White Paper

Carry-Less Multiplication and Its Usage for Computing the GCM Mode

Mobility Group, Israel Development Center

Shay Gueron

Corporate Technology Group, Communications Technology Lab

Michael E. Kounavis

Intel's PCLMULQDQ is a new Single Instruction Multiple Data (SIMD) instruction that will be introduced in the next generation of Intel® processor, as of 2009. PCLMULQDQ performs carry-less multiplication of two 64-bit operands. This paper provides information on using this instruction for computing the Galois Counter Mode.

April 2008

Intel Corporation

Contents

Figures

Introduction

This white paper provides details on Intel's carry-less multiplication instruction, PCLMULQDQ, that computes the carry-less multiplication of two 64-bit operands. The paper explains algorithms for using this instruction for computing the Galois Hash, which is the underlying computation of the Galois Counter Mode (GCM).

An important usage model is AES-GCM, where in this case the AES encryption/decryption part could be efficiently implemented by the AES instructions that are also being introduced into the ISA.

Preliminaries

Usage Models of Carry-less Multiplication

Carry-less multiplication is the mathematical operation of computing the (carry-less) product of two operands without the generation or propagation of carry values. It is relatively a time consuming operation when implemented with the current ISA of the IA architecture. For example, software implementation of carry-less multiplication that uses the best-known method (found at the OpenSSL source code distribution, www.openssl.org) computes a 64 by 64 bit carry-less product in about 100 cycles.

Carry-less multiplication is an essential processing component of several cryptographic systems and standards. Hence, accelerating carry-less multiplication can significantly contribute to achieving high speed secure computing and communication. Carry-less multiplication is especially important for implementing the Galois Counter Mode (GCM), which is a recently defined mode of operation of block ciphers [4, 6, 7, 8, 14, 15]. The GCM mode was endorsed by the US government in April 2006 and is used together with AES, which is part of the NSA Suite B. It is also defined in IEEE 802.1ae standard, where it is recommended for forwarding rates higher than 10 Gbps. Other usage models of GCM include IPsec (IPsec RFC 4106), the storage standard P1619 and security protocols over fiber channel (ISO-T11 standard).

GCM requires the carry-less multiplication of 128-bit operands, producing a 256-bit product. This is the first step of computing a 'Galois hash', which is part of the GCM mode. The PCLMUQDQ instruction computes the 128-bit product of two 64-bit operands. It can be used by software as a building block for generating the 256-bit result required for GCM.

The other step in GCM is reduction modulo of a pentanomial $x^{128} + x^7 + x^2 + x + 1$. In this document, we describe a new efficient algorithm for performing this reduction in the Intel SSE domain (using PSRLD, PSLLD PSHUFD instructions). The combination of the PCLMUQDQ instruction, together with this algorithm speeds up the GCM mode.

Carry-less multiplication is also in the core computation in Elliptic Curve Cryptography (ECC) over binary fields [2] and in Cyclic Redundancy Check (CRC) computations. The carry-less multiplication instruction PCLMUQDQ can speedup the computation of CRC with polynomials other than the iSCSI polynomial, for which there is already a dedicated instruction in the ISA (namely, CRC32 that is part of the Intel SSE4 set).

Carry-Less Multiplication - Definition

Carry-less multiplication is the operation of multiplying two operands without generating and propagating carries. It is formally defined as follows. Let the two operands *A*, *B*, be defined by the following n-bit array notation

$$A=[a_{n-1}\ a_{n-2}\ \ldots\ a_0],\ B=[b_{n-1}\ b_{n-2}\ \ldots\ b_0] \quad (1)$$

and let the carry-less multiplication result be the following 2n bit array:

$$C=[c_{2n-1}\ c_{2n-2}\ \ldots\ c_0] \quad (2)$$

The bits of the output *C* are defined as the following logic functions of the bits of the inputs *A* and *B* as follows:

$$c_i=\bigoplus_{j=0}^{i} a_j b_{i-j} \quad (3)$$

for $0 \le i \le n-1$, and

$$c_i=\bigoplus_{j=i-n+1}^{n-1} a_j b_{i-j} \quad (4)$$

for $n \le i \le 2n-1$ (note that $c_{2n-1}=0$).

One can see that the logic functions of Equations (3) and (4) are somehow analogous to integer multiplication in the following sense. In integer multiplication, the first operand is shifted as many times as the positions of bits equal to "1" in the second operand. The integer multiplication is obtained by adding the shifted versions of the first operand with each other. The same procedure is followed in carry-less multiplication, but the ``additions'' do not generate or propagate carry, and are equivalent to the exclusive OR (XOR) logical operation.

Hereafter, carry-less multiplication is denoted by the symbol "•".

Carry-less Multiplication and Galois Field Multiplication

Typically, carry-less multiplication is used as the first step of multiplications in finite fields (aka Galois Fields) of characteristic 2 [10, 11, 12, 13].

A Galois Field is a finite set of elements where the operations addition '+' and multiplication '·' are defined. The set is closed under these operations where they satisfy the following properties: associativity, commutativity, existence of a neutral element (for addition it is called "0" and for multiplication it is called "1"), existence of additive inverse, existence of multiplicative inverse for each element except for zero, and a distributive law for multiplication over addition (i.e., a·(b+c) = a·b+a·c).

The number of elements of a finite field must be a power of some prime *p*, where in such case the field is denoted by GF(p^k). A binary field (field with characteristic 2) is the case where $p=2$, and is denoted by GF (2^k).

In general, the elements in GF(p^k) can be viewed as polynomials of degree *k* where the operations are defined using some irreducible polynomial of degree *k*: addition of two elements is defined as polynomial addition (adding the

corresponding coefficients modulo *p*). Multiplication is defined by polynomial multiplication, which is subsequently reduced modulo the irreducible polynomial that defines the finite field.

Typically, there are multiple different irreducible polynomials of degree *k*. Using them to define the finite field results in isomorphic representations of the field.

For binary fields GF (2^n), one can view the elements as n-bit strings, where each bit represents the corresponding coefficient of the polynomial. In such cases, addition is equivalent to the bitwise XOR of the two strings. Multiplication consists of two steps. The first step is carry-less multiplication of the two operands. The second step is the reduction of this carry-less product modulo the polynomial that defines that field.

For example, consider the field GF(2^4) (i.e., n=4) defined by the reduction polynomial x^4+x+1. Let A = [1110] and B = [1011] be two elements in that field. Then, their product in this field (i.e., their Galois Field multiplication) is [1000]. This result is obtained as follows: computing the carry-less multiplication C = A • B = [01100010], followed by reducing C modulo x^4+x+1. The reduction result is obtained by finding that [01100010] = [0110] • [10011] XOR [1000].

PCLMULQDQ Instruction Definition

PCLMULQDQ instruction performs carry-less multiplication of two 64-bit quadwords which are selected from the first and the second operands according to the immediate byte value.

