![intel]

White Paper

Shay Gueron
Intel Architecture Group, Israel Development Center
Intel Corporation

Michael E. Kounavis
Intel Labs, Circuits and Systems Research
Intel Corporation

Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode

The Intel® PCLMULQDQ instruction is a new instruction available beginning with the all new 2010 Intel® Core™ processor family based on the 32nm Intel® microarchitecture codename Westmere. The PCLMULQDQ instruction performs carry-less multiplication of two 64-bit operands.

This paper provides information on the instruction, and its usage for computing the Galois Hash. It also provides code examples for the usage of PCLMULQDQ, together with the Intel® AES New Instructions for efficient implementation of AES in Galois Counter Mode (AES-GCM).

This version of the paper also provides high performance code examples for AES-GCM, and discloses, for the first time, their measured performance numbers.

Contents

Figures

Tables

§

Introduction

PCLMULQDQ is a new processor instruction that Intel is introducing in the 2010 Intel®
Core™ processor family based on the 32nm Intel® microarchitecture codename
Westmere, entering production starting the end of 2009. It computes the carry-less
product of two 64-bit operands.

This paper provides details the PCLMULQDQ instruction, and describes several
algorithms for using it for computing the Galois Hash, which is the underlying
computation of the Galois Counter Mode (GCM).

An important usage model is AES in Galois counter Mode (AES-GCM), where the AES
encryption/decryption part can be implemented efficiently using the Intel AES New
Instructions which are also being introduced (see Reference [17],
http://software.intel.com/en-us/articles/advanced-encryption-standard-aes-instructions-set/
for details) . This paper provides code examples for AES-GCM, using the new
instructions, and also discloses their resulting performance.

This version of the paper provides high performance code examples for AES-GCM, and
discloses, for the first time, their measured performance numbers.

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 a
relatively time consuming operation when implemented with the current ISA of the IA
architecture. For example, software implementation of carry-less multiplication that
uses one of the best-known methods (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 performs carry-less multiplication of 128-bit operands, producing a 255-bit
product. This is the first step of computing a 'Galois hash', which is part of the GCM
mode. The PCLMULQDQ instruction computes the 127-bit product of two 64-bit
operands. It can be used by software as a building block for generating the 255-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
PCLMULQDQ instruction, together with this algorithm speeds up the GCM mode.

Carry-less multiplication is also in the core computation of Elliptic Curve Cryptography (ECC) over binary fields [2] and Cyclic Redundancy Checks (CRCs). The carry-less multiplication instruction PCLMULQDQ 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

$$A = [a_{n-1} \ a_{n-2} \ ... \ a_0], \ B = [b_{n-1} \ b_{n-2} \ ... \ b_0] \tag{1}$$

and let the carry-less multiplication result be the following 2n-1 bit array:

$$C = [c_{2n-2} \ c_{2n-2} \ ... \ c_0] \tag{2}$$

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

$$c_i = \bigoplus_{j=0}^{i} a_j b_{i-j} \tag{3}$$

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

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

for $n \le i \le 2n$-2.

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., $a \cdot (b+c) = a \cdot b + a \cdot c$).

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 is [1000]. This result is obtained as follows: Performing the carry-less multiplication C = A • B results in [01100010]. This is 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 bytes) of xmm1 by one quadword (8 bytes) of xmm2/m128, returning a double quadword (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]

NOTE:
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 follows.

```
IF imm8[0] == 0 THEN
        Temp1 = xmm1[63:0]
ELSE
        Temp1 = xmm1[127:64]
ENDIF
IF imm8[4] == 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;
```

The AVX (Nondestructive Destination) Variant VPCLMULQDQ

The carry-less multiplication instruction PCLMULQDQ has a nondestrructive version (VEX.128 encoded version), namely, VPCLMULQDQ, which is defined as follows.

Instruction format: VPCLMULQDQ xmm1, xmm2, xmm3/m128, imm8

Description: Carry-less multiplication of one quadword of xmm2 by one quadword of xmm3/m128, returning a double quadword (16 bytes) in register xmm1. The immediate is used for determining which quadwords of xmm2 and xmm3/m128 should be carry-less multiplied. The immediate byte is used exactly as in PCLMULQDQ. Functionally, this provides a non-destructive variant of PCLMULQDQ.

Opcode: VEX.NDS.128.66.0F3A 44 /r ib

The presence of VPCLMULQDQ is indicated by CPUID leaf 1 ECX[1].

The instruction operates on xmm states. The detection sequence must combine checking for CPUID.01H:ECX.PCLMULQDQ[bit 1] = 1, and the sequence for detection application support for GSSE.

Bits (255:128) of the destination ymm register are zeroed.

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 lookup table 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
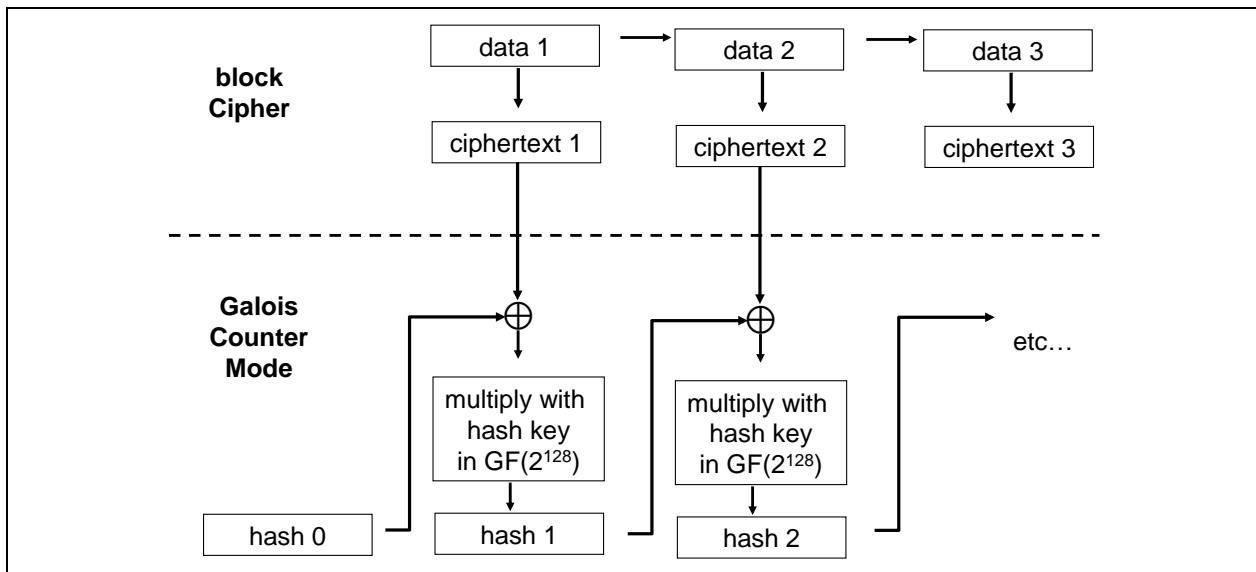
The multiplication in $GF(2^{128})$ involves carry-less multiplication of two 128-bit operands, to generate a 255-bit result (or, equivalently, 256-bit result, where the most significant bit equals 0), followed by and reduction modulo the irreducible polynomial g.

Current Software Implementation of GCM

Some of the current software implementations of GCM implement the Galois Filed multiplication directly. Such implementations are not very efficient when using only the current IA instructions set. One example is given in Figure 2, taken from OpenSolaris. Solaris has C function gcm_mul() that takes two 64-bit elements (*x_in and *y in Figure 2), and places their carryless product in a third 64-bit integer (*res). Figure 2 is the current C source code from http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/common/crypto/modes/gcm.c#46 (Copyright 2009 Sun Microsystems, Inc.).

Note: that after the publication of this paper, E. Käsper, P. Schwabe published a report on a much faster AES-GCM implementation in "Faster and Timing-Attack Resistant AES-GCM", Proceedings of CHES 2009, Lecture Notes in Computer Science, 5747:1-17.

Figure 2. The OpenSolaris "gfmul" C Function

```
/* copied from
http://src.opensolaris.org/source/xref/onnv/onnv-
gate/usr/src/common/crypto/modes/gcm.c#46
(Copyright 2009 Sun Microsystems, Inc.)
 */
#include <stdint.h>
#include <wmmintrin.h>
struct aes_block {
    uint64_t a;
    uint64_t b;
};
void
gfmul(__m128i x_in_m, __m128i y_m, uint64_t *res)
{
    uint64_t R = { 0xe100000000000000ULL };
    struct aes_block z = { 0, 0 };
    struct aes_block v;
    uint64_t x;
    int i, j;
    uint64_t *x_in=(uint64_t*)&x_in_m;
    uint64_t *y=(uint64_t*)&y_m;

    v.a=y[1];
    v.b=y[0];

    for (j = 1; j>=0; j--) {
        x = x_in[j];
        for (i = 0; i < 64; i++, x <<= 1) {
            if (x & 0x8000000000000000ULL) {
                z.a ^= v.a;
                z.b ^= v.b;
            }
            if (v.b & 1ULL) {
                v.b = (v.a << 63)|(v.b >> 1);
                v.a = (v.a >> 1) ^ R;
            } else {
                v.b = (v.a << 63)|(v.b >> 1);
                v.a = v.a >> 1;
            }
        }
    }
    res[0] = z.b;
    res[1] = z.a;
}
```

A more optimized software implementations of the GCM mode use a lookup table based algorithm [5], shown in Figure 3 (for AES-GCM). The pseudo-code of Figure 3 does not take into account bit and byte reflection peculiarities of GCM implementations. 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 involves 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 3. Lookup Table Based Implementation of AES-GCM

```
// Code snippet illustrating AES-GCM (AES-128)
// The AES round keys are assumed to be already expanded
//
         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 *)galoishash;
    copy128(temp, 0);
    for(j=0; j < 16; j++)
        xor128(temp, tables[j][bytes_gh[j]]);
      // tables[i][j] = j*hash_key*256*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 lookup table 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 $g = x^{128} + x^7 + x^2 + x + 1$.

Performing 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 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:

$$[A_1 : A_0] \bullet [B_1 : B_0] = [D_1 : F_1 \oplus E_1 \oplus D_0 : F_0 \oplus E_0 \oplus C_1 : C_0] \tag{5}$$

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:

$$[A_1 : A_0] \bullet [B_1 : B_0] = [C_1 : \ C_0 \oplus C_1 \oplus D_1 \oplus E_1 : D_1 \oplus C_0 \oplus D_0 \oplus E_0 : D_0] \tag{6}$$

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:

$$p(x) = c(x) \cdot x^t \bmod g(x) \tag{7}$$

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:

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

$$p(x) = p_{t-1}x^{t-1} + p_{t-2}x^{t-2} + ... + p_1 x + p_0 \,, \text{ and} \tag{8}$$

$$g(x) = g_t x^t + g_{t-1}x^{t-1} + ... + g_1 x + g_0$$

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:

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

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:

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

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,

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

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

$$p(x) = L^t(q(x) \cdot g^*(x)) \qquad (14)$$

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:

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

Let

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

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

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

and

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

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,

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

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:

$$A = X_3 >> 63$$
$$B = X_3 >> 62 \qquad (23)$$
$$C = X_3 >> 57$$

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

$$D = X_2 \oplus A \oplus B \oplus C \qquad (24)$$

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

$$[E_1 : E_0] = [X_3 : D] << 1$$
$$[F_1 : F_0] = [X_3 : D] << 2 \qquad (25)$$
$$[G_1 : G_0] = [X_3 : D] << 7$$

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:

$$[H_1 : H_0] = [X_3 \oplus E_1 \oplus F_1 \oplus G_1 : D \oplus E_0 \oplus F_0 \oplus G_0] \qquad (26)$$

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.

Definition: For a 128 bits quantity Q, Reflect (Q) is defined as the 128 bits quantity R whose i-th bit equal to the (127-i)-th bit of R, (for $0 \le i \le 127$).

This peculiarity applies to the two operands being multiplied in the finite field GF(2^{128}) (defined by the specification), and also to the order of bits in the reduction polynomial, which is [11100001:*<120 zeros>*:1] as opposed to [1:*<120 zeros>*:10000111].

Note that this peculiarity is not merely the difference between Little Endian and Big Endian notations.

We discuss here two approaches for handling the bit reflection peculiarity.

Bit Reflecting the Inputs

One approach for way handle the bit reflection peculiarity is to bit-reflect the input to the gfmul function. In GCM, one of the inputs is fixed for the whole process (it is the hash key), it can be reflected once generated. The other inputs need to be reflected as they are produced (by the AES-CTR module).

A simple way to implement bit reflection is to use lookup tables. Ideally we want to use an 8bit → 8bit lookup table, that matches a byte to its reflected value, and then swap the order of the bytes, as required. However, a more efficient way is to implement bit reflection using a 4bit → 4bit lookup table, which can fit into a single xmm register, and to apply the PSHUFB instruction in some sophisticated way.

To bit reflect a 128bit quantity (in an xmm), we first isolate the 4bit nibbles within each byte. Then we can reflect the nibble (8 at a time) by using PSHUFB with some pre-computed (constant) masks stored in two other xmm registers. We illustrate the method with the following example.

This is the contents of xmm1:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x56	0xff	0x13	0x00	0x09	0x43	0xb2	0x4a	0x17	0x8d	0xfc	0x3a	0xfd	0x34	0x43	0x11

xmm1 is copied to xmm2 and the nibbles are isolated in each xmm:

xmm1 holds the lower nibbles (in each byte):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x06	0x0f	0x03	0x00	0x09	0x03	0x02	0x0a	0x07	0x0d	0x0c	0x0a	0x0d	0x04	0x03	0x01

xmm2 holds the upper nibbles (in each byte):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x05	0x0f	0x01	0x00	0x00	0x04	0x0b	0x04	0x01	0x08	0x0f	0x03	0x0f	0x03	0x04	0x01

Now let xmm3 and xmm4 hold the reflected values of the 16 possible nibbles

xmm3 holds the low reflected nibbles:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x0f	0x07	0x0b	0x03	0x0d	0x05	0x09	0x01	0x0e	0x06	0x0a	0x02	0x0c	0x04	0x08	0x00

xmm4 holds the high reflected nibbles:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0xf0	0x70	0xb0	0x30	0xd0	0x50	0x90	0x10	0xe0	0x60	0xa0	0x20	0xc0	0x40	0x80	0x00

Now by using xmm1 and xmm2 as the mask and xmm3 with xmm4 as the operand, one will have the reflected nibbles in the right order in xmm3 and xmm4. By adding them and using another PSHUFB to swap the order of bytes, the reflected value of the whole xmm is obtained.

This method can be used to implement any 4bit → 4bit lookup table.

Figure 4. Code Sample – Reflecting Bits of a 128-bits Quantity

