery and Barrett), special forms of the field polynomial lend themselves to very efficient, direct reduction via shift-and-xor algorithms. The equivalent can be implemented more efficiently when using the special form of the field polynomial selected for ECC standard curves, based on identities, such as the following one suitable for any trinomial:

$$
\begin{aligned}
T &= R \bmod (x^n + x^m + 1) \\
&= r_{0..n-1} \oplus (r_{n..\deg(R)} \otimes (x^m + 1))
\end{aligned}
$$

Proposed Modifications

To evaluate the PCLMULQDQ instruction, engineers from Intel investigated its applicability to the implementation of ECC-based public key algorithms with cryptographic strength comparable to the 2048-bit RSA/DH/DSA family. IETF RFC4492 "ECC Cipher Suites for TLS" suggests that 233-bit curves, specifically 'sect233k1' and 'sect233r1', satisfy the requirement.

Since the implementation of ECC algorithms based on those curves already existed in the OpenSSL v1.0.0a toolkit ('nistk233' and 'nistb233' respectively), the engineers from Intel decided to modify its source code, for experimental purposes and benchmark the PCLMULQDQ instruction performance using this modified framework. The OpenSSL implementation of ECC's primitives (GF2 point multiplication, modular multiplication, modular reduction and so on) is generic, in that it is curve type independent. The engineers from Intel extended it by adding dedicated code paths for 233-bit curves. The main OpenSSL code dealing with GF2 operations is located inside the `crypto/bn/gf2m.c` source file. Initially, a few functions, listed below (with code snippets in the Appendix) were modified:

- int GF2m_mod_arr(…)
  - Added XMM register-based Barrett's modular reduction
- int GF2m_mod_mul_arr(…)
  - Added PCLMULQDQ-based squaring
  - Added PCLMULQDQ-based Karatsuba multiplication
  - Added XMM register-based Barrett's modular reduction
- int GF2m_mod_sqr_arr(…)
  - Added PCLMULQDQ-based squaring
  - Added XMM register-based Barrett's modular reduction
- int rshift1(…) from *crypto/bn/shift.c*
  - Added XMM register-based right shift by 1-bit

These changes provided significant ECC DH and ECC DSA speed improvements, but the ECC DSA Sign operation only improved by a small amount.

The GF2 point multiplication operation, which forms a basis for ECC GF2 algorithms, can be implemented in the affine (Euclidean) or projective (Jacobian) coordinate systems. The multiplication algorithm for affine coordinates requires modular inversion to be performed after every multiplication step. However, for projective coordinates, more modular multiplications are required, but the modular inversion is required only at the end of the entire point multiplication algorithm. OpenSSL v1.0.0a uses affine point coordinates for ECC GF2 curves and projective point coordinates (by

default, affine is available too) for ECC GFp curves. Assuming that the standard OpenSSL's implementation of modular inversion represents a significant portion of the CPU computing task, then ECC DSA Sign operation involving a large number of point multiplication iterations will be slow by design (slower than GFp). To improve GF2 ECC DSA Sign implementation, three options were considered:

1. Speed-up modular reduction by taking advantage of PCLMULQDQ in the inversion phase.
2. Modify OpenSSL to use the projective coordinates system for GF2 curves.
3. Implement both 1 and 2 at the same time.

Although options 2 and 3 offered the biggest potential DSA Sign speed improvements, due to time constraints, the engineers at Intel pursued option 1, leaving 2 and 3 for future development. The modular reduction algorithm for GF2 based curves, as implemented in OpenSSL v1.0.0a, is based on MAIA from [1] and it extensively uses right-shift-by-1bit operations; unfortunately, there is no straightforward way to speed up this code using the PCLMULQDQ instruction. Overcoming this limitation, the engineers at Intel implemented a modular inversion algorithm, described by Kobayashi et al. [2] and ported it to the OpenSSL framework by making a straightforward change to the following function:

- int GF2m_mod_inv(…)
  - Added PCLMULQDQ based inversion

Computing 233-bit numbers is fairly convenient in the XMM registers domain, as a single number takes only two 128-bit XMMs (256-bits array). Temporary numbers – such as the result of 233-bit by 233-bit multiplication – require four 128-bit XMMs (512-bits array). This is not an issue in 64-bit execution mode, since the CPU provides enough temporary registers – 16 XMMs in total.