Instruction format: PCLMULQDQ xmm1, xmm2/m128, imm8
Description: Carry-less multiplication of one quadword (8-byte) of xmm1 by one quadword (8-byte) of xmm2/m128, returning double quadwords (16 bytes). The immediate byte is used for determining which quadwords of xmm1 and xmm2/m128 should be used. **Opcode:** 66 0f 3a 44 The presence of PCLMULQDQ is indicated by the CPUID leaf 1 ECX[1]. Operating systems that support the handling of Intel SSE state will also support applications that use AES extensions and the PCLMULQDQ instruction. This is the same requirement for Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.

The immediate byte values are used as follows.

imm[7:0]	Operation
0x00	xmm2/m128[63:0] • xmm1[63:0]
0x01	xmm2/m128[63:0] • xmm1[127:64]
0x10	xmm2/m128[127:64] • xmm1[63:0]
0x11	xmm2/m128[127:64] • xmm1[127:64]

NOTES:

1. The symbol "•" denotes carry-less multiplication
2. Immediate bits other than 0 and 4 are ignored.

The pseudo code for defining the operation is as follow.

```
IF imm8[0] == 0 THEN
        Temp1 = xmm1[63:0]
ELSE
        Temp1 = xmm1[127:64]
ENDIF
IF imm8[1] == 0 THEN
        Temp2 = xmm2/m128[63:0]
ELSE
        Temp2 = xmm2/m128[127:64]
ENDIF
FOR i = 0 TO  63
        TempB [i] := (Temp1[0] AND Temp2[i]);
        FOR j = 1 TO i, 1
                TempB [i] := TempB [i] XOR (Temp1[j] AND Temp2[i -j])
        NEXT j
        Dest[i] := TempB[i];
NEXT i
FOR i = 64 TO 126, 1
        TempB [i] := (Temp1[ i-63] AND Temp2[63]);
        FOR j = i-62 TO 63, 1
                TempB [i] := TempB [i] XOR (Temp1[j] AND Temp2[i -j])
        NEXT j
        Dest[i] := TempB [i];
NEXT i
Dest[127] := 0;
```

Identifying PCLMULQDQ Support by the Processor

Before an application attempts to use the PCLMULQDQ instruction it should check for its availability, which is indicated if CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1.

Operating systems that support handling Intel® SSE state support also applications that use PCLMULQDQ. This is the same requirement for Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4.

The Galois Counter Mode (GCM)

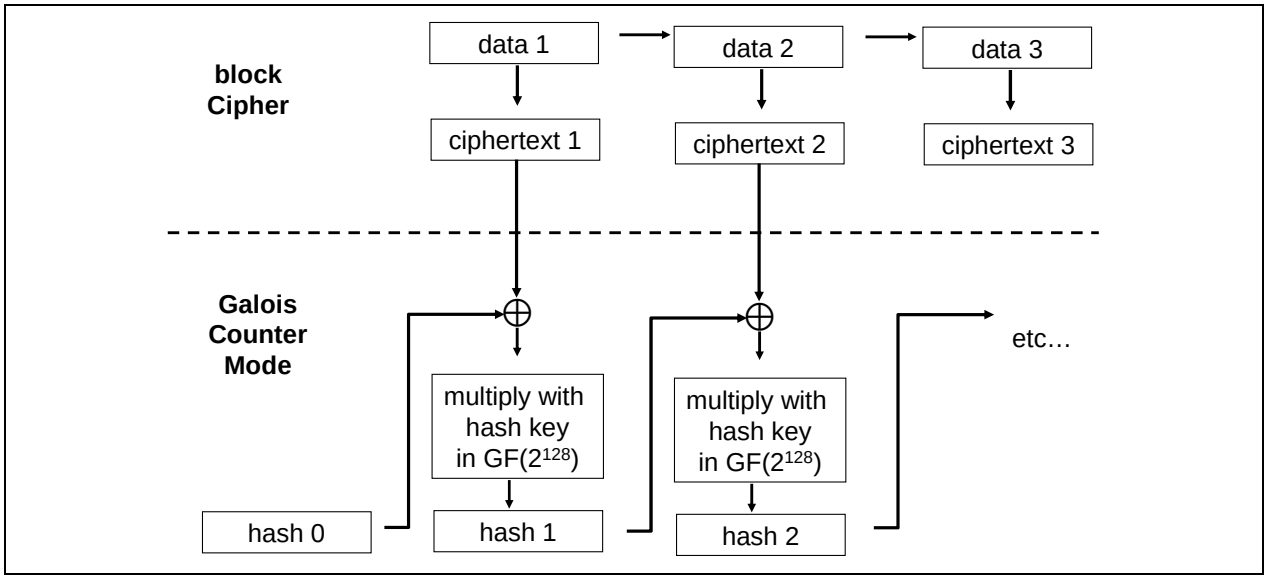
This section described the Galois Counter Mode and its current table lookup based software implementation (see for example [5] and www.openssl.org).

The Definition of GCM

The Galois Counter Mode is illustrated in Figure 1. This mode produces a message digest, called "Galois Hash'", from the encrypted data. This Galois Hash is used for high performance message authentication. In each step of the mode, the previous Galois Hash value is XOR-ed with the current ciphertext block. The result is then multiplied in

GF(2^{128}) with a hash key value. GCM uses GF(2^{128}) defined by the irreducible polynomial $g = g(x) = x^{128} + x^7 + x^2 + x + 1$.

Figure 1. The Galois Counter Mode



The multiplication in GF(2^{128}) involves carry-less multiplication of the two 128-bit operands, to generate a 256-bit result, followed by and reduction modulo the irreducible polynomial *g*.

Current Software Implementation of GCM

Current software implementations of the GCM mode use a table lookup based algorithm [5], shown in Figure 2 (for AES-GCM). This algorithm consists of two phases:

Preprocessing phase: generation of 16 lookup tables. Each table has 256 128-bit entries where entry *j* of table T_i stores the value (j * *hash key* * 2^{8i}) mod *g* for *j* =0, 1, …255, and *i* = 0, 1, …, 15

Run time phase: The algorithm takes the next ciphertext block and XOR-s it with the current value of the Galois Hash. The result (dynamic value) is multiplied with the Hash Key (fixed value) in GF(2^{128}). The GF(2^{128}) multiplication is carried out as follows: the value of the result is segmented into 16 8-bit slices. Subsequently, 16 table lookups are performed, using the slices, for indexing the tables. The results from the table lookups are XOR-ed with each other.

This algorithm performs operations on a per-byte basis: each 128-bit block involve 16 table lookups and 16 128-bit XOR operations.

This algorithm is not very efficient in software due to the cost of table lookups. It also suffers from potential side channel leakage based on memory access patterns (the accessed cache lines for the table lookup are data dependent).