```
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

__m128i reflect_xmm(__m128i X)
{
    __m128i tmp1,tmp2;

    __m128i AND_MASK =
        _mm_set_epi32(0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f);
    __m128i LOWER_MASK =
        _mm_set_epi32(0x0f070b03, 0x0d050901, 0x0e060a02, 0x0c040800);
    __m128i HIGHER_MASK =
        _mm_set_epi32(0xf070b030, 0xd0509010, 0xe060a020, 0xc0408000);
    __m128i BSWAP_MASK =
        _mm_set_epi8(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);

    tmp2 = _mm_srli_epi16(X, 4);
    tmp1 = _mm_and_si128(X, AND_MASK);
    tmp2 = _mm_and_si128(tmp2, AND_MASK);
    tmp1 = _mm_shuffle_epi8(HIGHER_MASK ,tmp1);
    tmp2 = _mm_shuffle_epi8(LOWER_MASK ,tmp2);
    tmp1 = _mm_xor_si128(tmp1, tmp2);
    return _mm_shuffle_epi8(tmp1, BSWAP_MASK);
}
```

Avoiding Bit Reflecting

Here, we show an alternative that helps avoiding bit reflecting the inputs. We start with the following fundamental property of carry-less multiplication:

$$reflected \ (A) \bullet reflected \ (B) = reflected \ (A \bullet B) >> 1 \qquad (27)$$

Using this identity, 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

$$[X_3 X_2 X_1 X_0] = [X_3 X_2 X_1 X_0] << 1 \qquad (28)$$

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

$$A = X_0 << 63$$
$$B = X_0 << 62 \qquad (29)$$
$$C = X_0 << 57$$

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

$$D = X_1 \oplus A \oplus B \oplus C \qquad (30)$$

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

$$[E_1 : E_0] = [D : X_0] >> 1$$
$$[F_1 : F_0] = [D : X_0] >> 2 \qquad (31)$$
$$[G_1 : G_0] = [D : X_0] >> 7$$

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:

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

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

The advantage of this method, over reflecting the inputs (as shows above) is that it does not require the data to be reflected. This allows for aggregating several reductions together, for a more efficient over all implementation, as shown below.

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:

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

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:

$$p(x) = (c(x) \cdot x^t + d(x)) \bmod g(x) =$$
$$c(x) \cdot x^t \bmod g(x) + d(x) \bmod g(x) = \qquad (34)$$
$$(c(x) \cdot (x^t \bmod g(x)) + d(x)) \bmod g(x)$$

The quantity x^t mod $g(x)$ 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 mod $g(x)$.

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 mod $g(x)$ 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 mod $g(x)$ can be potentially small, equals 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 $[H_2{:}H_1{:}H_0] = [C_1{:}C_0] \bullet$ 0xe100000000000000   (64 bit)

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

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

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

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 $[H_2{:}H_1{:}H_0] = [C_1{:}C_0] \bullet$ 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 $A = X_1 \,\&\, 1(<< 63)$, $B = X_1 << 1$, $C = (X_1 \,\&\, 0x7f00000000000000) >> 1$, $D = (X_1 \,\&\, 0x7f00000000000000) >> 6$, $E = X_1 \,\&\, 0x7f00000000000000$.

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

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

$^{(*)}$ **Note**: 0x1c200000000000000 is a 65-bits quantity, and therefore four multiplications are required for computing $[C_1:C_0]$ • 0x1c200000000000000

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 • 0xe100000000000000   (64 bit)

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

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

Step 4: Compute $A = X_1$ & (1 <<63), $B = X_1$ << 1, $C_1 = (X_1$ & 0x7fffffffffffffff) >> 1, $C_0 = X_1$<<63, $D_1 = (X_1$ & 0x7fffffffffffffff) >> 6, $D_0 = X_1$<<58, $E = X_1$ & 0x7fffffffffffffff,

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 • 0x1c200000000000000   (65 bit)

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

Step 3: Compute $A = X_1$ & (1 << 63), $B = X_1$ << 1, $C_1 = (X_1$ & 0x7fffffffffffffff) >> 1, $C_0 = X_1$<<63, $D_1 = (X_1$ & 0x7fffffffffffffff) >> 6, $D_0 = X_1$<<58, $E = X_1$ & 0x7fffffffffffffff,

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

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

Aggregated Reduction Method

This method was proposed by Krzysztof Jankowski, Pierre Laurent (see [17]). It is a way to delay the reduction step and apply the reducing to the aggregated result only once every few multiplications.

The standard Ghash formula is the following:

$$Y_i = [(X_i + Y_{i-1}) \bullet H] \mod P \qquad (P = x^{128} + x^7 + x^2 + x + 1)$$

Instead, one can apply the following recursion:

$$Y_i = [(X_i + Y_{i-1}) \bullet H] \mod P$$

$$= [(X_i \bullet H) + (Y_{i-1} \bullet H)] \mod P$$

$$= [(X_i \bullet H) + (X_{i-1} + Y_{i-2}) \bullet H^2] \mod P$$

$$= [(X_i \bullet H) + (X_{i-1} \bullet H^2) + (X_{i-2} + Y_{i-3}) \bullet H^3] \mod P$$

$$= [(X_i \bullet H) + (X_{i-1} \bullet H^2) + (X_{i-2} \bullet H^3) + (X_{i-3} + Y_{i-4}) \bullet H^4] \mod P$$

This can be further expanded to any depth. The only overhead in this computation is the need to calculate the powers of H mod P in advance, but the gain is that reduction is required only once in a few blocks. As can be appreciated now, the use of the identity (27) is advantageous because the data (from the AES-CTR) does not need to be manipulated, as would be the case if we choose the bit reflect method.

A good choice, which we adopt here, is to compute one reduction every 4 blocks.

Code Examples: Ghash Computation

Figures 5-8 are assembly and C code examples for implementing the Galois Field multiplication portion of GCM, using some combinations of the algorithms described in the section "Efficient Algorithms for Computing GCM".

These codes implement the "gfmul" function which is, in the subsequent chapter, integrated into the full AES-GCM code.

Figures 5-6 implement the same algorithms – one example in C and the other example in assembly (AT&T syntax). The assembly code example can be compiled from an assembly file, using the gcc compiler, with –maes and –msse4 flags.

The code in Figure 7 implements a different selection of algorithms.

The code in Figure 8 implements the Aggregated Reduction Method

Figure 5. Code Sample - Performing Ghash Using Algorithms 1 and 5 (C)

```
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>
void gfmul (__m128i a, __m128i b, __m128i *res){
    __m128i tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6, tmp7, tmp8, tmp9;

    tmp3 = _mm_clmulepi64_si128(a, b, 0x00);
    tmp4 = _mm_clmulepi64_si128(a, b, 0x10);
    tmp5 = _mm_clmulepi64_si128(a, b, 0x01);
    tmp6 = _mm_clmulepi64_si128(a, b, 0x11);

    tmp4 = _mm_xor_si128(tmp4, tmp5);
    tmp5 = _mm_slli_si128(tmp4, 8);
    tmp4 = _mm_srli_si128(tmp4, 8);
    tmp3 = _mm_xor_si128(tmp3, tmp5);
    tmp6 = _mm_xor_si128(tmp6, tmp4);

    tmp7 = _mm_srli_epi32(tmp3, 31);
    tmp8 = _mm_srli_epi32(tmp6, 31);
    tmp3 = _mm_slli_epi32(tmp3, 1);
    tmp6 = _mm_slli_epi32(tmp6, 1);

    tmp9 = _mm_srli_si128(tmp7, 12);
    tmp8 = _mm_slli_si128(tmp8, 4);
    tmp7 = _mm_slli_si128(tmp7, 4);
    tmp3 = _mm_or_si128(tmp3, tmp7);
    tmp6 = _mm_or_si128(tmp6, tmp8);
    tmp6 = _mm_or_si128(tmp6, tmp9);

    tmp7 = _mm_slli_epi32(tmp3, 31);
    tmp8 = _mm_slli_epi32(tmp3, 30);
    tmp9 = _mm_slli_epi32(tmp3, 25);

    tmp7 = _mm_xor_si128(tmp7, tmp8);
    tmp7 = _mm_xor_si128(tmp7, tmp9);
    tmp8 = _mm_srli_si128(tmp7, 4);
    tmp7 = _mm_slli_si128(tmp7, 12);
    tmp3 = _mm_xor_si128(tmp3, tmp7);
```

```
    tmp2 = _mm_srli_epi32(tmp3, 1);
    tmp4 = _mm_srli_epi32(tmp3, 2);
    tmp5 = _mm_srli_epi32(tmp3, 7);
    tmp2 = _mm_xor_si128(tmp2, tmp4);
    tmp2 = _mm_xor_si128(tmp2, tmp5);
    tmp2 = _mm_xor_si128(tmp2, tmp8);
    tmp3 = _mm_xor_si128(tmp3, tmp2);
    tmp6 = _mm_xor_si128(tmp6, tmp3);

    *res = tmp6;
}
```

Figure 6. Code Sample - Performing Ghash Using Algorithms 1 and 5 (Assembly)

```
.globl    gfmul
gfmul:
        #xmm0 holds operand a (128 bits)
        #xmm1 holds operand b (128 bits)
        #rdi  holds the pointer to output (128 bits)

 movdqa     %xmm0, %xmm3
 pclmulqdq  $0, %xmm1, %xmm3    # xmm3 holds a0*b0
 movdqa     %xmm0, %xmm4
 pclmulqdq  $16, %xmm1, %xmm4    #xmm4 holds a0*b1
 movdqa     %xmm0, %xmm5
 pclmulqdq  $1, %xmm1, %xmm5    # xmm5 holds a1*b0
 movdqa     %xmm0, %xmm6
 pclmulqdq  $17, %xmm1, %xmm6    # xmm6 holds a1*b1
 pxor       %xmm5, %xmm4        # xmm4 holds a0*b1 + a1*b0
 movdqa     %xmm4, %xmm5
 psrldq     $8, %xmm4
 pslldq     $8, %xmm5
 pxor       %xmm5, %xmm3
 pxor       %xmm4, %xmm6        # <xmm6:xmm3> holds the result of
 # the carry-less multiplication of xmm0 by xmm1

 # shift the result by one bit position to the left cope for the fact
 # that bits are reversed
 movdqa     %xmm3, %xmm7
 movdqa     %xmm6, %xmm8
 pslld      $1, %xmm3
 pslld      $1, %xmm6
 psrld      $31, %xmm7
 psrld      $31, %xmm8
 movdqa     %xmm7, %xmm9
 pslldq     $4, %xmm8
 pslldq     $4, %xmm7
 psrldq     $12, %xmm9
 por        %xmm7, %xmm3
 por        %xmm8, %xmm6
 por        %xmm9, %xmm6

 #first phase of the reduction
 movdqa     %xmm3, %xmm7
 movdqa     %xmm3, %xmm8
 movdqa     %xmm3, %xmm9
 pslld      $31, %xmm7              # packed right shifting << 31
 pslld      $30, %xmm8              # packed right shifting shift << 30
 pslld      $25, %xmm9              # packed right shifting shift << 25
 pxor       %xmm8, %xmm7              # xor the shifted versions
 pxor       %xmm9, %xmm7

 movdqa     %xmm7, %xmm8
```

```
pslldq    $12, %xmm7
psrldq    $4, %xmm8
pxor      %xmm7, %xmm3          # first phase of the reduction complete
movdqa    %xmm3,%xmm2           # second phase of the reduction
movdqa    %xmm3,%xmm4
movdqa    %xmm3,%xmm5
psrld     $1, %xmm2            # packed left shifting >> 1
psrld     $2, %xmm4            # packed left shifting >> 2
psrld     $7, %xmm5            # packed left shifting >> 7

pxor      %xmm4, %xmm2          # xor the shifted versions
pxor      %xmm5, %xmm2
pxor      %xmm8, %xmm2
pxor      %xmm2, %xmm3
pxor      %xmm3, %xmm6          # the result is in xmm6
movdqu    %xmm6, (%rdi)         # store the result
ret
```

Figure 7. Code Sample - Performing Ghash Using Algorithms 2 and 4 with Reflected Input and Output