Performance

The following performance results were obtained from running an OpenSSL executable on a 3.33 GHz Intel® processor based on Intel® microarchitecture codenamed "Westmere", with gcc v4.4.0 on 64-bit Linux*:

Domain	OpenSSL v1.0.0a Test	Unit	Vanilla	Patched	P/V
gf2	speed ecdhk233	[op/s]	1374.1	7036.9	5.2
	speed ecdsak233	[sign/s]	1099.8	3036.8	2.8
		[verify/s]	660.3	3256.7	5.0
	speed ecdhb233	[op/s]	1309.4	7407.8	5.7
	speed ecdsab233	[sign/s]	1095.8	3026.4	2.8
		[verify/s]	631.7	3402.4	5.4
gfp	speed ecdhp224	[op/s]	1925.5	---	---
	speed ecdsap224	[sign/s]	7278.8	---	---
		[verify/s]	1599.8	---	---
int	speed dsa2048	[sign/s]	1237.6	---	---
		[verify/s]	1061.4	---	---
	speed rsa2048	[sign/s]	361.5	---	---
		[verify/s]	12461.6	---	---

The next table shows the performance results obtained by running OpenSSL tests on a 1.60 GHz Intel® processor based on Intel® microarchitecture codenamed "Sandy Bridge", with gcc v4.5.1 on 64-bit Linux*.

Domain	OpenSSL v1.0.0a Test	Unit	Vanilla	Patched	P/V
gf2	speed ecdhk233	[op/s]	710.1	3877.8	5.5
	speed ecdsak233	[sign/s]	478.7	1832.7	3.9
		[verify/s]	341.7	1835.7	5.4
	speed ecdhb233	[op/s]	667.1	4116.2	6.2
	speed ecdsab233	[sign/s]	481.7	1820.7	3.8
		[verify/s]	322.2	1909.3	6.0
Gfp	speed ecdhp224	[op/s]	1031.5	---	---
	speed ecdsap224	[sign/s]	3854.3	---	---
		[verify/s]	853.5	---	---

Domain	OpenSSL v1.0.0a Test	Unit	Vanilla	Patched	P/V
Int	speed dsa2048	[sign/s]	887.6	---	---
		[verify/s]	768.3	---	---
	speed rsa2048	[sign/s]	264.6	---	---
		[verify/s]	9080.0	---	---

Results Analysis

The following were observed:

- The PCLMULQDQ instruction executing on a Westmere core increased the speed of GF2 ECC DH and ECC DSA by a factor of 5.3 times, and ECC DSA Sign by 2.8 times.
- The Sandy Bridge core produced even better results when executing the same tests.
- Patched GF2 ECC DH and ECC DSA outperformed the corresponding ECC algorithms in the GFp domain.
- GFp DSA Sign was faster than GF2 DSA Sign, but GF2 based on projective point coordinates could produce superior results.
- Patched GF2 ECC DSA operations outperformed 2048-bit integer DSA.
- 2048-bit RSA Sign was the slowest operation tested (even for generic cases).
- 2048-bit RSA performance is hard to compare to any ECC-based operation.

Conclusion and Future Experiments

We have shown that using the PCLMULQDQ instruction has benefits for the public key operations of a TLS transaction (that is, the handshake), but there are other areas where the instruction can be of benefit as outlined below.

TLS/SSL Optimizations Using PCLMULQDQ

In addition to improvements in handshake operations, the PCLMULQDQ instruction can also be used to speed up the associated secured connection (for example, if the connection is secured with AES-GCM). See [3] for more information on this.

Higher Level Algorithm Improvements

A key learning of this investigation is that more effort is needed than simply putting the PCLMULQDQ instruction into GF2 modular operations; it was necessary to revisit the higher-level algorithms for inversion, due to the availability of a hardware multiplication. Furthermore, there may be some other high-level tasks worth exploring.

Generic Modular Reduction Optimized for Generic Curves

While this investigation focused on a subset of curves (B-233 and K-233), it should be possible to provide a faster generic modular reduction optimized for generic curves.

Better Scheduling of Instructions between Multiplications and Reductions

This study did not explore the possibility of decoupling some of the instructions in modular operations to allow the compiler and/or hardware to better pipeline the instructions. This could provide some improvement if multiplication and reductions can be interleaved.