Figure 2. Table Lookup-Based Implementation of AES-GCM

```
// Code snippet illustrating AES-GCM (AES-128)
// The AES round keys are assumed to be already expanded from the
cipher key.
//
           m128i enc_data[length];
          _m128i IV;
          _m128i counter;
          _m128i roundkeys[11];
          _m128i temp;
          _m128i galoishash;
          _m128i tables[16][256]
int i;
initall();
init_galoishash();
copy128(counter, IV);
for(i=0; i < length; i++)
           {
    copy128(temp, counter);
    xor128(temp, roundkeys[0]);
    for(j=0; j <9; j++)
            aes_encrypt_round(temp, roundkeys[j+1]);
    aes_encrypt_last_round(temp, roundkeys[10]);
    xor128(temp, data[i]);
    copy128(enc_data[i], temp);
    add128(counter, 1);
    xor128(galoishash, enc_data);
    uint8_t *bytes_gh = (uint8_t *)galoihash;
    copy128(temp, 0);
    for(j=0; j < 16; j++)
        xor128(temp, tables[j][bytes_gh[j]]);
       // tables[i][j] = j*hash_key*28*i mod x^128 + x^7 + x^2 + x + 1
                copy128(galoishash, temp);
             }
```

Efficient Algorithms for Computing GCM

This section describes several new methods for computing the Galois Counter Mode. This method is an extension of the algorithm described in [6, 7].

The method described here uses the 64-bit PCLMULQDQ instruction in place of the table lookup method, for computing 128-bit-by-128-bit carry-less products, and an efficient novel method for reducing the result modulo the irreducible polynomial g of the finite field GF(2^{128}). The proposed algorithms are carried out in two steps: carry-less multiplication and reduction modulo $g = x^{128} + x^7 + x^2 + x + 1$.

Generating Carry-less Multiplication of 128-bit Operands Using PCLMULQDQ

Denote the input operands by $[A_1:A_0]$ and $[B_1:B_0]$, where A_0, A_1, B_0 and B_1 are 64 bit long each. The proposed algorithm is the following:

The following algorithm can be viewed as "one iteration carry-less schoolbook'" multiplication.

Algorithm 1

Step 1: multiply carry-less the following operands: A_0 with B_0, A_1 with B_1, A_0 with B_1, and A_1 with B_0. Let the results of the above four multiplications be:

$$A_0 \bullet B_0 = [C_1 : C_0],\ \ A_1 \bullet B_1 = [D_1 : D_0],\ \ A_0 \bullet B_1 = [E_1 : E_0],\ \ A_1 \bullet B_0 = [F_1 : F_0]$$

Step 2: construct the 256-bit output of the multiplication $[A_1:A_0] \bullet [B_1:B_0]$ as follows:

$$[A_1 : A_0] \bullet [B_1 : B_0] = [D_1 : F_1 \oplus E_1 \oplus D_0 : F_0 \oplus E_0 \oplus C_1 : C_0] \qquad (5)$$

An alternative technique trades-off one multiplication for additional XOR operations. It can be viewed as "one iteration carry-less Karatsuba" multiplication [7, 9].

Algorithm 2

Step 1: multiply carry-less the following operands: A_1 with B_1, A_0 with B_0, and $A_0 \oplus A_1$ with $B_0 \oplus B_1$. Let the results of the above three multiplications be: $[C_1:C_0]$, $[D_1:D_0]$ and $[E_1:E_0]$, respectively.

Step 2: construct the 256-bit output of the multiplication $[A_1:A_0] * [B_1:B_0]$ as follows:

$$[A_1 : A_0] \bullet [B_1 : B_0] = [C_1 :\ C_0 \oplus C_1 \oplus D_1 \oplus E_1 : D_1 \oplus C_0 \oplus D_0 \oplus E_0 : D_0] \qquad (6)$$

Efficient Reduction Algorithm

To reduce a 256-bit carry-less product modulo a polynomial g of degree 128, we first split it into two 128-bit halves. The least significant half is simply XOR-ed with the final remainder (since the degree of g is 128).

For the most significant part, we develop an algorithm that realizes division via two multiplications. This algorithm can be seen as an extension of the Barrett reduction algorithm [1] to modulo-2 arithmetic, or as an extension of the Feldmeier CRC generation algorithm [3] to dividends and divisors of arbitrary size.

Since we do not need to take into account the least significant half of the input (see above), we investigate the efficient generation of a remainder $p(x)$ defined as follows:

$$p(x) = c(x) \cdot x^t \bmod\ g(x) \qquad (7)$$

Where,

1. $c(x)$ is a polynomial of degree s-1 with coefficients in GF(2), representing the most significant bits of the carry-less product. (for GCM, s =128).
2. t is the degree of the polynomial g. (for GCM, t= 128).
3. $g(x)$ is the irreducible polynomial defining the final field (for GCM, g = $g(x) = x^{128} + x^7 + x^2 + x + 1$).

For the polynomials $p(x)$, $c(x)$, and $g(x)$ we write:

$$c(x) = c_{s-1}x^{s-1} + c_{s-2}x^{s-2} + \ldots + c_1 x + c_0\,, \qquad (8)$$

$$p(x) = p_{t-1}x^{t-1} + p_{t-2}x^{t-2} + ... + p_1x + p_0 \text{ , and}$$

$$g(x) = g_t x^t + g_{t-1}x^{t-1} + ... + g_1x + g_0$$

Hereafter, we use the notation $L^u(v)$ to denote the coefficients of the u least significant terms of the polynomial v and $M^u(v)$ to denote the coefficients of its u most significant terms. The polynomial $p(x)$ can be expressed as:

$$p(x) = c(x) \cdot x^t \mod g(x) = g(x) \cdot q(x) \mod x^t \tag{9}$$

where $q(x)$ is a polynomial of degree s-1 equal to the quotient from the division of $c(x) \cdot x^t$ with g. The intuition behind equation (9) is that the t least significant terms of the dividend $c(x) \cdot x^t$ equal zero. Further, the dividend $c(x) \cdot x^t$ can be expressed as the sum of the polynomials $g \cdot q$ and p:

$$c(x) \cdot x^t = g(x) \cdot q(x) + p(x) \tag{10}$$

where operator '+' means XOR ('⊕'). From equation (10) one can expect that the t least significant terms of the polynomial $g \cdot q$ are equal to the terms of the polynomial p. Only if these terms are equal to each other, the result of the XOR operation $g \cdot q \oplus p$ is zero for its t least significant terms. Hence:

$$p(x) = g(x) \cdot q(x) \mod x^t = L^t(g(x) \cdot q(x)) \tag{11}$$

Now we define

$$g(x) = g_t x^t + g^*(x) \tag{12}$$

The polynomial g^* represents the t least significant terms of the polynomial g. Obviously,

$$p(x) = L^t(g(x) \cdot q(x)) = L^t(q(x) \cdot g^*(x) + q(x) \cdot g_t x^t) \tag{13}$$

However, the t least significant terms of the polynomial $q \cdot g_t \cdot x^t$ are zero. Therefore,

$$p(x) = L^t(q(x) \cdot g^*(x)) \tag{14}$$

From Equation (14) it follows that in order to compute the remainder p we need to know the value of the quotient q. The quotient can be calculated in a similar manner as in the Barrett reduction algorithm:

$$(9) \Leftrightarrow c(x) \cdot x^{t+s} = g(x) \cdot q(x) \cdot x^s + p(x) \cdot x^s \tag{15}$$