```
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

void gfmul (__m128i a, __m128i b, __m128i *res){
    __m128i tmp0, tmp1, tmp2, tmp3, tmp4, tmp5, tmp6,
            tmp7, tmp8, tmp9, tmp10, tmp11, tmp12;
    __m128i XMMMASK = _mm_setr_epi32(0xffffffff, 0x0, 0x0, 0x0);

    tmp3 = _mm_clmulepi64_si128(a, b, 0x00);
    tmp6 = _mm_clmulepi64_si128(a, b, 0x11);

    tmp4 = _mm_shuffle_epi32(a,78);
    tmp5 = _mm_shuffle_epi32(b,78);
    tmp4 = _mm_xor_si128(tmp4, a);
    tmp5 = _mm_xor_si128(tmp5, b);

    tmp4 = _mm_clmulepi64_si128(tmp4, tmp5, 0x00);
    tmp4 = _mm_xor_si128(tmp4, tmp3);
    tmp4 = _mm_xor_si128(tmp4, tmp6);

    tmp5 = _mm_slli_si128(tmp4, 8);
    tmp4 = _mm_srli_si128(tmp4, 8);
    tmp3 = _mm_xor_si128(tmp3, tmp5);
    tmp6 = _mm_xor_si128(tmp6, tmp4);

    tmp7 = _mm_srli_epi32(tmp6, 31);
    tmp8 = _mm_srli_epi32(tmp6, 30);
    tmp9 = _mm_srli_epi32(tmp6, 25);

    tmp7 = _mm_xor_si128(tmp7, tmp8);
    tmp7 = _mm_xor_si128(tmp7, tmp9);

    tmp8 = _mm_shuffle_epi32(tmp7, 147);

    tmp7 = _mm_and_si128(XMMMASK, tmp8);
    tmp8 = _mm_andnot_si128(XMMMASK, tmp8);
    tmp3 = _mm_xor_si128(tmp3, tmp8);
    tmp6 = _mm_xor_si128(tmp6, tmp7);
```

```
    tmp10 = _mm_slli_epi32(tmp6, 1);
    tmp3 = _mm_xor_si128(tmp3, tmp10);
    tmp11 = _mm_slli_epi32(tmp6, 2);
    tmp3 = _mm_xor_si128(tmp3, tmp11);
    tmp12 = _mm_slli_epi32(tmp6, 7);
    tmp3 = _mm_xor_si128(tmp3, tmp12);

    *res = _mm_xor_si128(tmp3, tmp6);
 }
```

Figure 8. Code Sample - Performing Ghash Using an Aggregated Reduction Method

```
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

void reduce4 (__m128i H1, __m128i H2, __m128i H3, __m128i H4,
              __m128i X1, __m128i X2, __m128i X3, __m128i X4, __m128i *res)
    {
    /*algorithm by Krzysztof Jankowski,  Pierre Laurent  - Intel*/
    __m128i H1_X1_lo, H1_X1_hi,
            H2_X2_lo, H2_X2_hi,
            H3_X3_lo, H3_X3_hi,
            H4_X4_lo, H4_X4_hi,
            lo, hi;
    __m128i tmp0, tmp1, tmp2, tmp3;
    __m128i tmp4, tmp5, tmp6, tmp7;
    __m128i tmp8, tmp9;


    H1_X1_lo = _mm_clmulepi64_si128(H1, X1, 0x00);
    H2_X2_lo = _mm_clmulepi64_si128(H2, X2, 0x00);
    H3_X3_lo = _mm_clmulepi64_si128(H3, X3, 0x00);
    H4_X4_lo = _mm_clmulepi64_si128(H4, X4, 0x00);

    lo = _mm_xor_si128(H1_X1_lo, H2_X2_lo);
    lo = _mm_xor_si128(lo, H3_X3_lo);
    lo = _mm_xor_si128(lo, H4_X4_lo);

    H1_X1_hi = _mm_clmulepi64_si128(H1, X1, 0x11);
    H2_X2_hi = _mm_clmulepi64_si128(H2, X2, 0x11);
    H3_X3_hi = _mm_clmulepi64_si128(H3, X3, 0x11);
    H4_X4_hi = _mm_clmulepi64_si128(H4, X4, 0x11);

    hi = _mm_xor_si128(H1_X1_hi, H2_X2_hi);
    hi = _mm_xor_si128(hi, H3_X3_hi);
    hi = _mm_xor_si128(hi, H4_X4_hi);

    tmp0 = _mm_shuffle_epi32(H1, 78);
    tmp4 = _mm_shuffle_epi32(X1, 78);
    tmp0 = _mm_xor_si128(tmp0, H1);
    tmp4 = _mm_xor_si128(tmp4, X1);
    tmp1 = _mm_shuffle_epi32(H2, 78);
    tmp5 = _mm_shuffle_epi32(X2, 78);
    tmp1 = _mm_xor_si128(tmp1, H2);
    tmp5 = _mm_xor_si128(tmp5, X2);
    tmp2 = _mm_shuffle_epi32(H3, 78);
    tmp6 = _mm_shuffle_epi32(X3, 78);
    tmp2 = _mm_xor_si128(tmp2, H3);
    tmp6 = _mm_xor_si128(tmp6, X3);
    tmp3 = _mm_shuffle_epi32(H4, 78);
```

```
    tmp7 = _mm_shuffle_epi32(X4, 78);
    tmp3 = _mm_xor_si128(tmp3, H4);
    tmp7 = _mm_xor_si128(tmp7, X4);

    tmp0 = _mm_clmulepi64_si128(tmp0, tmp4, 0x00);
    tmp1 = _mm_clmulepi64_si128(tmp1, tmp5, 0x00);
    tmp2 = _mm_clmulepi64_si128(tmp2, tmp6, 0x00);
    tmp3 = _mm_clmulepi64_si128(tmp3, tmp7, 0x00);

    tmp0 = _mm_xor_si128(tmp0,  lo);
    tmp0 = _mm_xor_si128(tmp0,  hi);
    tmp0 = _mm_xor_si128(tmp1,  tmp0);
    tmp0 = _mm_xor_si128(tmp2,  tmp0);
    tmp0 = _mm_xor_si128(tmp3,  tmp0);

    tmp4 = _mm_slli_si128(tmp0, 8);
    tmp0 = _mm_srli_si128(tmp0, 8);

    lo = _mm_xor_si128(tmp4,  lo);
    hi = _mm_xor_si128(tmp0,  hi);

    tmp3 = lo;
    tmp6 = hi;

    tmp7 = _mm_srli_epi32(tmp3, 31);
    tmp8 = _mm_srli_epi32(tmp6, 31);
    tmp3 = _mm_slli_epi32(tmp3, 1);
    tmp6 = _mm_slli_epi32(tmp6, 1);

    tmp9 = _mm_srli_si128(tmp7, 12);
    tmp8 = _mm_slli_si128(tmp8, 4);
    tmp7 = _mm_slli_si128(tmp7, 4);
    tmp3 = _mm_or_si128(tmp3, tmp7);
    tmp6 = _mm_or_si128(tmp6, tmp8);
    tmp6 = _mm_or_si128(tmp6, tmp9);

    tmp7 = _mm_slli_epi32(tmp3, 31);
    tmp8 = _mm_slli_epi32(tmp3, 30);
    tmp9 = _mm_slli_epi32(tmp3, 25);

    tmp7 = _mm_xor_si128(tmp7, tmp8);
    tmp7 = _mm_xor_si128(tmp7, tmp9);
    tmp8 = _mm_srli_si128(tmp7, 4);
    tmp7 = _mm_slli_si128(tmp7, 12);
    tmp3 = _mm_xor_si128(tmp3, tmp7);

    tmp2 = _mm_srli_epi32(tmp3, 1);
    tmp4 = _mm_srli_epi32(tmp3, 2);
    tmp5 = _mm_srli_epi32(tmp3, 7);
    tmp2 = _mm_xor_si128(tmp2, tmp4);
    tmp2 = _mm_xor_si128(tmp2, tmp5);
    tmp2 = _mm_xor_si128(tmp2, tmp8);
    tmp3 = _mm_xor_si128(tmp3, tmp2);
    tmp6 = _mm_xor_si128(tmp6, tmp3);

    *res = tmp6;
    }
```

Code Examples: AES128-GCM

This chapter provides several code examples for implementing AES-GCM using PCLMULQDQ and the AES instructions (see [16] for details). We demonstrate here AES128 in GCM mode. The extension to AES192 and AES256 in GCM mode is straightforward (implementation of AES192/AES256 CTR is given in [16]). One of the AES-CTR implementations (Figure 12) uses the technique of processing several (four in this case) blocks in parallel. Reference [17] described the motivation and benefits of such parallelization techniques for AES in any parallel modes of operation.

How to Use the Code Examples

In the following, we provide three possible options for using the provided code examples.

For AES-GCM, using the simplest method (one blocks at a time): save Figure 5 as gfmul.c or, alternatively, Figure 6 as gfmul.s. Save Figure 9 as aes_gcm.c.

For AES-GCM, using the reflected variation, use Figure 7 for gfmul.c, and for aes_gcm.c. (the provided decypt function does not support that gfmul function, therefore the decrpyt will produce incorrect results in that case).

For AES-GCM, using the parallelized method, save Figure 5 and Figure 8 together as gfmul.c, save

Figure 12. as aes_gcm.c. Alternatively save Figure 15 as aes_gcm.s

For all methods, save

Figure 10 as decrypt.c ,

Figure 13 as key_schedule.c and Figure 14 as gcm_main.c.

For Intel Compiler (icc) users: the files can be compiled using the compilation line:

```
icc aes_gcm.c key_schedule.c gcm_main.c gfmul.c decrypt.c -o AES128_GCM –
DTEST[1-6]
```

For gcc users: the files can be compiled using the compilation line

```
gcc –maes –msse4 aes_gcm.c key_schedule.c gcm_main.c gfmul.c decrypt.c -o
AES128_GCM -DTEST[1-6]
```

The compilation flag -DTEST[1-6] selects one of the 6 AES GCM tests that are including in the main funrcion. Running the resulting executable (AES128_GCM) would give the respective printout, as described in the section "Code Outputs" below (e.g., -DTEST1 selects test number 1). The test vectors were taken from the specification document: http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

The code can be run even without a processor based on Intel microarchitecture codename Westmere, using Intel® Software Development Emulator (Intel® SDE), which can be downloaded from http://www.intel.com/software/sde.

AES-GCM Code Examples

Figure 9. AES-GCM – Encrypt With Single Block Ghash at a Time

```c
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

extern void gfmul(__m128i a,__m128i b,__m128i* c); //must be implemented elsewhere

void AES_GCM_encrypt(const unsigned char *in,
                     unsigned char *out,
                     const unsigned char* addt,
                     const unsigned char* ivec,
                     unsigned char *tag,
                     int nbytes,
                     int abytes,
                     int ibytes,
                     const unsigned char* key,
                     int nr)
    {
    int i, j ,k;
    __m128i hlp1, hlp2, hlp3, hlp4;
    __m128i tmp1, tmp2, tmp3, tmp4;
    __m128i H, Y, T;
    __m128i *KEY = (__m128i*)key;
    __m128i ctr1, ctr2, ctr3, ctr4;
    __m128i last_block = _mm_setzero_si128();
    __m128i ONE = _mm_set_epi32(0, 1, 0, 0);
    __m128i FOUR = _mm_set_epi32(0, 4, 0, 0);
    __m128i BSWAP_EPI64 = _mm_set_epi8(8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4,
5, 6, 7);
    __m128i BSWAP_MASK = _mm_set_epi8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15);
    __m128i X = _mm_setzero_si128();

    if(ibytes == 96/8){
        Y = _mm_loadu_si128((__m128i*)ivec);
        Y = _mm_insert_epi32(Y, 0x1000000, 3);
        /*(Compute E[ZERO, KS] and E[Y0, KS] together*/
        tmp1 = _mm_xor_si128(X, KEY[0]);
        tmp2 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr-1; j+=2) {
                tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
                tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);

                tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
                tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
                };
        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);

        H = _mm_aesenclast_si128(tmp1, KEY[nr]);
        T = _mm_aesenclast_si128(tmp2, KEY[nr]);

        H = _mm_shuffle_epi8(H, BSWAP_MASK);
        }
    else{
        tmp1 = _mm_xor_si128(X, KEY[0]);
        for(j=1; j <nr; j++)
```