Choice of Coordinate System

Although OpenSSL implements GF2 point operations based on the affine Weierstrass coordinates system, other coordinate representations (GFP uses modified Jacobian coordinates) may produce a better result. Consequently, others working in this field may find it worthwhile to investigate alternative coordinate systems.

References

[1] "Software Implementation of Elliptic Curve Cryptography over Binary Fields", D. Hankerson, J. L. Hernandez, A. Menezes

[2] "An Algorithm for Inversion in GF(2^m) Suitable for Implementation Using a Polynomial Multiply Instruction on GF(2)", K. Kobayashi, N. Takagi, K. Takagi

[3] "Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions," Krzysztof Jankowski, Pierre Laurent, IEEE Transactions on Computers, pp. 135-138, January, 2011

[4] "Explicit-formulas database (2007)", Daniel J. Bernstein, Tanja Lange, URL: http://hyperelliptic.org/EFD

[5] "Five, Six, and Seven-Term Karatsuba-Like Formulae", P. L. Montgomery, IEEE Transactions on Computers, vol. 54, no. 3, pp. 362-369, 2005

[6] "Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C"; Order Number: 325462-041US, December 2011
URL: http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-software-developer-vol-1-2a-2b-3a-3b-manual.pdf

[7] "A Technique for Accelerating Characteristic 2 Elliptic Curve Cryptography", Shay Gueron, Michael E. Kounavis; Fifth "International Conference on Information Technology: New Generations", IEEE Computer Society, (ITNG 2008), p. 265-272 (2008)

[8] "Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode", Shay Gueron, Michael E. Kounavis; January 2010
URL: http://software.intel.com/en-us/articles/intel-carry-less-multiplication-instruction-and-its-usage-for-computing-the-gcm-mode/

Appendix

Karatsuba Multiplication 128-bit x 128-bit = 256-bit

```
// c1c0 = a * b
void GF2m_mul_2x2_xmm(__m128i *c1, __m128i *c0, __m128i b,
__m128i a)
{
     __m128i t1, t2;
     *c0 = _mm_clmulepi64_si128(a, b, 0x00);
     *c1 = _mm_clmulepi64_si128(a, b, 0x11);
     t1  = _mm_shuffle_epi32(a, 0xEE);
     t1  = _mm_xor_si128(a, t1);
     t2  = _mm_shuffle_epi32(b, 0xEE);
     t2  = _mm_xor_si128(b, t2);
     t1  = _mm_clmulepi64_si128(t1, t2, 0x00);
     t1  = _mm_xor_si128(*c0, t1);
     t1  = _mm_xor_si128(*c1, t1);
     t2  = t1;
     t1  = _mm_slli_si128(t1, 8);
     t2  = _mm_srli_si128(t2, 8);
     *c0 = _mm_xor_si128(*c0, t1);
     *c1 = _mm_xor_si128(*c1, t2);
}
```

Karatsuba Multiplication 256-bit x 256-bit = 512-bit

```
// c3c2c1c0 = a1a0 * b1b0
{
     a1 = _mm_set_epi64x(a->d[3], a->d[2]);
     a0 = _mm_set_epi64x(a->d[1], a->d[0]);
     b1 = _mm_set_epi64x(b->d[3], b->d[2]);
     b0 = _mm_set_epi64x(b->d[1], b->d[0]);
     GF2m_mul_2x2_xmm(&c1, &c0, a0, b0);
     GF2m_mul_2x2_xmm(&c3, &c2, a1, b1);
     a0 = _mm_xor_si128(a0, a1);
     b0 = _mm_xor_si128(b0, b1);
     GF2m_mul_2x2_xmm(&c5, &c4, a0, b0);
     c4 = _mm_xor_si128(c4, c0);
     c4 = _mm_xor_si128(c4, c2);
     c5 = _mm_xor_si128(c5, c1);
     c5 = _mm_xor_si128(c5, c3);
     c1 = _mm_xor_si128(c1, c4);
     c2 = _mm_xor_si128(c2, c5);
}
```

Quick Squaring $(256\text{-bit})^2 = 512\text{-bit}$

```
// c3c2c1c0 = a1a0 * a1a0
{
    c0  = _mm_clmulepi64_si128(a0, a0, 0x00);
    c1  = _mm_clmulepi64_si128(a0, a0, 0x11);
    c2  = _mm_clmulepi64_si128(a1, a1, 0x00);
    c3  = _mm_clmulepi64_si128(a1, a1, 0x11);
}
```