Let

$$x^{t+s} = g(x) \cdot q^+(x) + p^+(x) \tag{16}$$

where q^+ is an s-degree polynomial equal to the quotient from the division of x^{t+s} with g and p^+ is the remainder from this division. The degree of the polynomial p^+ is t-1. From equations (15) and (16) we get

$$\left.\begin{matrix}(15)\\(16)\end{matrix}\right\} \Leftrightarrow c(x)\cdot g(x)\cdot q^{+}(x)+c(x)\cdot p^{+}(x) = g(x)\cdot q(x)\cdot x^{s}+p(x)\cdot x^{s} \tag{17}$$

and

$$(17) \Rightarrow M^{s}(c(x)\cdot g(x)\cdot q^{+}(x)+c(x)\cdot p^{+}(x)) = M^{s}(g(x)\cdot q(x)\cdot x^{s}+p(x)\cdot x^{s}) \tag{18}$$

One can see that the polynomials $c\cdot g\cdot q^{+}$ and $g\cdot q\cdot x^{s}$ are of degree $t+2\cdot s-1$ the polynomial $c\cdot p^{+}$ is of degree $t+s-2$, and the polynomial $p\cdot x^{s}$ is of degree $t+s-1$. As a result the s most significant terms of the polynomials in the left and right hand side of equation (18) are not affected by the polynomials $c\cdot p^{+}$ and $p\cdot x^{s}$. Hence,

$$(18) \Leftrightarrow M^{s}(c(x)\cdot g(x)\cdot q^{+}(x)) = M^{s}(g(x)\cdot q(x)\cdot x^{s}) \tag{19}$$

Next, we observe that the s most significant terms of the polynomial $c\cdot g\cdot q^{+}$ equal to the s most significant terms of the polynomial $g\cdot M^{s}(c\cdot q^{+})\cdot x^{s}$. The polynomial $M^{s}(c\cdot q^{+})\cdot x^{s}$ results from $c\cdot q^{+}$ by replacing the s least significant terms of this polynomial with zeros. The intuition behind this observation is the following: the s most significant terms of the polynomial $c\cdot g\cdot q^{+}$ are calculated by adding the s most significant terms of the polynomial $c\cdot q^{+}$ with each other in as many offset positions as defined by the terms of the polynomial g. Thus, the s most significant terms of $c\cdot g\cdot q^{+}$ do not depend on the s least significant terms of $c\cdot q^{+}$, and consequently,

$$(19) \Leftrightarrow M^{s}(g(x)\cdot M^{s}(c(x)\cdot q^{+}(x))\cdot x^{s}) = M^{s}(g(x)\cdot q(x)\cdot x^{s}) \tag{20}$$

Equation (20) is satisfied for q given by

$$q = M^{s}(c(x)\cdot q^{+}(x)) \tag{21}$$

Since there is a unique quotient q satisfying equation (10) one can show that there is a unique quotient q satisfying equation (20). As a result this quotient q must be equal to $M^{s}(c(x)\cdot q^{+}(x))$.

It follows that the polynomial p is found by

$$p(x) = L^{t}(g^{*}(x)\cdot M^{s}(c(x)\cdot q^{+}(x))) \tag{22}$$

Equation (22) indicates the algorithm for computing the polynomial p.

Algorithm 3:

Preprocessing: For the given irreducible polynomial g the polynomials g^{*} and q^{+} are computed first. The polynomial g^{*} is of degree $t-1$ consisting of the t least significant

terms of g, whereas the polynomial q^+ is of degree s and is equal to the quotient of the division of x^{t+s} with the polynomial g.

Calculation of the remainder polynomial:

Step 1: The input c is multiplied with q^+. The result is a polynomial of degree $2s$-1.

Step 2: The s most significant terms of the polynomial resulting from step 1 are multiplied with g^*. The result is a polynomial of degree $t+s$-2.

Step 3: The algorithm returns the t least significant terms of the polynomial resulting from step 2. This is the desired remainder.

Application of the method for reduction modulo $x^{128} + x^7 + x^2 + x + 1$

One can see that the quotient from the division of x^{256} with g is g itself. The polynomial g = $g(x) = x^{128} + x^7 + x^2 + x + 1$ contains only 5 non-zero coefficients (therefore also called "pentanomial"). This polynomial can be represented as the bit sequence [1:<*120 zeros*>:10000111]. Multiplying this carry-less with a 128-bit value and keeping the 128 most significant bit can be obtained by: (i) Shifting the 64 most significant bits of the input by 63, 62 and 57-bit positions to the right. (ii) XOR-ing these shifted copies with the 64 least significant bits of the input. Next, we carry-less multiply this 128-bit result with g, and keep the 128 least significant bits. This can be done by: (i) shifting the 128-bit input by 1, 2 and 7 positions to the left. (ii) XOR-ing the results. Algorithm 4 provides a detailed description of the reduction algorithm.

Algorithm 4

Denote the input operand by $[X_3:X_2:X_1:X_0]$ where X_3, X_2, X_1 and X_0 are 64 bit long each.

Step 1: shift X_3 by 63, 62 and 57-bit positions to the right. Compute the following numbers:

$$\begin{aligned} A &= X_3 >> 63 \\ B &= X_3 >> 62 \\ C &= X_3 >> 57 \end{aligned} \tag{23}$$

Step 2: We XOR A, B, and C with X_2. We compute a number D as follows:

$$D = X_2 \oplus A \oplus B \oplus C \tag{24}$$

Step 3: shift $[X_3:D]$ by 1, 2 and 7 bit positions to the left. Compute the following numbers:

$$\begin{aligned} [E_1 : E_0] &= [X_3 : D] << 1 \\ [F_1 : F_0] &= [X_3 : D] << 2 \\ [G_1 : G_0] &= [X_3 : D] << 7 \end{aligned} \tag{25}$$

Step 4: XOR $[E_1:E_0]$, $[F_1:F_0]$, and $[G_1:G_0]$ with each other and $[X_3:D]$. Compute a number $[H_1:H_0]$ as follows:

$$[H_1 : H_0] = [X_3 \oplus E_1 \oplus F_1 \oplus G_1 : D \oplus E_0 \oplus F_0 \oplus G_0] \tag{26}$$

Return $[X_1 \oplus H_1 : X_0 \oplus H_0]$.

Bit Reflection Peculiarity of GCM

Special peculiarity should be taken into account when implementing the GCM mode, because the standard specifies that the bits inside their 128-bit double-quad-words are reflected. That is, the bit corresponding to the least significant coefficient of the polynomial representation of the entities which are multiplied is bit number 127 rather than bit number 0. This also implies that the order of bits in the reduction polynomial is [11100001:<*120 zeros*>:1] as opposed to [1:<*120 zeros*>:10000111]. Note that this property is not merely the difference between Little Endian and Big Endian notations.

To handle this peculiarity, we point out the following fundamental property of carry-less multiplication, namely

$$reflected\ (A) \bullet reflected\ (B) = reflected\ (A \bullet B) >> 1 \tag{27}$$

Therefore, the PCLMULQDQ instruction can be used for performing multiplication in the finite field GF(2^{128}) seamlessly, regardless on the representation of the input and the output operands.

Algorithm 5 outlines the required modification of the reduction algorithm that accommodates bit reflection of the inputs and the outputs in the GCM mode.