```
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
        H = _mm_aesenclast_si128(tmp1, KEY[nr]);
        H = _mm_shuffle_epi8(H, BSWAP_MASK);
        Y = _mm_xor_si128(Y, Y);

        for(i=0; i < ibytes/16; i++){
            tmp1 = _mm_loadu_si128(&((__m128i*)ivec)[i]);
            tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
            Y = _mm_xor_si128(Y, tmp1);
            gfmul(Y, H, &Y);
            }
        if(ibytes%16){
            for(j=0; j < ibytes%16; j++)
                ((unsigned char*)&last_block)[j] = ivec[i*16+j];
            tmp1 = last_block;
            tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
            Y = _mm_xor_si128(Y, tmp1);
            gfmul(Y, H, &Y);
            }
        tmp1 = _mm_insert_epi64(tmp1, ibytes*8, 0);
        tmp1 = _mm_insert_epi64(tmp1, 0, 1);

        Y = _mm_xor_si128(Y, tmp1);
        gfmul(Y, H, &Y);
        Y = _mm_shuffle_epi8(Y, BSWAP_MASK);
        /*Compute E(K, Y0)*/
        tmp1 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr; j++)
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
        T = _mm_aesenclast_si128(tmp1, KEY[nr]);
    }

    for(i=0; i<abytes/16; i++){
        tmp1 = _mm_loadu_si128(&((__m128i*)addt)[i]);
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        }
    if(abytes%16){
        last_block = _mm_setzero_si128();
        for(j=0; j<abytes%16; j++)
            ((unsigned char*)&last_block)[j] = addt[i*16+j];
        tmp1 = last_block;
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        }

    ctr1 = _mm_shuffle_epi8(Y, BSWAP_EPI64);
    ctr1 = _mm_add_epi32(ctr1, ONE);
    ctr2 = _mm_add_epi32(ctr1, ONE);
    ctr3 = _mm_add_epi32(ctr2, ONE);
    ctr4 = _mm_add_epi32(ctr3, ONE);

    for(i=0; i < nbytes/16/4; i++){
        tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
        tmp2 = _mm_shuffle_epi8(ctr2, BSWAP_EPI64);
        tmp3 = _mm_shuffle_epi8(ctr3, BSWAP_EPI64);
        tmp4 = _mm_shuffle_epi8(ctr4, BSWAP_EPI64);
```

```
        ctr1 = _mm_add_epi32(ctr1, FOUR);
        ctr2 = _mm_add_epi32(ctr2, FOUR);
        ctr3 = _mm_add_epi32(ctr3, FOUR);
        ctr4 = _mm_add_epi32(ctr4, FOUR);

        tmp1 =_mm_xor_si128(tmp1, KEY[0]);
        tmp2 =_mm_xor_si128(tmp2, KEY[0]);
        tmp3 =_mm_xor_si128(tmp3, KEY[0]);
        tmp4 =_mm_xor_si128(tmp4, KEY[0]);

        for(j=1; j < nr-1; j+=2){
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j+1]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j+1]);
            }

        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);
        tmp3 = _mm_aesenc_si128(tmp3, KEY[nr-1]);
        tmp4 = _mm_aesenc_si128(tmp4, KEY[nr-1]);

        tmp1 =_mm_aesenclast_si128(tmp1, KEY[nr]);
        tmp2 =_mm_aesenclast_si128(tmp2, KEY[nr]);
        tmp3 =_mm_aesenclast_si128(tmp3, KEY[nr]);
        tmp4 =_mm_aesenclast_si128(tmp4, KEY[nr]);

        tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[i*4+0]));
        tmp2 = _mm_xor_si128(tmp2, _mm_loadu_si128(&((__m128i*)in)[i*4+1]));
        tmp3 = _mm_xor_si128(tmp3, _mm_loadu_si128(&((__m128i*)in)[i*4+2]));
        tmp4 = _mm_xor_si128(tmp4, _mm_loadu_si128(&((__m128i*)in)[i*4+3]));

        _mm_storeu_si128(&((__m128i*)out)[i*4+0], tmp1);
        _mm_storeu_si128(&((__m128i*)out)[i*4+1], tmp2);
        _mm_storeu_si128(&((__m128i*)out)[i*4+2], tmp3);
        _mm_storeu_si128(&((__m128i*)out)[i*4+3], tmp4);

        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        tmp2 = _mm_shuffle_epi8(tmp2, BSWAP_MASK);
        tmp3 = _mm_shuffle_epi8(tmp3, BSWAP_MASK);
        tmp4 = _mm_shuffle_epi8(tmp4, BSWAP_MASK);

        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        X = _mm_xor_si128(X, tmp2);
        gfmul(X, H, &X);
        X = _mm_xor_si128(X, tmp3);
        gfmul(X, H, &X);
        X = _mm_xor_si128(X, tmp4);
        gfmul(X, H, &X);
        }
    for(k = i*4; k < nbytes/16; k++){
        tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
        ctr1 = _mm_add_epi32(ctr1, ONE);
        tmp1 = _mm_xor_si128(tmp1, KEY[0]);
        for(j=1; j<nr-1; j+=2){
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
```

```
              }
         tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
         tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
         tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
         _mm_storeu_si128(&((__m128i*)out)[k], tmp1);
         tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
         X = _mm_xor_si128(X, tmp1);
         gfmul(X, H, &X);
         }
//If one partial block remains
    if(nbytes%16){
         tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
         tmp1 = _mm_xor_si128(tmp1, KEY[0]);
         for(j=1; j<nr-1; j+=2){
              tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
              tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
              }
         tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
         tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
         tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
         last_block = tmp1;
         for(j=0; j < nbytes%16; j++)
              out[k*16+j]=((unsigned char*)&last_block)[j];
         for(j; j<16; j++)
              ((unsigned char*)&last_block)[j]=0;
         tmp1 = last_block;
         tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
         X = _mm_xor_si128(X, tmp1);
         gfmul(X, H, &X);
         }
    tmp1 = _mm_insert_epi64(tmp1, nbytes*8, 0);
    tmp1 = _mm_insert_epi64(tmp1, abytes*8, 1);

    X = _mm_xor_si128(X, tmp1);
    gfmul(X, H, &X);
    X = _mm_shuffle_epi8(X, BSWAP_MASK);
    T = _mm_xor_si128(X, T);
    _mm_storeu_si128((__m128i*)tag, T);
}
```

Figure 10. AES-GCM – Decrypt With Single Block Ghash at a Time

```c
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

extern void gfmul(__m128i a,__m128i b,__m128i* c); //must be implemented elsewhere

int AES_GCM_decrypt (const unsigned char *in,
                     unsigned char *out,
                     const unsigned char* addt,
                     const unsigned char* ivec,
                     unsigned char *tag,
                     int nbytes,
                     int abytes,
                     int ibytes,
                     const unsigned char* key,
                     int nr)
    {
    int i, j ,k;
    __m128i hlp1, hlp2, hlp3, hlp4;
    __m128i tmp1, tmp2, tmp3, tmp4;
    __m128i H, Y, T;
    __m128i *KEY = (__m128i*)key;
    __m128i ctr1, ctr2, ctr3, ctr4;
    __m128i last_block = _mm_setzero_si128();
    __m128i ONE = _mm_set_epi32(0, 1, 0, 0);
    __m128i FOUR = _mm_set_epi32(0, 4, 0, 0);
    __m128i BSWAP_EPI64 = _mm_set_epi8(8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7);
    __m128i BSWAP_MASK = _mm_set_epi8(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
    __m128i X = _mm_setzero_si128();

    if(ibytes == 96/8){
        Y = _mm_loadu_si128((__m128i*)ivec);
        Y = _mm_insert_epi32(Y, 0x1000000, 3);
        /*(Compute E[ZERO, KS] and E[Y0, KS] together*/
        tmp1 = _mm_xor_si128(X, KEY[0]);
        tmp2 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr-1; j+=2) {
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            };
        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);

        H = _mm_aesenclast_si128(tmp1, KEY[nr]);
        T = _mm_aesenclast_si128(tmp2, KEY[nr]);

        H = _mm_shuffle_epi8(H, BSWAP_MASK);
        }
    else{
        tmp1 = _mm_xor_si128(X, KEY[0]);
        for(j=1; j <nr; j++)
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
        H = _mm_aesenclast_si128(tmp1, KEY[nr]);
        H = _mm_shuffle_epi8(H, BSWAP_MASK);
        Y = _mm_xor_si128(Y, Y);

        for(i=0; i < ibytes/16; i++){
            tmp1 = _mm_loadu_si128(&((__m128i*)ivec)[i]);
```

```
                tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
                Y = _mm_xor_si128(Y, tmp1);
                gfmul(Y, H, &Y);
                }
        if(ibytes%16){
                for(j=0; j < ibytes%16; j++)
                    ((unsigned char*)&last_block)[j] = ivec[i*16+j];
                tmp1 = last_block;
                tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
                Y = _mm_xor_si128(Y, tmp1);
                gfmul(Y, H, &Y);
                }
        tmp1 = _mm_insert_epi64(tmp1, ibytes*8, 0);
        tmp1 = _mm_insert_epi64(tmp1, 0, 1);

        Y = _mm_xor_si128(Y, tmp1);
        gfmul(Y, H, &Y);
        Y = _mm_shuffle_epi8(Y, BSWAP_MASK);
        /*Compute E(K, Y0)*/
        tmp1 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr; j++)
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
        T = _mm_aesenclast_si128(tmp1, KEY[nr]);
    }

    for(i=0; i<abytes/16; i++){
        tmp1 = _mm_loadu_si128(&((__m128i*)addt)[i]);
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        }
    if(abytes%16){
        last_block = _mm_setzero_si128();
        for(j=0;j<abytes%16;j++)
            ((unsigned char*)&last_block)[j] = addt[i*16+j];
        tmp1 = last_block;
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X =_mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        }

    for(i=0; i<nbytes/16; i++){
        tmp1 = _mm_loadu_si128(&((__m128i*)in)[i]);
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        }
    if(nbytes%16){
        last_block = _mm_setzero_si128();
        for(j=0; j<nbytes%16; j++)
            ((unsigned char*)&last_block)[j] = in[i*16+j];
        tmp1 = last_block;
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        }

    tmp1 =_mm_insert_epi64(tmp1, nbytes*8, 0);
    tmp1 =_mm_insert_epi64(tmp1, abytes*8, 1);
```

```
    X = _mm_xor_si128(X, tmp1);
    gfmul(X, H, &X);
    X = _mm_shuffle_epi8(X, BSWAP_MASK);
    T = _mm_xor_si128(X, T);

    if(0xffff!=_mm_movemask_epi8(_mm_cmpeq_epi8(T, _mm_loadu_si128((__m128i*)tag))))
        return 0;    //in case the authentication failed

    ctr1 = _mm_shuffle_epi8(Y, BSWAP_EPI64);
    ctr1 = _mm_add_epi32(ctr1, ONE);
    ctr2 = _mm_add_epi32(ctr1, ONE);
    ctr3 = _mm_add_epi32(ctr2, ONE);
    ctr4 = _mm_add_epi32(ctr3, ONE);

    for(i=0; i < nbytes/16/4; i++){
        tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
        tmp2 = _mm_shuffle_epi8(ctr2, BSWAP_EPI64);
        tmp3 = _mm_shuffle_epi8(ctr3, BSWAP_EPI64);
        tmp4 = _mm_shuffle_epi8(ctr4, BSWAP_EPI64);

        ctr1 = _mm_add_epi32(ctr1, FOUR);
        ctr2 = _mm_add_epi32(ctr2, FOUR);
        ctr3 = _mm_add_epi32(ctr3, FOUR);
        ctr4 = _mm_add_epi32(ctr4, FOUR);

        tmp1 =_mm_xor_si128(tmp1, KEY[0]);
        tmp2 =_mm_xor_si128(tmp2, KEY[0]);
        tmp3 =_mm_xor_si128(tmp3, KEY[0]);
        tmp4 =_mm_xor_si128(tmp4, KEY[0]);

        for(j=1; j < nr-1; j+=2){
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j+1]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j+1]);
            }

        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);
        tmp3 = _mm_aesenc_si128(tmp3, KEY[nr-1]);
        tmp4 = _mm_aesenc_si128(tmp4, KEY[nr-1]);

        tmp1 =_mm_aesenclast_si128(tmp1, KEY[nr]);
        tmp2 =_mm_aesenclast_si128(tmp2, KEY[nr]);
        tmp3 =_mm_aesenclast_si128(tmp3, KEY[nr]);
        tmp4 =_mm_aesenclast_si128(tmp4, KEY[nr]);

        tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[i*4+0]));
        tmp2 = _mm_xor_si128(tmp2, _mm_loadu_si128(&((__m128i*)in)[i*4+1]));
        tmp3 = _mm_xor_si128(tmp3, _mm_loadu_si128(&((__m128i*)in)[i*4+2]));
        tmp4 = _mm_xor_si128(tmp4, _mm_loadu_si128(&((__m128i*)in)[i*4+3]));

        _mm_storeu_si128(&((__m128i*)out)[i*4+0], tmp1);
        _mm_storeu_si128(&((__m128i*)out)[i*4+1], tmp2);
        _mm_storeu_si128(&((__m128i*)out)[i*4+2], tmp3);
        _mm_storeu_si128(&((__m128i*)out)[i*4+3], tmp4);

        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
```

```
          tmp2 = _mm_shuffle_epi8(tmp2, BSWAP_MASK);
          tmp3 = _mm_shuffle_epi8(tmp3, BSWAP_MASK);
          tmp4 = _mm_shuffle_epi8(tmp4, BSWAP_MASK);
          }
    for(k = i*4; k < nbytes/16; k++){
          tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
          ctr1 = _mm_add_epi32(ctr1, ONE);
          tmp1 = _mm_xor_si128(tmp1, KEY[0]);
          for(j=1; j<nr-1; j+=2){
              tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
              tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
              }
          tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
          tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
          tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
          _mm_storeu_si128(&((__m128i*)out)[k], tmp1);
          }
//If one partial block remains
    if(nbytes%16){
          tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
          tmp1 = _mm_xor_si128(tmp1, KEY[0]);
          for(j=1; j<nr-1; j+=2){
              tmp1 =_mm_aesenc_si128(tmp1, KEY[j]);
              tmp1 =_mm_aesenc_si128(tmp1, KEY[j+1]);
              }
          tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
          tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
          tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
          last_block = tmp1;
          for(j=0; j<nbytes%16; j++)
              out[k*16+j]=((unsigned char*)&last_block)[j];
          }
    return 1; //when sucessfull returns 1
}
```

Figure 11. AES-GCM – One Block at a Time with Bit Reflection (to Be Used with the Multiplication Function from Figure 7).