Barrett's Modular Reduction with $p(x) = x^{233} + x^{74} + 1$

```
// x = (x << n), z = 0
#define XMM_SHL_N(x, n, z)          \
{                                   \
    __m128i u, v;                   \
    u = x;                          \
    x = _mm_slli_epi64(x, n);       \
    u = _mm_srli_epi64(u, (64-n));  \
    v = _mm_unpacklo_epi64(z, u);   \
    x = _mm_or_si128(x, v);         \
} while(0)

// c1c0 = c3c2c1c0 MOD p
#define GF2m_barrett_xmm(c3, c2, c1, c0)                \
{                                                       \
        ULONG m[2] = {0xffffffffffffffff,               \
                      0x000001ffffffffff};              \
    __m128i b3, b2, b1, b0, a1, a0, m0, z0;             \
    m0 = _mm_set_epi64x(m[1], m[0]);                    \
    z0 = _mm_xor_si128(z0, z0);                         \
    b1 = c1;                                            \
    a1 = c1;                                            \
    a0 = _mm_move_epi64(c1);                            \
    a1 = _mm_slli_epi64(a1, 23);                        \
    a1 = _mm_srli_epi64(a1, 23);                        \
    c1 = _mm_or_si128(a1, a0);                          \
    b2 = _mm_srli_epi64(c2, (64-23));                   \
    XMM_SHL_N(c3, 23, z0);                              \
    a0 = _mm_unpackhi_epi64(b2, z0);                    \
    c3 = _mm_or_si128(c3, a0);                          \
    b1 = _mm_srli_epi64(b1, (64-23));                   \
    XMM_SHL_N(c2, 23, z0);                              \
    a0 = _mm_unpackhi_epi64(b1, z0);                    \
    c2 = _mm_or_si128(c2, a0);                          \
    b3 = c3;                                            \
    b2 = _mm_srli_epi64(c2, (64-10));                   \
    XMM_SHL_N(b3, 10, z0);                              \
    a0 = _mm_unpackhi_epi64(b2, z0);                    \
    b3 = _mm_or_si128(b3, a0);                          \
    a0 = _mm_unpackhi_epi64(c3, z0);                    \
    b3 = _mm_xor_si128(b3, a0);                         \
```

```
    b1 = _mm_srli_epi64(b3, (64-23));                \
    XMM_SHL_N(b3, 23, z0);                           \
    b3 = _mm_unpackhi_epi64(b3, z0);                 \
    b3 = _mm_or_si128(b3, b1);                       \
    c2 = _mm_xor_si128(c2, b3);                      \
    b3 = c3;                                         \
    b2 = _mm_srli_epi64(c2, (64-10));                \
    XMM_SHL_N(b3, 10, z0);                           \
    b2 = _mm_unpackhi_epi64(b2, z0);                 \
    b3 = _mm_or_si128(b3, b2);                       \
    b2 = c2;                                         \
    XMM_SHL_N(b2, 10, z0);                           \
    a0 = _mm_unpacklo_epi64(z0, b2);                 \
    c2 = _mm_xor_si128(c2, a0);                      \
    a0 = _mm_unpacklo_epi64(z0, b3);                 \
    a1 = _mm_unpackhi_epi64(b2, z0);                 \
    a0 = _mm_or_si128(a0, a1);                       \
    c3 = _mm_xor_si128(c3, a0);                      \
    c0 = _mm_xor_si128(c0, c2);                      \
    c1 = _mm_xor_si128(c1, c3);                      \
    c1 = _mm_and_si128(c1, m0);                      \
} while(0)
```

Authors

Krzysztof Jankowski is a firmware engineer at Intel Corporation.

Pierre Laurent is a firmware engineer at Intel Corporation.

Aidan O'Mahony was a firmware engineer at Intel Corporation during 2011.

Acronyms

AES	Advanced Encryption Standard
CRC	Cyclic Redundancy Check
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
DH	Diffie-Hellman
DSA	Digital Signature Algorithm
GCM	Galois/Counter Mode
RSA	Rivest-Shamir-Adleman (algorithm)
SSL	Secure Sockets Layer
TLS	Transport Layer Security

§