Algorithm 5

Denote the input operand by $[X_3:X_2:X_1:X_0]$ where X_3, X_2, X_1 and X_0 are 64 bit long each.

Step 1: compute

$$[X_3X_2X_1X_0] = [X_3X_2X_1X_0] << 1 \tag{28}$$

Step 2: shift X_0 by 63, 62 and 57 bit positions to the left. We compute the following numbers:

$$\begin{aligned} A &= X_0 << 63 \\ B &= X_0 << 62 \\ C &= X_0 << 57 \end{aligned} \tag{29}$$

Step 2: XOR *A*, *B*, and *C* with X_1. Compute a number *D* as follows:

$$D = X_1 \oplus A \oplus B \oplus C \tag{30}$$

Step 3: shift $[D:X_0]$ by 1, 2 and 7 bit positions to the right. Compute the following numbers:

$$\begin{aligned} [E_1 : E_0] &= [D : X_0] >> 1 \\ [F_1 : F_0] &= [D : X_0] >> 2 \\ [G_1 : G_0] &= [D : X_0] >> 7 \end{aligned} \tag{31}$$

Step 4: XOR $[E_1:E_0]$, $[F_1:F_0]$, and $[G_1:G_0]$ with each other and $[D:X_0]$. Compute a number $[H_1:H_0]$ as follows:

$$[H_1 : H_0] = [D \oplus E_1 \oplus F_1 \oplus G_1 : X_0 \oplus E_0 \oplus F_0 \oplus G_0] \quad (32)$$

Return $[X_3 \oplus H_1 : X_2 \oplus H_0]$.

Implementation Using Linear Folding

Linear folding is the mathematical operation of replacing a number of most significant bits of a quantity with the product of these bits times a constant during a reduction. Folding helps with speeding up reduction because it decreases the length of the quantity which is being reduced at the expense of the number of multiplications needed. In what follows we assume that all operations that take place are carry-less, i.e., in GF(2) arithmetic.

Suppose that the quantity to be reduced can be expressed as the sum of two polynomials:

$$p(x) = (c(x) \cdot x^t + d(x)) \bmod g(x) \quad (33)$$

Where,

1. $c(x)$ is a polynomial of degree s-1 with coefficients in GF(2), representing the most significant bits of the quantity to be reduced
2. t-1 is the length of the degree of $d(x)$
3. $g(x)$ is the irreducible polynomial defining the field (for GCM, g = $g(x) = x^{128} + x^7 + x^2 + x + 1$).

For the polynomial $p(x)$ we write:

$$\begin{aligned} &p(x) = (c(x) \cdot x^t + d(x)) \bmod g(x) = \\ &c(x) \cdot x^t \bmod g(x) + d(x) \bmod g(x) = \\ &(c(x) \cdot (x^t \bmod g(x)) + d(x)) \bmod g(x) \end{aligned} \quad (34)$$

The quantity $x^t \bmod g(x)$ depends only on the reduction polynomial. Hence it can be treated as a constant. Equation (34) indicates a method for performing reduction which is called linear folding and works as follows:

Step 1: The polynomial *c(x)* is multiplied carry-less with the constant $x^t \bmod g(x)$.

Step 2: The result of the multiplication is XOR-ed with $d(x)$.

Step 3: The reduction proceeds using any known technique.

The remainder from the division of x^t with $g(x) = x^{128} + x^7 + x^2 + x + 1$ is the bit sequence <10000111: *t*-128 zeros>. Since $t+s = 255$ one can see that the length of the carry-less multiplication of $c(x)$ with $x^t \bmod g(x)$ is 134 and is independent of the choice of t and s.

In GCM all operations are bit reflected, so folding needs to be implemented in a bit reflected manner too. The designer of an algorithm based on linear folding has several degrees of freedom that depend on the choice of t and s. If the length of the folding quantity $c(x)$ spans a single 64-bit word then the cost of multiplication with $x^t \bmod g(x)$ can be potentially small equal to 1 or 2 64-bit carry-less multiplication operations. On the other hand the cost of further reducing the given polynomial after folding may be higher.

Another issue related to the design of a folding algorithm has to do with the fact that the reflected version of x^t mod $g(x)$ may span one or multiple words. Theoretically one can multiply $c(x)$ not with x^t mod $g(x)$ but with an appropriately shifted version of the bit sequence <11100001> so that the second operand of the multiplication spans exactly one 64 bit word. The result can then be corrected with further shift operations. There is one case for which the second operand of the multiplication spans one word and no further shifts are required: this is for t = 193. For this case, the subsequent reduction steps after folding requiring reducing a 193 bit quantity.

In what follows we describe four representative algorithms two for s = 120 and two for s = 64 bits. In each pair the second operand of the folding multiplication spans 1 or 2 words.

Algorithm 6: Folding length *s* = 120, two multiplications version

Denote the input operand by $[X_3:X_2:X_1:X_0]$ where X_3, X_2, X_1 and X_0 are 64 bit long each.

Step 1: Compute $[C_1:C_0]$ =$[X_1:X_0]$ AND 0x00ffffffffffffffffffffffffffff

Step 2: Compute $[H_2:H_1:H_0]$ =$[C_1:C_0]$ • 0xe100000000000000

Step 3: Shift $[H_2:H_1:H_0]$ by one bit position to the left

Step 4: XOR $[H_2:H_1:H_0:0]$ with $[X_3:X_2:X_1:X_0]$ and replace $[X_3:X_2:X_1:X_0]$

Step 5: Replace X_1 with the result of logical AND between X_1 and 0xff00000000000000

Step 6: Compute $A = X_1$ >> 63, $B = X_1$ << 1, C = (X_1 & 0x7f00000000000000) >> 1, D = (X_1 & 0x7f00000000000000) >> 6, $E = X_1$ & 0x7f00000000000000.

Step 7: Shift $[X_3:X_2:A]$ by one bit position to the left

Return $[X_3 \oplus B \oplus C \oplus D \oplus E:X_2]$.

An assembly implementation of this algorithm is shown in Figure 3.

Algorithm 7: Folding length *s* = 120, four multiplications version

Denote the input operand by $[X_3:X_2:X_1:X_0]$ where X_3, X_2, X_1 and X_0 are 64 bit long each.

Step 1: Compute $[C_1:C_0]$ =$[X_1:X_0]$ AND 0x00ffffffffffffffffffffffffffff

Step 2: Compute $[H_2:H_1:H_0]$ =$[C_1:C_0]$ • 0x1c200000000000000

Step 3: XOR $[H_2:H_1:H_0:0]$ with $[X_3:X_2:X_1:X_0]$ and replace $[X_3:X_2:X_1:X_0]$

Step 4: Replace X_1 with the result of logical AND between X_1 and 0xff00000000000000

Step 5: Compute $A = X_1$ >> 63, $B = X_1$ << 1, C = (X_1 & 0x7f00000000000000) >> 1, D = (X_1 & 0x7f00000000000000) >> 6, $E = X_1$ & 0x7f00000000000000.

Step 6: Shift $[X_3:X_2:A]$ by one bit position to the left

Return $[X_3 \oplus B \oplus C \oplus D \oplus E:X_2]$.