```c
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

#define REFLECT(X)\
    hlp1 = _mm_srli_epi16(X,4);\
    X = _mm_and_si128(AMASK,X);\
    hlp1 = _mm_and_si128(AMASK,hlp1);\
    X = _mm_shuffle_epi8(MASKH,X);\
    hlp1 = _mm_shuffle_epi8(MASKL,hlp1);\
    X = _mm_xor_si128(X,hlp1)

extern void gfmul(__m128i a,__m128i b,__m128i* c); //must be implemented elsewhere

void AES_GCM_encrypt(const unsigned char *in,
                     unsigned char *out,
                     const unsigned char* addt,
                     const unsigned char* ivec,
                     unsigned char *tag,
                     int nbytes,
                     int abytes,
                     int ibytes,
                     const unsigned char* key,
                     int nr)
    {
    int i, j ,k;
    __m128i hlp1, hlp2, hlp3, hlp4;
    __m128i tmp1, tmp2, tmp3, tmp4;
    __m128i H, Y, T;
    __m128i *KEY = (__m128i*)key;
    __m128i ctr1, ctr2, ctr3, ctr4;
    __m128i last_block = _mm_setzero_si128();
    __m128i ONE = _mm_set_epi32(0, 1, 0, 0);
    __m128i FOUR = _mm_set_epi32(0, 4, 0, 0);
    __m128i BSWAP_EPI64 = _mm_set_epi8(8, 9, 10, 11, 12, 13, 14, 15, 0, 1, 2, 3, 4,
5, 6, 7);
    __m128i BSWAP_MASK = _mm_set_epi8(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15);
    __m128i X = _mm_setzero_si128();
    __m128i AMASK = _mm_set_epi32(0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f, 0x0f0f0f0f);
    __m128i MASKL = _mm_set_epi32(0x0f070b03, 0x0d050901, 0x0e060a02, 0x0c040800);
    __m128i MASKH = _mm_set_epi32(0xf070b030, 0xd0509010, 0xe060a020, 0xc0408000);
    __m128i MASKF = _mm_set_epi32(0x00010203, 0x04050607, 0x08090a0b, 0x0c0d0e0f);

    if(ibytes == 96/8){
        Y = _mm_loadu_si128((__m128i*)ivec);
        Y = _mm_insert_epi32(Y, 0x1000000, 3);
        /*(Compute E[ZERO, KS] and E[Y0, KS] together*/
        tmp1 = _mm_xor_si128(X, KEY[0]);
        tmp2 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr-1; j+=2) {
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            };
        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);
```

```
            H = _mm_aesenclast_si128(tmp1, KEY[nr]);
            T = _mm_aesenclast_si128(tmp2, KEY[nr]);


            REFLECT(H);
            }
    else{
            tmp1 = _mm_xor_si128(X, KEY[0]);
            for(j=1; j < nr; j++)
                    tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            H = _mm_aesenclast_si128(tmp1, KEY[nr]);
            REFLECT(H);
            Y = _mm_xor_si128(Y, Y);


            for(i=0; i < ibytes/16; i++){
                    tmp1 = _mm_loadu_si128(&((__m128i*)ivec)[i]);
                    REFLECT(tmp1);
                    Y = _mm_xor_si128(Y, tmp1);
                    gfmul(Y, H, &Y);
                    }
            if(ibytes%16){
                    for(j=0; j < ibytes%16; j++)
                            ((unsigned char*)&last_block)[j] = ivec[i*16+j];
                    tmp1 = last_block;
                    REFLECT(tmp1);
                    Y = _mm_xor_si128(Y, tmp1);
                    gfmul(Y, H, &Y);
                    }
            tmp1 = _mm_insert_epi64(tmp1, ibytes*8, 0);
            tmp1 = _mm_insert_epi64(tmp1, 0, 1);
            REFLECT(tmp1);
            tmp1 = _mm_shuffle_epi8(tmp1, MASKF);
            Y = _mm_xor_si128(Y, tmp1);
            gfmul(Y, H, &Y);
            REFLECT(Y);
            /*Compute E(K, Y0)*/
            tmp1 = _mm_xor_si128(Y, KEY[0]);
            for(j=1; j < nr; j++)
                    tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            T = _mm_aesenclast_si128(tmp1, KEY[nr]);
    }

    for(i=0; i<abytes/16; i++){
            tmp1 = _mm_loadu_si128(&((__m128i*)addt)[i]);
            REFLECT(tmp1);
            X = _mm_xor_si128(X, tmp1);
            gfmul(X, H, &X);
            }
    if(abytes%16){
            last_block = _mm_setzero_si128();
            for(j=0; j<abytes%16; j++)
                    ((unsigned char*)&last_block)[j] = addt[i*16+j];
            tmp1 = last_block;
            REFLECT(tmp1);
            X = _mm_xor_si128(X, tmp1);
            gfmul(X, H, &X);
            }

    ctr1 = _mm_shuffle_epi8(Y, BSWAP_EPI64);
    ctr1 = _mm_add_epi32(ctr1, ONE);
    ctr2 = _mm_add_epi32(ctr1, ONE);
```

```
    ctr3 = _mm_add_epi32(ctr2, ONE);
    ctr4 = _mm_add_epi32(ctr3, ONE);

    for(i=0; i < nbytes/16/4; i++){
        tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
        tmp2 = _mm_shuffle_epi8(ctr2, BSWAP_EPI64);
        tmp3 = _mm_shuffle_epi8(ctr3, BSWAP_EPI64);
        tmp4 = _mm_shuffle_epi8(ctr4, BSWAP_EPI64);

        ctr1 = _mm_add_epi32(ctr1, FOUR);
        ctr2 = _mm_add_epi32(ctr2, FOUR);
        ctr3 = _mm_add_epi32(ctr3, FOUR);
        ctr4 = _mm_add_epi32(ctr4, FOUR);

        tmp1 =_mm_xor_si128(tmp1, KEY[0]);
        tmp2 =_mm_xor_si128(tmp2, KEY[0]);
        tmp3 =_mm_xor_si128(tmp3, KEY[0]);
        tmp4 =_mm_xor_si128(tmp4, KEY[0]);

        for(j=1; j < nr-1; j+=2){
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j+1]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j+1]);
            }

        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);
        tmp3 = _mm_aesenc_si128(tmp3, KEY[nr-1]);
        tmp4 = _mm_aesenc_si128(tmp4, KEY[nr-1]);

        tmp1 =_mm_aesenclast_si128(tmp1, KEY[nr]);
        tmp2 =_mm_aesenclast_si128(tmp2, KEY[nr]);
        tmp3 =_mm_aesenclast_si128(tmp3, KEY[nr]);
        tmp4 =_mm_aesenclast_si128(tmp4, KEY[nr]);

        tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[i*4+0]));
        tmp2 = _mm_xor_si128(tmp2, _mm_loadu_si128(&((__m128i*)in)[i*4+1]));
        tmp3 = _mm_xor_si128(tmp3, _mm_loadu_si128(&((__m128i*)in)[i*4+2]));
        tmp4 = _mm_xor_si128(tmp4, _mm_loadu_si128(&((__m128i*)in)[i*4+3]));

        _mm_storeu_si128(&((__m128i*)out)[i*4+0], tmp1);
        _mm_storeu_si128(&((__m128i*)out)[i*4+1], tmp2);
        _mm_storeu_si128(&((__m128i*)out)[i*4+2], tmp3);
        _mm_storeu_si128(&((__m128i*)out)[i*4+3], tmp4);

        REFLECT(tmp1);
        REFLECT(tmp2);
        REFLECT(tmp3);
        REFLECT(tmp4);

        X = _mm_xor_si128(X, tmp1);
        gfmul(X, H, &X);
        X = _mm_xor_si128(X, tmp2);
        gfmul(X, H, &X);
        X = _mm_xor_si128(X, tmp3);
        gfmul(X, H, &X);
        X = _mm_xor_si128(X, tmp4);
```

```
          gfmul(X, H, &X);
          }
     for(k = i*4; k < nbytes/16; k++){
          tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
          ctr1 = _mm_add_epi32(ctr1, ONE);
          tmp1 = _mm_xor_si128(tmp1, KEY[0]);
          for(j=1; j<nr-1; j+=2){
               tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
               tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
               }
          tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
          tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
          tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
          _mm_storeu_si128(&((__m128i*)out)[k], tmp1);
          REFLECT(tmp1);
          X =_mm_xor_si128(X, tmp1);
          gfmul(X, H, &X);
          }
//If one partial block remains
     if(nbytes%16){
          tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
          tmp1 = _mm_xor_si128(tmp1, KEY[0]);
          for(j=1; j<nr-1; j+=2){
               tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
               tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
               }
          tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
          tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
          tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
          last_block = tmp1;
          for(j=0; j < nbytes%16; j++)
               out[k*16+j]=((unsigned char*)&last_block)[j];
          for(j; j<16; j++)
               ((unsigned char*)&last_block)[j]=0;
          tmp1 = last_block;
          REFLECT(tmp1);
          X =_mm_xor_si128(X, tmp1);
          gfmul(X, H, &X);
          }
     tmp1 = _mm_insert_epi64(tmp1, nbytes*8, 0);
     tmp1 = _mm_insert_epi64(tmp1, abytes*8, 1);
     REFLECT(tmp1);
     tmp1 = _mm_shuffle_epi8(tmp1, MASKF);
     X = _mm_xor_si128(X, tmp1);
     gfmul(X, H, &X);
     REFLECT(X);
     T = _mm_xor_si128(X, T);
     _mm_storeu_si128((__m128i*)tag, T);
}
```

Figure 12. AES-GCM: Processing Four Blocks in Parallel with Aggregated Every Four Blocks

```
#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>

extern void gfmul(__m128i a,__m128i b, __m128i* c); //must be implemented elsewhere

extern void reduce4 (__m128i H1,__m128i H2,__m128i H3,__m128i H4,
            __m128i X1,__m128i X2,__m128i X3, __m128i X4, __m128i *res);

void AES_GCM_encrypt(const unsigned char *in,
                     unsigned char *out,
                     const unsigned char* addt,
                     const unsigned char* ivec,
                     unsigned char *tag,
                     int nbytes,
                     int abytes,
                     int ibytes,
                     const unsigned char* key,
                     int nr)
    {
    int i, j ,k;
    __m128i hlp1, hlp2, hlp3, hlp4;
    __m128i tmp1, tmp2, tmp3, tmp4;
    __m128i H, H1, H2, H3, H4, Y, T;
    __m128i *KEY = (__m128i*)key;
    __m128i ctr1, ctr2, ctr3, ctr4;
    __m128i last_block = _mm_setzero_si128();
    __m128i ONE = _mm_set_epi32(0, 1, 0, 0);
    __m128i FOUR = _mm_set_epi32(0, 4, 0, 0);
    __m128i BSWAP_EPI64 = _mm_set_epi8(8,9,10,11,12,13,14,15,0,1,2,3,4,5,6,7);
    __m128i BSWAP_MASK = _mm_set_epi8(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
    __m128i X = _mm_setzero_si128();

    if(ibytes == 96/8){
        Y = _mm_loadu_si128((__m128i*)ivec);
        Y = _mm_insert_epi32(Y, 0x1000000, 3);
        /*(Compute E[ZERO, KS] and E[Y0, KS] together*/
        tmp1 = _mm_xor_si128(X, KEY[0]);
        tmp2 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr-1; j+=2) {
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            };
        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);

        H = _mm_aesenclast_si128(tmp1, KEY[nr]);
        T = _mm_aesenclast_si128(tmp2, KEY[nr]);

        H = _mm_shuffle_epi8(H, BSWAP_MASK);
        }
    else{
        tmp1 = _mm_xor_si128(X, KEY[0]);
        for(j=1; j <nr; j++)
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
        H = _mm_aesenclast_si128(tmp1, KEY[nr]);
        H = _mm_shuffle_epi8(H, BSWAP_MASK);
        Y = _mm_xor_si128(Y, Y);
```

```
        for(i=0; i < ibytes/16; i++){
            tmp1 = _mm_loadu_si128(&((__m128i*)ivec)[i]);
            tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
            Y = _mm_xor_si128(Y, tmp1);
            gfmul(Y, H, &Y);
            }
        if(ibytes%16){
            for(j=0; j < ibytes%16; j++)
                ((unsigned char*)&last_block)[j] = ivec[i*16+j];
            tmp1 = last_block;
            tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
            Y = _mm_xor_si128(Y, tmp1);
            gfmul(Y, H, &Y);
            }
        tmp1 = _mm_insert_epi64(tmp1, ibytes*8, 0);
        tmp1 = _mm_insert_epi64(tmp1, 0, 1);

        Y = _mm_xor_si128(Y, tmp1);
        gfmul(Y, H, &Y);
        Y = _mm_shuffle_epi8(Y, BSWAP_MASK);
        /*Compute E(K, Y0)*/
        tmp1 = _mm_xor_si128(Y, KEY[0]);
        for(j=1; j < nr; j++)
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
        T = _mm_aesenclast_si128(tmp1, KEY[nr]);
    }

    gfmul(H,H,&H2);
    gfmul(H,H2,&H3);
    gfmul(H,H3,&H4);

    for(i=0; i<abytes/16/4; i++){
        tmp1 = _mm_loadu_si128(&((__m128i*)addt)[i*4]);
        tmp2 = _mm_loadu_si128(&((__m128i*)addt)[i*4+1]);
        tmp3 = _mm_loadu_si128(&((__m128i*)addt)[i*4+2]);
        tmp4 = _mm_loadu_si128(&((__m128i*)addt)[i*4+3]);

        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        tmp2 = _mm_shuffle_epi8(tmp2, BSWAP_MASK);
        tmp3 = _mm_shuffle_epi8(tmp3, BSWAP_MASK);
        tmp4 = _mm_shuffle_epi8(tmp4, BSWAP_MASK);

        tmp1 = _mm_xor_si128(X, tmp1);

        reduce4(H, H2, H3, H4, tmp4, tmp3, tmp2, tmp1, &X);
        }
    for(i=i*4; i<abytes/16; i++){
        tmp1 = _mm_loadu_si128(&((__m128i*)addt)[i]);
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X,tmp1);
        gfmul(X, H, &X);
        }
    if(abytes%16){
        last_block = _mm_setzero_si128();
        for(j=0; j<abytes%16; j++)
            ((unsigned char*)&last_block)[j] = addt[i*16+j];
        tmp1 = last_block;
        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        X = _mm_xor_si128(X,tmp1);
```

```
        gfmul(X,H,&X);
        }

    ctr1 = _mm_shuffle_epi8(Y, BSWAP_EPI64);
    ctr1 = _mm_add_epi32(ctr1, ONE);
    ctr2 = _mm_add_epi32(ctr1, ONE);
    ctr3 = _mm_add_epi32(ctr2, ONE);
    ctr4 = _mm_add_epi32(ctr3, ONE);

    for(i=0; i<nbytes/16/4; i++){
        tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
        tmp2 = _mm_shuffle_epi8(ctr2, BSWAP_EPI64);
        tmp3 = _mm_shuffle_epi8(ctr3, BSWAP_EPI64);
        tmp4 = _mm_shuffle_epi8(ctr4, BSWAP_EPI64);

        ctr1 = _mm_add_epi32(ctr1, FOUR);
        ctr2 = _mm_add_epi32(ctr2, FOUR);
        ctr3 = _mm_add_epi32(ctr3, FOUR);
        ctr4 = _mm_add_epi32(ctr4, FOUR);

        tmp1 =_mm_xor_si128(tmp1, KEY[0]);
        tmp2 =_mm_xor_si128(tmp2, KEY[0]);
        tmp3 =_mm_xor_si128(tmp3, KEY[0]);
        tmp4 =_mm_xor_si128(tmp4, KEY[0]);

        for(j=1; j<nr-1; j+=2){
            tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j]);

            tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
            tmp2 = _mm_aesenc_si128(tmp2, KEY[j+1]);
            tmp3 = _mm_aesenc_si128(tmp3, KEY[j+1]);
            tmp4 = _mm_aesenc_si128(tmp4, KEY[j+1]);
            }

        tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
        tmp2 = _mm_aesenc_si128(tmp2, KEY[nr-1]);
        tmp3 = _mm_aesenc_si128(tmp3, KEY[nr-1]);
        tmp4 = _mm_aesenc_si128(tmp4, KEY[nr-1]);

        tmp1 =_mm_aesenclast_si128(tmp1, KEY[nr]);
        tmp2 =_mm_aesenclast_si128(tmp2, KEY[nr]);
        tmp3 =_mm_aesenclast_si128(tmp3, KEY[nr]);
        tmp4 =_mm_aesenclast_si128(tmp4, KEY[nr]);

        tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[i*4+0]));
        tmp2 = _mm_xor_si128(tmp2, _mm_loadu_si128(&((__m128i*)in)[i*4+1]));
        tmp3 = _mm_xor_si128(tmp3, _mm_loadu_si128(&((__m128i*)in)[i*4+2]));
        tmp4 = _mm_xor_si128(tmp4, _mm_loadu_si128(&((__m128i*)in)[i*4+3]));

        _mm_storeu_si128(&((__m128i*)out)[i*4+0], tmp1);
        _mm_storeu_si128(&((__m128i*)out)[i*4+1], tmp2);
        _mm_storeu_si128(&((__m128i*)out)[i*4+2], tmp3);
        _mm_storeu_si128(&((__m128i*)out)[i*4+3], tmp4);

        tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
        tmp2 = _mm_shuffle_epi8(tmp2, BSWAP_MASK);
        tmp3 = _mm_shuffle_epi8(tmp3, BSWAP_MASK);
        tmp4 = _mm_shuffle_epi8(tmp4, BSWAP_MASK);

        tmp1 = _mm_xor_si128(X, tmp1);
```

```
                reduce4(H, H2, H3, H4, tmp4, tmp3, tmp2, tmp1, &X);
            }
        for(k=i*4; k<nbytes/16; k++){
            tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
            ctr1 = _mm_add_epi32(ctr1, ONE);
            tmp1 = _mm_xor_si128(tmp1, KEY[0]);
            for(j=1; j<nr-1; j+=2){
                tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
                tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
                }
            tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
            tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
            tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
            _mm_storeu_si128(&((__m128i*)out)[k], tmp1);
            tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
            X = _mm_xor_si128(X, tmp1);
            gfmul(X,H,&X);
            }
//If remains one incomplete block
        if(nbytes%16){
            tmp1 = _mm_shuffle_epi8(ctr1, BSWAP_EPI64);
            tmp1 = _mm_xor_si128(tmp1, KEY[0]);
            for(j=1; j<nr-1; j+=2){
                tmp1 = _mm_aesenc_si128(tmp1, KEY[j]);
                tmp1 = _mm_aesenc_si128(tmp1, KEY[j+1]);
                }
            tmp1 = _mm_aesenc_si128(tmp1, KEY[nr-1]);
            tmp1 = _mm_aesenclast_si128(tmp1, KEY[nr]);
            tmp1 = _mm_xor_si128(tmp1, _mm_loadu_si128(&((__m128i*)in)[k]));
            last_block = tmp1;
            for(j=0; j<nbytes%16; j++)
                out[k*16+j] = ((unsigned char*)&last_block)[j];
            for(j; j<16; j++)
                ((unsigned char*)&last_block)[j] = 0;
            tmp1 = last_block;
            tmp1 = _mm_shuffle_epi8(tmp1, BSWAP_MASK);
            X = _mm_xor_si128(X, tmp1);
            gfmul(X, H, &X);
            }
        tmp1 = _mm_insert_epi64(tmp1, nbytes*8, 0);
        tmp1 = _mm_insert_epi64(tmp1, abytes*8, 1);

        X = _mm_xor_si128(X, tmp1);
        gfmul(X,H,&X);
        X = _mm_shuffle_epi8(X, BSWAP_MASK);
        T = _mm_xor_si128(X, T);
        _mm_storeu_si128((__m128i*)tag, T);
}
```

Figure 13. AES128 Key Expansion

```
#include <wmmintrin.h>
#include <emmintrin.h>
#include <stdint.h>

#if !defined (ALIGN16)
#if defined (__GNUC__)
#  define ALIGN16  __attribute__  ( (aligned (16)))
# else
#  define ALIGN16 __declspec (align (16))
# endif
#endif

typedef struct KEY_SCHEDULE
{
   ALIGN16 unsigned char KEY[16*15];
   unsigned int nr;
} AES_KEY;

inline void key_expansion_128(__m128i* temp1,
                              __m128i* temp2,
                               int KS_Pointer,
                              __m128i *Key_Schedule)

    {
    __m128i temp3;

    *temp2 = _mm_shuffle_epi32 (*temp2, 0xff);
    temp3  = _mm_slli_si128 (*temp1, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp3);
    temp3  = _mm_slli_si128 (temp3, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp3);
    temp3  = _mm_slli_si128 (temp3, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp3);
    *temp1 = _mm_xor_si128 (*temp1, *temp2);

    Key_Schedule[KS_Pointer]=*temp1;
    }

void AES_128_Key_Expansion (const uint8_t *userkey,
                                    AES_KEY *key)
    {
    key->nr=10;

    __m128i temp1, temp2, temp3;
    __m128i *Key_Schedule=(__m128i*)key->KEY;
    int KS_Pointer=1;
    int i;

    temp1= _mm_loadu_si128((__m128i*)userkey);
    Key_Schedule[0]=temp1;

    temp2 = _mm_aeskeygenassist_si128 (temp1,0x1);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x2);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x4);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x8);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x10);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x20);
```

```
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x40);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x80);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x1b);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x36);
    key_expansion_128(&temp1, &temp2, KS_Pointer++, Key_Schedule);
    }
```

Figure 14. A Main Function for Testing

```
//#define TEST1
//#define TEST2
//#define TEST3
//#define TEST4
//#define TEST5
//#define TEST6
//#define CUSTOM_TEST

#include <wmmintrin.h>
#include <emmintrin.h>
#include <smmintrin.h>
#include <stdint.h>
#include <stdio.h>

#if !defined (ALIGN16)
# if defined (__GNUC__)
#  define ALIGN16 __attribute__  ( (aligned (16)))
# else
#  define ALIGN16 __declspec (align (16))
# endif
#endif
#ifndef CUSTOM_TEST
#ifndef TEST6
#ifndef TEST5
#ifndef TEST4
#ifndef TEST3
#ifndef TEST2
#ifndef TEST1
#define TEST1
#endif
#endif
#endif
#endif
#endif
#endif
#endif

typedef struct KEY_SCHEDULE{
    ALIGN16 uint8_t KEY[16*15];
    int nr;
    }AES_KEY;

#define cpuid(func,ax,bx,cx,dx)\
         __asm__ __volatile__ ("cpuid":\
         "=a" (ax), "=b" (bx), "=c" (cx), "=d" (dx) : "a" (func));
```

```c
extern   AES_GCM_encrypt(void*, void*, void*, void*, void*, uint64_t, uint64_t,
uint64_t, void*, uint64_t);

int Check_CPU_support_AES()
    {
    unsigned int a,b,c,d;
    cpuid(1, a,b,c,d);
    return (c & 0x2000000);
    }

/***************************************************************************/
void print_m128i_with_string(char* string,__m128i data)
    {
    unsigned char *pointer = (unsigned char*)&data;
    int i;
    printf("%-40s[0x",string);
    for (i=0; i<16; i++)
        printf("%02x",pointer[i]);
    printf("]\n");
    }
void print_m128i_with_string_short(char* string,__m128i data,int length)
    {
    unsigned char *pointer = (unsigned char*)&data;
    int i;
    printf("%-40s[0x",string);
    for (i=0; i<length; i++)
        printf("%02x",pointer[i]);
    printf("]\n");
    }
/***************************************************************************/
int main (){
//The test vectors are taken from:
//http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf
#ifdef TEST1
#define NBYTES 0
#define MBYTES 0
#define IBYTES 12
    ALIGN16 uint8_t K[16]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                           0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
    ALIGN16 uint8_t P[NBYTES]={};
    ALIGN16 uint8_t IV[IBYTES]={0x00,0x00,0x00,0x00,0x00,0x00,
                                0x00,0x00,0x00,0x00,0x00,0x00};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={};
    ALIGN16 uint8_t EXPECTED_T[16]={0x58,0xe2,0xfc,0xce,0xfa,0x7e,0x30,0x61,
                                    0x36,0x7f,0x1d,0x57,0xa4,0xe7,0x45,0x5a};
    ALIGN16 uint8_t A[MBYTES]={};
#endif