Algorithm 8: Folding length *s* = 64, single multiplication version

Denote the input operand by $[X_3:X_2:X_1:X_0]$ where X_3, X_2, X_1 and X_0 are 64 bit long each.

Step 1: Compute $[H_1:H_0] = X_0 \bullet$ 0xe100000000000000

Step 2: Shift $[H_1:H_0]$ by one bit position to the left

Step 3: XOR $[0:H_1:H_0:0]$ with $[X_3:X_2:X_1:X_0]$ and replace $[X_3:X_2:X_1:X_0]$

Step 4: Compute $A = X_1 >> 63$, $B = X_1 << 1$, $C_1 = (X_1$ & 0x7f00000000000000) >> 1, $C_0 = X_1<<63$, $D_1 = (X_1$ & 0x7f00000000000000) >> 6, $D_0 = X_1<<58$, $E = X_1$ & 0x7f00000000000000,

Step 5: Shift $[X_3:X_2:A]$ by one bit position to the left

Return $[X_3 \oplus B \oplus C_1 \oplus D_1 \oplus E:X_2 \oplus D_0 \oplus C_0]$.

Algorithm 9: Folding length s = 64, two multiplications version

Denote the input operand by $[X_3:X_2:X_1:X_0]$ where X_3, X_2, X_1 and X_0 are 64 bit long each.

Step 1: Compute $[H_2:H_1:H_0] = X_0 \bullet$ 0x1c200000000000000

Step 2: XOR $[H_2:H_1:H_0:0]$ with $[X_3:X_2:X_1:X_0]$ and replace $[X_3:X_2:X_1:X_0]$

Step 3: Compute $A = X_1 >> 63$, $B = X_1 << 1$, $C_1 = (X_1$ & 0x7f00000000000000) >> 1, $C_0 = X_1<<63$, $D_1 = (X_1$ & 0x7f00000000000000) >> 6, $D_0 = X_1<<58$, $E = X_1$ & 0x7f00000000000000,

Step 4: Shift $[X_3:X_2:A]$ by one bit position to the left

Return $[X_3 \oplus B \oplus C_1 \oplus D_1 \oplus E:X_2 \oplus D_0 \oplus C_0]$.

Code Examples: GCM Computation

The following assembler code snippets illustrate the implementation of the Galois Field multiplication portion of GCM using the algorithms described in Efficient Algorithms for Computing GCM.

These examples use the Intel SSE version of PCLMULQDQ.

Figure 3. Code Sample for Performing GCM Using the Schoolbook Method

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;int sse_clmul_gcm()
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

sse_clmul_gcm PROC

; xmm0, xmm1 hold the values for the two operands which are carry-less multiplied

movdqu       xmm3, xmm0
pclmulqdq    xmm3, xmm1, 0x00   ;xmm3 holds a0*b0
movdqu       xmm4, xmm0
pclmulqdq    xmm4, xmm1, 0x10   ;xmm4 holds a0*b1
movdqu       xmm5, xmm0
pclmulqdq    xmm5, xmm1, 0x01   ;xmm5 holds a1*b0
```

```
movdqu       xmm6, xmm0
pclmulqdq    xmm6, xmm1, 0x11   ;xmm6 holds a1*b1
pxor         xmm4, xmm5       ;xmm4 holds a0*b1 + a1*b0
pshufd       xmm5, xmm4, 78     ;swap the 64 most and least significant bits of xmm4
movdqu       xmm4, xmm5
pand         xmm4, xmm14        ;xmm14 holds the mask
0x00000000000000000ffffffffffffffff
pand         xmm5, xmm15      ;xmm15 holds the mask
0xffffffffffffffff0000000000000000
  pxor             xmm3, xmm5
pxor         xmm6, xmm4       ; register pair <xmm6:xmm3> holds the result of
                                ; the carry-less multiplication of xmm0 by xmm1

;we shift the result of the multiplication by one bit position to the left cope for
the fact
;that bits are reversed
movdqu       xmm7, xmm3
movdqu       xmm8, xmm6
pslld        xmm3, 1
pslld        xmm6, 1
psrld        xmm7, 31
psrld        xmm8, 31
pshufd       xmm7, xmm7, 147
pshufd       xmm8, xmm8, 147
movdqu       xmm9,  xmm7
pand         xmm7, xmm13    ;xmm13 holds the mask 0xffffffffffffffffffffffff00000000
pand         xmm8, xmm13
pand         xmm9, xmm12    ;xmm12 holds the mask 0x00000000000000000000000ffffffff
por          xmm3, xmm7
por          xmm6, xmm8
por          xmm6, xmm9

;first phase of the reduction
Movdqu       xmm7, xmm3
Movdqu       xmm8, xmm3
Movdqu       xmm9, xmm3     ;move xmm3 into xmm7, xmm8, xmm9 in order to perform
                            ; the three shifts independently
pslld        xmm7, 31        ; packed right shifting << 31
pslld        xmm8, 30        ; packed right shifting shift << 30
pslld        xmm9, 25        ; packed right shifting shift << 25
pxor         xmm7, xmm8      ;xor the shifted versions
pxor         xmm7, xmm9

pshufd       xmm8, xmm7, 57  ;move the least significant 32-bit word to
                             ; the most significant word position
                             ; and right shift all other three 32-bit words by 32
bits
movdqu       xmm7, xmm8
pand         xmm7, xmm11     ;xmm11 holds the mask
0xffffffff00000000000000000000000
pxor         xmm3, xmm7      ;first phase of the reduction complete

;second phase of the reduction
movdqu      xmm2,xmm3    ; make 3 copies of xmm3 (in in xmm10, xmm11, xmm12)
                          ; for doing three shift operations
Movdqu       xmm4,xmm3
Movdqu       xmm5,xmm3
psrld        xmm2,1       ; packed left shifting >> 1
```

```
psrld        xmm4,2       ; packed left shifting >> 2
psrld        xmm5,7       ; packed left shifting >> 7
pxor         xmm2,xmm4    ; xor the shifted versions
pxor         xmm2,xmm5
movdqu       xmm5, xmm11
pandn        xmm5, xmm8
pxor         xmm2,xmm5
pxor         xmm3, xmm2
pxor         xmm6, xmm3    ;the result is in xmm6
RET