#ifdef TEST2
#define NBYTES 16
#define MBYTES 0
#define IBYTES 12
    ALIGN16 uint8_t K[16]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                           0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
    ALIGN16 uint8_t P[NBYTES]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                               0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};
    ALIGN16 uint8_t IV[IBYTES]={0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
                                0x00,0x00,0x00,0x00};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={0x03,0x88,0xda,0xce,0x60,0xb6,0xa3,0x92,
                                        0xf3,0x28,0xc2,0xb9,0x71,0xb2,0xfe,0x78};
    ALIGN16 uint8_t EXPECTED_T[16]={0xab,0x6e,0x47,0xd4,0x2c,0xec,0x13,0xbd,
                                    0xf5,0x3a,0x67,0xb2,0x12,0x57,0xbd,0xdf};
    ALIGN16 uint8_t A[MBYTES]={};
```

```
#endif

#ifdef TEST3
#define NBYTES 64
#define MBYTES 0
#define IBYTES 12
    ALIGN16 uint8_t K[16]={0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,
                           0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08};
    ALIGN16 uint8_t P[NBYTES]={0xd9,0x31,0x32,0x25,0xf8,0x84,0x06,0xe5,
                               0xa5,0x59,0x09,0xc5,0xaf,0xf5,0x26,0x9a,
                               0x86,0xa7,0xa9,0x53,0x15,0x34,0xf7,0xda,
                               0x2e,0x4c,0x30,0x3d,0x8a,0x31,0x8a,0x72,
                               0x1c,0x3c,0x0c,0x95,0x95,0x68,0x09,0x53,
                               0x2f,0xcf,0x0e,0x24,0x49,0xa6,0xb5,0x25,
                               0xb1,0x6a,0xed,0xf5,0xaa,0x0d,0xe6,0x57,
                               0xba,0x63,0x7b,0x39,0x1a,0xaf,0xd2,0x55};
    ALIGN16 uint8_t IV[IBYTES]={0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad,
                                0xde,0xca,0xf8,0x88};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={0x42,0x83,0x1e,0xc2,0x21,0x77,0x74,0x24,
                                        0x4b,0x72,0x21,0xb7,0x84,0xd0,0xd4,0x9c,
                                        0xe3,0xaa,0x21,0x2f,0x2c,0x02,0xa4,0xe0,
                                        0x35,0xc1,0x7e,0x23,0x29,0xac,0xa1,0x2e,
                                        0x21,0xd5,0x14,0xb2,0x54,0x66,0x93,0x1c,
                                        0x7d,0x8f,0x6a,0x5a,0xac,0x84,0xaa,0x05,
                                        0x1b,0xa3,0x0b,0x39,0x6a,0x0a,0xac,0x97,
                                        0x3d,0x58,0xe0,0x91,0x47,0x3f,0x59,0x85};
    ALIGN16 uint8_t EXPECTED_T[16]={0x4d,0x5c,0x2a,0xf3,0x27,0xcd,0x64,0xa6,
                                    0x2c,0xf3,0x5a,0xbd,0x2b,0xa6,0xfa,0xb4};
    ALIGN16 uint8_t A[MBYTES]={};
#endif

#ifdef TEST4
#define NBYTES 60
#define MBYTES 20
#define IBYTES 12
    ALIGN16 uint8_t K[16]={0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,
                           0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08};
    ALIGN16 uint8_t P[NBYTES]={0xd9,0x31,0x32,0x25,0xf8,0x84,0x06,0xe5,
                               0xa5,0x59,0x09,0xc5,0xaf,0xf5,0x26,0x9a,
                               0x86,0xa7,0xa9,0x53,0x15,0x34,0xf7,0xda,
                               0x2e,0x4c,0x30,0x3d,0x8a,0x31,0x8a,0x72,
                               0x1c,0x3c,0x0c,0x95,0x95,0x68,0x09,0x53,
                               0x2f,0xcf,0x0e,0x24,0x49,0xa6,0xb5,0x25,
                               0xb1,0x6a,0xed,0xf5,0xaa,0x0d,0xe6,0x57,
                               0xba,0x63,0x7b,0x39};
    ALIGN16 uint8_t IV[IBYTES]={0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad,
                                0xde,0xca,0xf8,0x88};
    ALIGN16 uint8_t A[MBYTES]={0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,
                               0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,
                               0xab,0xad,0xda,0xd2};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={0x42,0x83,0x1e,0xc2,0x21,0x77,0x74,0x24,
                                        0x4b,0x72,0x21,0xb7,0x84,0xd0,0xd4,0x9c,
                                        0xe3,0xaa,0x21,0x2f,0x2c,0x02,0xa4,0xe0,
                                        0x35,0xc1,0x7e,0x23,0x29,0xac,0xa1,0x2e,
                                        0x21,0xd5,0x14,0xb2,0x54,0x66,0x93,0x1c,
                                        0x7d,0x8f,0x6a,0x5a,0xac,0x84,0xaa,0x05,
                                        0x1b,0xa3,0x0b,0x39,0x6a,0x0a,0xac,0x97,
                                        0x3d,0x58,0xe0,0x91};
    ALIGN16 uint8_t EXPECTED_T[16]={0x5b,0xc9,0x4f,0xbc,0x32,0x21,0xa5,0xdb,
                                    0x94,0xfa,0xe9,0x5a,0xe7,0x12,0x1a,0x47};
```

```
#endif

#ifdef TEST5
#define NBYTES 60
#define MBYTES 20
#define IBYTES 8
    ALIGN16 uint8_t K[16]={0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,
                        0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08};
    ALIGN16 uint8_t P[NBYTES]={0xd9,0x31,0x32,0x25,0xf8,0x84,0x06,0xe5,
                            0xa5,0x59,0x09,0xc5,0xaf,0xf5,0x26,0x9a,
                            0x86,0xa7,0xa9,0x53,0x15,0x34,0xf7,0xda,
                            0x2e,0x4c,0x30,0x3d,0x8a,0x31,0x8a,0x72,
                            0x1c,0x3c,0x0c,0x95,0x95,0x68,0x09,0x53,
                            0x2f,0xcf,0x0e,0x24,0x49,0xa6,0xb5,0x25,
                            0xb1,0x6a,0xed,0xf5,0xaa,0x0d,0xe6,0x57,
                            0xba,0x63,0x7b,0x39};
    ALIGN16 uint8_t A[MBYTES]={0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,
                            0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,
                            0xab,0xad,0xda,0xd2};
    ALIGN16 uint8_t IV[IBYTES]={0xca,0xfe,0xba,0xbe,0xfa,0xce,0xdb,0xad};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={0x61,0x35,0x3b,0x4c,0x28,0x06,0x93,0x4a,
                                    0x77,0x7f,0xf5,0x1f,0xa2,0x2a,0x47,0x55,
                                    0x69,0x9b,0x2a,0x71,0x4f,0xcd,0xc6,0xf8,
                                    0x37,0x66,0xe5,0xf9,0x7b,0x6c,0x74,0x23,
                                    0x73,0x80,0x69,0x00,0xe4,0x9f,0x24,0xb2,
                                    0x2b,0x09,0x75,0x44,0xd4,0x89,0x6b,0x42,
                                    0x49,0x89,0xb5,0xe1,0xeb,0xac,0x0f,0x07,
                                    0xc2,0x3f,0x45,0x98};
    ALIGN16 uint8_t EXPECTED_T[16]={0x36,0x12,0xd2,0xe7,0x9e,0x3b,0x07,0x85,
                                  0x56,0x1b,0xe1,0x4a,0xac,0xa2,0xfc,0xcb};
#endif

#ifdef TEST6
#define NBYTES 60
#define MBYTES 20
#define IBYTES 60
    ALIGN16 uint8_t K[16]={0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,
                        0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08};
    ALIGN16 uint8_t P[NBYTES]={0xd9,0x31,0x32,0x25,0xf8,0x84,0x06,0xe5,
                            0xa5,0x59,0x09,0xc5,0xaf,0xf5,0x26,0x9a,
                            0x86,0xa7,0xa9,0x53,0x15,0x34,0xf7,0xda,
                            0x2e,0x4c,0x30,0x3d,0x8a,0x31,0x8a,0x72,
                            0x1c,0x3c,0x0c,0x95,0x95,0x68,0x09,0x53,
                            0x2f,0xcf,0x0e,0x24,0x49,0xa6,0xb5,0x25,
                            0xb1,0x6a,0xed,0xf5,0xaa,0x0d,0xe6,0x57,
                            0xba,0x63,0x7b,0x39};
    ALIGN16 uint8_t A[MBYTES]={0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,
                            0xfe,0xed,0xfa,0xce,0xde,0xad,0xbe,0xef,
                            0xab,0xad,0xda,0xd2};
    ALIGN16 uint8_t IV[IBYTES]={0x93,0x13,0x22,0x5d,0xf8,0x84,0x06,0xe5,
                             0x55,0x90,0x9c,0x5a,0xff,0x52,0x69,0xaa,
                             0x6a,0x7a,0x95,0x38,0x53,0x4f,0x7d,0xa1,
                             0xe4,0xc3,0x03,0xd2,0xa3,0x18,0xa7,0x28,
                             0xc3,0xc0,0xc9,0x51,0x56,0x80,0x95,0x39,
                             0xfc,0xf0,0xe2,0x42,0x9a,0x6b,0x52,0x54,
                             0x16,0xae,0xdb,0xf5,0xa0,0xde,0x6a,0x57,
                             0xa6,0x37,0xb3,0x9b};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={0x8c,0xe2,0x49,0x98,0x62,0x56,0x15,0xb6,
                                    0x03,0xa0,0x33,0xac,0xa1,0x3f,0xb8,0x94,
                                    0xbe,0x91,0x12,0xa5,0xc3,0xa2,0x11,0xa8,
                                    0xba,0x26,0x2a,0x3c,0xca,0x7e,0x2c,0xa7,
                                    0x01,0xe4,0xa9,0xa4,0xfb,0xa4,0x3c,0x90,
                                    0xcc,0xdc,0xb2,0x81,0xd4,0x8c,0x7c,0x6f,
```

```
                                          0xd6,0x28,0x75,0xd2,0xac,0xa4,0x17,0x03,
                                          0x4c,0x34,0xae,0xe5};
    ALIGN16 uint8_t EXPECTED_T[16]={0x61,0x9c,0xc5,0xae,0xff,0xfe,0x0b,0xfa,
                                    0x46,0x2a,0xf4,0x3c,0x16,0x99,0xd0,0x50};
#endif

#ifdef CUSTOM_TEST
#ifndef NBLOCKS
#define NBLOCKS 64
#endif
#define NBYTES NBLOCKS*16
#define MBYTES 0
#define IBYTES 12
    ALIGN16 uint8_t K[16]={0xfe,0xff,0xe9,0x92,0x86,0x65,0x73,0x1c,
                           0x6d,0x6a,0x8f,0x94,0x67,0x30,0x83,0x08};
    ALIGN16 uint8_t P[NBYTES]={0xd9,0x31,0x32,0x25,0xf8,0x84,0x06,0xe5,
                               0xa5,0x59,0x09,0xc5,0xaf,0xf5,0x26,0x9a,
                               0x86,0xa7,0xa9,0x53,0x15,0x34,0xf7,0xda,
                               0x2e,0x4c,0x30,0x3d,0x8a,0x31,0x8a,0x72,
                               0x1c,0x3c,0x0c,0x95,0x95,0x68,0x09,0x53,
                               0x2f,0xcf,0x0e,0x24,0x49,0xa6,0xb5,0x25,
                               0xb1,0x6a,0xed,0xf5,0xaa,0x0d,0xe6,0x57,
                               0xba,0x63,0x7b,0x39};
    ALIGN16 uint8_t A[MBYTES]={};
    ALIGN16 uint8_t
IV[IBYTES]={0x93,0x13,0x22,0x5d,0xf8,0x84,0x06,0xe5,0x55,0x90,0x9c,0x5a};
    ALIGN16 uint8_t EXPECTED_C[NBYTES]={0};
    ALIGN16 uint8_t EXPECTED_T[16]={0};
#endif
    ALIGN16 uint8_t C[NBYTES],DECRYPTED_TEXT[NBYTES];
    __m128i T;
    int i,j,tag_correct=1,cipher_correct=1,decrypt_correct=1,decrypt_success;

     /* Verify AES support */
    if (Check_CPU_support_AES( ) == 0) {
        printf ("CPU does not support AES instructions. Bailing out. \n");
        exit (1);
    }
    else printf("CPU check passed. AES instructions are supported.\n\n");

    AES_KEY key;
    AES_128_Key_Expansion (K, key.KEY);
    key.nr = 10;

    AES_GCM_encrypt(P, C, A, IV, &T, NBYTES, MBYTES, IBYTES, key.KEY, key.nr);

    decrypt_success = AES_GCM_decrypt(C, DECRYPTED_TEXT, A, IV, &T, NBYTES, MBYTES,
IBYTES, key.KEY, key.nr);
/****************************************************************************
Here we print the results of the test
****************************************************************************/
    print_m128i_with_string("The Key:", *(__m128i*)K);
    printf("\n");

    for(i=0;i<IBYTES/16;i++)
        print_m128i_with_string("The IV:", ((__m128i*)IV)[i]);
    if(IBYTES%16)
        print_m128i_with_string_short("The IV:", ((__m128i*)IV)[i], IBYTES%16);
    printf("\n");
```