sse_gfmul_gcm  ENDP
```

Figure 4. Code Sample for Performing GCM Using the Karatsuba Method

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;int sse_clmul_ka_gcm()
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

sse_clmul_gcm PROC

; xmm0, xmm1 hold the values for the two operands which are carry-less  multiplied

movdqu       xmm3, xmm0
pclmulqdq    xmm3, xmm1, 0x00   ;xmm3 holds a0*b0
movdqu       xmm6, xmm0
pclmulqdq    xmm6, xmm1, 0x11   ;xmm6 holds a1*b1
pshufd       xmm4, xmm0, 78
  pshufd        xmm5, xmm1, 78
pxor         xmm4, xmm0
pxor         xmm5, xmm1
pclmulqdq    xmm4, xmm5, 0x00   ;xmm4 holds (a1+a0)*(b1+b0)
pxor         xmm4, xmm3
pxor         xmm4, xmm6        ;xmm4 holds a0*b1 + a1*b0
pshufd       xmm5, xmm4, 78     ;swap the 64 most and least significant bits of xmm4
movdqu       xmm4, xmm5
pand         xmm4, xmm14        ;xmm14 holds the mask
0x00000000000000007fffffffffffffff
pand         xmm5, xmm15       ;xmm15 holds the mask
0xffffffffffffffff0000000000000000
pxor         xmm3, xmm5
pxor         xmm6, xmm4        ; register pair <xmm6:xmm3> holds the result of
                                 ; the carry-less multiplication of xmm0 by xmm1

;we shift the result of the multiplication by one bit position to the left cope for
the fact
;that bits are reversed
movdqu       xmm7, xmm3
movdqu       xmm8, xmm6
pslld        xmm3, 1
pslld        xmm6, 1
psrld        xmm7, 31
psrld        xmm8, 31
pshufd       xmm7, xmm7, 147
pshufd       xmm8, xmm8, 147
```

```
movdqu        xmm9,  xmm7
pand          xmm7, xmm13     ;xmm13 holds the mask 0xffffffffffffffffffffffff00000000
pand          xmm8, xmm13
pand          xmm9, xmm12     ;xmm12 holds the mask 0x00000000000000000000000ffffffff
por           xmm3, xmm7
por           xmm6, xmm8
por           xmm6, xmm9

;first phase of the reduction
Movdqu        xmm7, xmm3
Movdqu        xmm8, xmm3
Movdqu        xmm9, xmm3   ;move xmm3 into xmm7, xmm8, xmm9 in order to perform
                           ; the three shifts independently
pslld        xmm7, 31        ; packed right shifting << 31
pslld        xmm8, 30        ; packed right shifting shift << 30
pslld        xmm9, 25        ; packed right shifting shift << 25
pxor         xmm7, xmm8      ;xor the shifted versions
pxor         xmm7, xmm9

pshufd       xmm8, xmm7, 57  ;move the least significant 32-bit word to
                             ; the most significant word position
                             ; and right shift all other three 32-bit words by 32
bits
movdqu       xmm7, xmm8
pand         xmm7, xmm11   ;xmm11 holds the mask 0xffffffff0000000000000000000000000
pxor         xmm3, xmm7    ;first phase of the reduction complete

;second phase of the reduction
movdqu      xmm2,xmm3    ; make 3 copies of xmm3 (in in xmm10, xmm11, xmm12)
                          ; for doing three shift operations
movdqu      xmm4,xmm3
movdqu      xmm5,xmm3
psrld        xmm2,1       ; packed left shifting >> 1
psrld        xmm4,2       ; packed left shifting >> 2
psrld        xmm5,7      ; packed left shifting >> 7
pxor         xmm2,xmm4   ; xor the shifted versions
pxor         xmm2,xmm5
movdqu       xmm5, xmm11
pandn        xmm5, xmm8
pxor         xmm2,xmm5
pxor         xmm3, xmm2
pxor         xmm6, xmm3    ;the result is in xmm6
RET

sse_gfmul_ka_gcm  ENDP
```

Figure 5. Code Sample for Performing GCM Using Linear Folding

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
;int sse_clmul_gcm_folding()
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

sse_clmul_gcm_folding PROC

; xmm0, xmm1 hold the values for the two operands which are carry-less  multiplied
```

```
movdqu       xmm3, xmm0
pclmulqdq    xmm3, xmm1, 0x00   ;xmm3 holds a0*b0
movdqu       xmm4, xmm0
pclmulqdq    xmm4, xmm1, 0x10   ;xmm4 holds a0*b1
movdqu       xmm5, xmm0
pclmulqdq    xmm5, xmm1, 0x01   ;xmm5 holds a1*b0
movdqu       xmm6, xmm0
pclmulqdq    xmm6, xmm1, 0x11   ;xmm6 holds a1*b1
pxor         xmm4, xmm5       ;xmm4 holds a0*b1 + a1*b0
pshufd       xmm5, xmm4, 78     ;swap the 64 most and least significant bits of xmm4
movdqu       xmm4, xmm5
pand         xmm4, xmm14        ;xmm14 holds the mask
0x00000000000000000ffffffffffffffff
pand         xmm5, xmm15        ;xmm15 holds the mask
0xffffffffffffffff0000000000000000
pxor         xmm3, xmm5
pxor         xmm6, xmm4         ; register pair <xmm6:xmm3> holds the result of
                                ;the carry-less multiplication of xmm0 by xmm1
;first we grab 120 least significant bits to accomplish the linear folding
movdqu       xmm2, xmm3
pand         xmm2, xmm13   ;xmm13 holds the mask 0x00ffffffffffffffffffffffffffff
movdqu       xmm4, xmm2
;then we do two multiplies
Pclmulqdq    xmm4, xmm11, 0x00   ;xmm12 holds 0x0000000000000000e100000000000000
pclmulqdq    xmm2, xmm11, 0x01
pshufd       xmm5, xmm4, 78      ;swap the 64 most and least significant bits of
xmm4
movdqu       xmm4, xmm5
pand         xmm4, xmm14         ;xmm14 holds the mask
0x00000000000000000ffffffffffffffff
pand         xmm5, xmm15       ;xmm15 holds the mask
0xffffffffffffffff0000000000000000
pshufd       xmm15, xmm15, 232
pxor         xmm2, xmm4          ;the result from multiplication is in the pair
<xmm2:xmm5>

;then we shift <xmm2:xmm5> by 1 bit position to the left
movdqu       xmm8, xmm2
movdqu       xmm9, xmm5
pslld        xmm2, 1
pslld        xmm5, 1
psrld        xmm8, 31
psrld        xmm9, 31
pshufd       xmm8, xmm8, 147
pshufd       xmm9, xmm9, 147
pand         xmm8, xmm15    ;xmm15 holds the mask 0xffffffffffffffffffffffff00000000
pand         xmm9, xmm12    ;xmm12 holds the mask 0x00000000000000000000000000000000
por          xmm2, xmm8
por          xmm2, xmm9

;we xor with <xmm6:xmm3>
pxor        xmm6, xmm2
pxor        xmm3, xmm5
movdqu      xmm4, xmm13
pandn       xmm4, xmm3

;folding is complete: the result is in the pair <xmm6:xmm4>
```

```
;second phase of the reduction
movdqu      xmm3, xmm4
psrld       xmm3, 31
pshufd      xmm3, xmm3, 147
pand        xmm4, xmm10      ;xmm10 holds 0x7f000000000000000000000000000000
movdqu      xmm5, xmm4
movdqu      xmm7, xmm4
movdqu      xmm8, xmm4
pslld       xmm5, 1
psrld       xmm7, 1
psrld       xmm8, 6

;one more shift to the left for xmm6
Movdqu       xmm2, xmm6
pslld        xmm6, 1
psrld        xmm2, 31
pshufd       xmm2, xmm2, 147
pand         xmm2, xmm15  ;xmm15 holds the mask 0xffffffffffffffffffffffff00000000
por          xmm6, xmm2
por          xmm6, xmm3
pxor         xmm6, xmm4
pxor         xmm7, xmm8
pxor         xmm6, xmm5
pxor         xmm6, xmm7
;the result is in xmm6
RET

sse_gfmul_gcm_folding  ENDP
```

Summary

We present Intel's PCLMULQDQ, a new Single Instruction Multiple Data (SIMD) instruction that will be introduced in the next generation of Intel® processor, as of 2009. PCLMULQDQ performs carry-less multiplication of two 64-bit operands. This paper provides information on using this instruction for computing the Galois Counter Mode. Several algorithms are described that take into account the fact that the irreducible polynomial specified by the binary field of GCM is sparse. Carry-less multiplication is performed using the instruction PCLMULQDQ as well as the schoolbook and Karatsuba algorithms. Reduction is performed using a small number of shift and XOR operations as well as linear folding. The bit reflection peculiarity of GCM is taken into account.

Appendix A: Test Vectors

This appendix provides a few test vectors. Little Endian notation is used unless specified otherwise.

PCLMULQDQ Test Vectors

All 4 test vectors have the same xmm1 and xmm2 input, and vary only in the value of the immediate byte. This way, the result of all of the four combinations, High/Low from xmm1 and xmm2, are generates. Only bits [4] and [0] of the immediate byte are used for selecting the multiplicands.

Input to the PCLMULQDQ instruction

```
xmm1:= 7b5b546573745666563746f725d53475d
xmm2:= 48692853686179295b477565726f6e5d

The corresponding 64-bit Low/High halves are
xmm1High:= 7b5b546573745665   xmm1Low:= 63746f725d53475d
xmm2High:= 4869285368617929   xmm2Low:= 5b477565726f6e5d

Test Vector 1:
immediate = 0x00; carry-less multiply xmm2Low by xmm1Low
PCLMULQDQ result:= 1d4d84c85c3440c0929633d5d36f0451

Test Vector 2:
immediate = 0x10; carry-less multiply xmm2High by xmm1Low
PCLMULQDQ result:= 1bd17c8d556ab5a17fa540ac2a281315

Test Vector 3:
immediate = 0x01; carry-less multiply xmm2Low by xmm1High
PCLMULQDQ result:= 1a2bf6db3a30862fbabf262df4b7d5c9

Test Vector 4:
immediate = 0x11; carry-less multiply xmm2High by xmm1High
PCLMULQDQ result:= 1d1e1f2c592e7c45d66ee03e410fd4ed
```

Multiplication in GF(2^{128}) Defined by the Polynomial $x^{128} + x^7 + x^2 + x + 1$

Input to the PCLMULQDQ instruction

```
xmm1 := 7b5b546573745666563746f725d53475d
xmm2 := 48692853686179295b477565726f6e5d
Product := 40229a09a5ed12e7e4e10da323506d2
```

(Product in GF(2^{128}) defined by the reduction polynomial $g = g(x) = x^{128} + x^7 + x^2 + x + 1$)

GCM Test Vector

The following test vector takes into account the special peculiarity of GCM which specifies that the bits of the input operands (data and hash keys) and the output operand should be reflected

```
Data:                    0x952b2a56a5604ac0b32b6656a05b40b6
Hash Key:                0xdfa6bf4ded81db03ffcaff95f830f061
Multiplication Result:   0xda53eb0ad2c55bb64fc4802cc3feda60
```

References

[1] P. Barrett, *Implementing the Rivest, Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor*, Master's Thesis, University of Oxford, UK, 1986.

[2] *Digital Signature Standard*, Federal Information Processing Standard Publication FIPS 186-2, January 27, 2000, available at: http://csrc.nist.gov/publications/fips.

[3] D. Feldmeier, *Fast Software Implementation of Error Correcting Codes*, IEEE Transactions on Networking, December, 1995. [1] M. Dworkin,

Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication, Federal Information Processing Standard Publication FIPS 800-38D, April 20, 2006, available at: http://csrc.nist.gov/publications/drafts.html.

[4] *The Fibre Channel Security Protocols Project*, ISO-T11 Committee Archive, available at: http://www.t11.org/

[5] Brian Gladman, *AES and Combined Encryption/Authentication Modes*, Public Domain Source Code, available at: http://fp.gladman.plus.com/AES/index.htm [2] *IEEE 802.1AE - Media Access Control (MAC) Security*, IEEE 802.1 MAC Security Task Group Document, available at: http://www.ieee802.org/1/pages/802.1ae.html.

[6] S. Gueron, and M. E. Kounavis, *Efficient Implementation of the Galois Counter Mode Using a Carry-less Multiplier and a Fast Reduction Method*, Information Processing Letters, submitted, 2007.

[7] S. Gueron, and M. E. Kounavis, *Efficient Implementation of the Galois Counter Mode in the SSE Domain*, Technical Report, Intel Corporation, 2007.

[8] *IEEE Project 1619.1 Home*, available at: http://ieee-p1619.wetpaint.com/page/IEEE+Project+1619.1+Home.

[9] A. Karatsuba and Y. Ofman, *Multiplication of Multidigit Numbers on Automata*, Soviet Physics – Doklady, Vol. 7, pg 595-596, 1963.

[10] C. Koc and T. Acar, *Montgomery Multiplication in GF(2^k)*, Designs, Codes and Cryptography, Vol. 14, No. 1, pages 57-69, April 1998.

[11] C. H. Lim and P. J. Lee, *More Flexible Exponentiation with Precomputation*, Advances in Cryptography (CRYPTO'94), pages 95-107, 1997.

[12] J. Lopez and R. Dahab, *High Speed Software Multiplication in GF2^m*, Lecture Notes in Computer Science, Vol. 1977, pages 203-212, 2000.

[13] A. Menezes, P. Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[14] J. Salowey, A. Choudhury and D. McGrew, *RSA-based AES-GCM Cipher Suites for TLS*, available at: http://www1.ietf.org/mail-archive/web/tls/current/threads.html.

[15] J. Viega and D. McGrew, *The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP)*, IETF RFC 4106, available at: http://www.rfc-archive.org/.

About the Authors

Shay Gueron is a Security Architect in the Mobility Group at Intel Corporation, working at the Israel Design Center. His interests include applied security, cryptography, and algorithms. He holds a Ph.D. in applied mathematics from Technion—Israel Institute of Technology, and is also an Associate Professor at the Department of Mathematics, Faculty of Science and Science Education, at the University of Haifa, Israel.

Michael E. Kounavis is a Senior Research Scientist at Intel Corporation. He joined Intel in 2003. Since then he has worked on developing novel algorithms and solutions for line rate packet processing, security and data integrity. His current research focuses on accelerating cryptographic algorithms and protocols. Prior to joining Intel, he worked on programming network architectures at Columbia University. He has published over 30 papers in leading technical journals and conferences and he is a regular program committee member of the IEEE ISCC, SPECTS and IWDDS conferences.

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

This white paper, as well as the software described in it, is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Intel processor numbers are not a measure of performance. Processor numbers differentiate features within each processor family, not across different processor families. See www.intel.com/products/processor_number for details.

The Intel processor/chipset families may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Copies of documents, which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Intel and the Intel Logo are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2008, Intel Corporation. All rights reserved.