```
    for(i=0;i<MBYTES/16;i++)
        print_m128i_with_string("The header buffer:", ((__m128i*)A)[i]);
    if(MBYTES%16)
        print_m128i_with_string_short("The header buffer:", ((__m128i*)A)[i],
MBYTES%16);
    if(!MBYTES) printf("NONE\n");
    printf("\n");

    for(i=0;i<NBYTES/16;i++)
        print_m128i_with_string("The PLAINTEXT:", ((__m128i*)P)[i]);
    if(NBYTES%16)
        print_m128i_with_string_short("The PLAINTEXT:", ((__m128i*)P)[i], NBYTES%16);
    if(!NBYTES) printf("NONE\n");
    printf("\n");

    for(i=0;i<NBYTES/16;i++)
        print_m128i_with_string("The CIPHERTEXT:", ((__m128i*)C)[i]);
    if(NBYTES%16)
        print_m128i_with_string_short("The CIPHERTEXT:", ((__m128i*)C)[i],
NBYTES%16);
    if(!NBYTES) printf("NONE\n");
    printf("\n");

    print_m128i_with_string("The tag:",T);
    printf("\n");
//Test correctness
    for(i=0;i<16;i++){
        if(((uint8_t*)&T)[i]!=EXPECTED_T[i]){
            tag_correct=0;
            break;
            }
        }
    for(i=0;i<NBYTES;i++){
        if(C[i]!=EXPECTED_C[i]){
            cipher_correct=0;
            break;
            }
        }
    for(i=0;i<NBYTES;i++){
        if(P[i]!=DECRYPTED_TEXT[i]){
            decrypt_correct=0;
            break;
            }
        }
#ifndef CUSTOM_TEST
    if(tag_correct)
        printf("The tag is equal to the expected tag.\n");
    else
        printf("The tag is NOT EQUAL to the expected tag!!!\n");
    if(cipher_correct)
        printf("The cipher text is equal to the expected cipher text.\n");
    else
        printf("The cipher text is NOT EQUAL to the expected cipher text!!!\n");
#endif
    if(decrypt_success)
        printf("Decryption succeeded.\n");
    else
        printf("Decryption FAILED!!!\n");
    if(decrypt_correct)
        printf("The decrypted text is equal to the original plaintext.\n");
    else
        printf("The decrypted text is NOT EQUAL to the original plaintext!!!\n");
    }
```

Figure 15. AES-GCM (Assembly Code): Processing Four Blocks in Parallel with Aggregated Every Four Blocks

```
.align   8
INS:
.quad 0x1000000
.align   16
ONE:
.quad 0x1,0x0
.align   16
TWO:
.quad 0x2,0x0
.align   16
THREE:
.quad 0x3,0x0
.align   16
FOUR:
.quad 0x4,0x0
.align   16
BSWAP_MASK:
.byte 15,14,13,12,11,10,9,8,7,6,5,4,3,2,1,0
.align   16
NEUTRAL_MASK:
.byte 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
.align   16
BSWAP_EPI_64:
.byte 7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8
.align   16
LOAD_HIGH_BROADCAST_AND_BSWAP:
.byte 15,14,13,12,11,10,9,8,15,14,13,12,11,10,9,8


.globl AES_GCM_encrypt
AES_GCM_encrypt:
# parameter 1: %rdi  # *input
# parameter 2: %rsi  # *output
# parameter 3: %rdx  # *addt
# parameter 4: %rcx  # *ivec
# parameter 5: %r8   # *tag
# parameter 6: %r9   # nbytes
# parameter 7: 128 + %rsp # abytes
# parameter 8: 136 + %rsp # ibytes
# parameter 9: 144 + %rsp # *key_schedule
# parameter 10:152 + %rsp # nr

  pushq %rbp
  pushq       %r8
  pushq       %r9
  pushq %r10
  pushq %r11
  pushq %r12
  pushq %r13
  pushq %r14
  pushq %r15
  pushq %rax
  pushq       %rbx
  pushq       %rcx
  pushq       %rdx
```

```
  pushq       %rsi
  pushq       %rdi

  movq  %rsp, %rbp
  andq  $-64, %rbp
  sub   $192, %rbp

HANDLE_IVEC:
  movq  136(%rsp), %rax
  cmp   $12, %rax
  jne   PROCESS_NON96BIT_IVEC
  jmp   PROCESS_96BIT_IVEC

CALCULATE_POWERS_OF_H:
  movdqa  (%rbp), %xmm1
  movdqa  %xmm1, %xmm2
  call  GFMUL
  movdqa  %xmm1, 16(%rbp)    # 16+rbp holds H^2
  call  GFMUL
  movdqa  %xmm1, 32(%rbp)    # 32+rbp holds H^3
  call  GFMUL
  movdqa  %xmm1, 48(%rbp)    # 48+rbp holds H^4


  pshufd  $78, (%rbp), %xmm8
  pshufd  $78, 16(%rbp), %xmm9
  pshufd  $78, 32(%rbp), %xmm10
  pshufd  $78, 48(%rbp), %xmm11

  pxor  (%rbp), %xmm8
  pxor  16(%rbp), %xmm9
  pxor  32(%rbp), %xmm10
  pxor  48(%rbp), %xmm11

  movdqa   %xmm8, 80(%rbp)
  movdqa   %xmm9, 96(%rbp)
  movdqa   %xmm10, 112(%rbp)
  movdqa   %xmm11, 128(%rbp)

HANDLE_ADDT:
  pxor %xmm1, %xmm1
  movq  128(%rsp), %r10    # abytes
  movq  %r10, %r11
  movq  %r10, %r12

  shl   $60, %r12
  shl   $58, %r11
  shr   $6, %r10
  je  ADDT_SINGLES

ADDT_QUADS:
#hash 4 blocks of header at a time
  movdqu  (%rdx), %xmm8
  movdqu  16(%rdx), %xmm9
  movdqu  32(%rdx), %xmm10
  movdqu  48(%rdx), %xmm11

  add     $64, %rdx
  dec     %r10

  pshufb  (BSWAP_MASK), %xmm8
  pshufb  (BSWAP_MASK), %xmm9
```

```
  pshufb  (BSWAP_MASK), %xmm10
  pshufb  (BSWAP_MASK), %xmm11

  pxor %xmm1, %xmm8
  call   REDUCE_FOUR
  jne    ADDT_QUADS

ADDT_SINGLES:     #hash remaining block
  shrq  $62, %r11
  je  ADDT_REMAINDER
  movdqu  (%rbp), %xmm2
ADDT_SINGLES_LOOP:
  movdqu  (%rdx), %xmm10
  add    $16, %rdx
  dec    %r11
  pshufb (BSWAP_MASK), %xmm10
  pxor   %xmm10, %xmm1
  call   GFMUL
  jne    ADDT_SINGLES_LOOP

ADDT_REMAINDER:
#hash the tail (partial block)
  shrq  $60, %r12
  je  HANDLE_DATA
  movdqu  (%rbp), %xmm2
  movq  (%rdx), %rax
  movq  8(%rdx), %rbx
  cmp    $8, %r12
  jl  ADDT_LESS_THEN_8
  jg  ADDT_MORE_THEN_8
  xorq   %rbx, %rbx
  jmp    ADDT_REMAINDER_END
ADDT_LESS_THEN_8:
  movq  $8, %rcx
  subq  %r12, %rcx
  shlq  $3, %rcx
  shlq  %cl, %rax
  shrq  %cl, %rax
  xorq  %rbx, %rbx
  jmp    ADDT_REMAINDER_END
ADDT_MORE_THEN_8:
  movq  $16, %rcx
  subq  %r12, %rcx
  shlq  $3, %rcx
  shlq  %cl, %rbx
  shrq  %cl, %rbx
ADDT_REMAINDER_END:
  pinsrq  $0, %rax, %xmm3
  pinsrq  $1, %rbx, %xmm3
  pshufb  (BSWAP_MASK), %xmm3
  pxor   %xmm3, %xmm1
  call   GFMUL

HANDLE_DATA:
  movq  %r9, %r10
  movq  %r9, %r11
  movq  %r9, %r12

CALCULATE_COUNTERS:
  pshufb  (BSWAP_MASK), %xmm0
```

```
   movdqa %xmm0, 64(%rbp)

   shl    $60, %r12
   shl    $58, %r11
   shr    $6, %r10
   je   DATA_SINGLES

DATA_FIRST_QUAD:
   movq  144(%rsp), %r15
   movq  152(%rsp), %r14

   movdqa   %xmm0, %xmm8
   movdqa   %xmm0, %xmm9
   movdqa   %xmm0, %xmm10
   movdqa   %xmm0, %xmm11

   paddd  (ONE), %xmm8
   paddd  (TWO), %xmm9
   paddd  (THREE), %xmm10
   paddd  (FOUR), %xmm11
   movdqa %xmm11, 64(%rbp)

   pshufb (BSWAP_MASK), %xmm8
   pshufb (BSWAP_MASK), %xmm9
   pshufb (BSWAP_MASK), %xmm10
   pshufb (BSWAP_MASK), %xmm11

   pxor   (%r15), %xmm8
   pxor   (%r15), %xmm9
   pxor   (%r15), %xmm10
   pxor   (%r15), %xmm11

ENC_FIRST_QUAD:
   cmpq  $11, %r14
   movdqa 160(%r15), %xmm7

   aesenc  16(%r15), %xmm8
   aesenc  16(%r15), %xmm9
   aesenc  16(%r15), %xmm10
   aesenc  16(%r15), %xmm11

   aesenc  32(%r15), %xmm8
   aesenc  32(%r15), %xmm9
   aesenc  32(%r15), %xmm10
   aesenc  32(%r15), %xmm11

   aesenc  48(%r15), %xmm8
   aesenc  48(%r15), %xmm9
   aesenc  48(%r15), %xmm10
   aesenc  48(%r15), %xmm11

   aesenc  64(%r15), %xmm8
   aesenc  64(%r15), %xmm9
   aesenc  64(%r15), %xmm10
   aesenc  64(%r15), %xmm11

   aesenc  80(%r15), %xmm8
   aesenc  80(%r15), %xmm9
   aesenc  80(%r15), %xmm10
   aesenc  80(%r15), %xmm11

   aesenc  96(%r15), %xmm8
   aesenc  96(%r15), %xmm9
```

```
 aesenc   96(%r15), %xmm10
 aesenc   96(%r15), %xmm11

 aesenc   112(%r15), %xmm8
 aesenc   112(%r15), %xmm9
 aesenc   112(%r15), %xmm10
 aesenc   112(%r15), %xmm11

 aesenc   128(%r15), %xmm8
 aesenc   128(%r15), %xmm9
 aesenc   128(%r15), %xmm10
 aesenc   128(%r15), %xmm11

 aesenc   144(%r15), %xmm8
 aesenc   144(%r15), %xmm9
 aesenc   144(%r15), %xmm10
 aesenc   144(%r15), %xmm11
 jb  FIRST_QUAD_LAST_ROUND
 cmpq  $13, %r14
 movdqa  192(%r15), %xmm7

 aesenc   160(%r15), %xmm8
 aesenc   160(%r15), %xmm9
 aesenc   160(%r15), %xmm10
 aesenc   160(%r15), %xmm11

 aesenc   176(%r15), %xmm8
 aesenc   176(%r15), %xmm9
 aesenc   176(%r15), %xmm10
 aesenc   176(%r15), %xmm11
 jb  FIRST_QUAD_LAST_ROUND
 movdqa  224(%r15), %xmm7

 aesenc   192(%r15), %xmm8
 aesenc   192(%r15), %xmm9
 aesenc   192(%r15), %xmm10
 aesenc   192(%r15), %xmm11

 aesenc   208(%r15), %xmm8
 aesenc   208(%r15), %xmm9
 aesenc   208(%r15), %xmm10
 aesenc   208(%r15), %xmm11
FIRST_QUAD_LAST_ROUND:
 aesenclast  %xmm7, %xmm8
 aesenclast  %xmm7, %xmm9
 aesenclast  %xmm7, %xmm10
 aesenclast  %xmm7, %xmm11

 pxor  (%rdi), %xmm8
 pxor  16(%rdi), %xmm9
 pxor  32(%rdi), %xmm10
 pxor  48(%rdi), %xmm11

 movdqa  %xmm8, (%rsi)
 movdqa  %xmm9, 16(%rsi)
 movdqa  %xmm10, 32(%rsi)
 movdqa  %xmm11, 48(%rsi)

 add $64, %rsi
 add $64, %rdi
```

```
dec %r10
je FINAL_REDUCTION
jmp DATA_QUADS
.align 64
DATA_QUADS:

movdqa    (BSWAP_MASK), %xmm2

movdqa    %xmm8, %xmm0
movdqa    %xmm9, %xmm13
movdqa    %xmm10, %xmm14
movdqa    %xmm11, %xmm15

movdqa    64(%rbp), %xmm8
movdqa    64(%rbp), %xmm9
movdqa    64(%rbp), %xmm10
movdqa    64(%rbp), %xmm11

pshufb  %xmm2, %xmm0
pshufb  %xmm2, %xmm13
pshufb  %xmm2, %xmm14
pshufb  %xmm2, %xmm15

pxor  %xmm1, %xmm0

movq  144(%rsp), %r15
movq  152(%rsp), %r14

paddd  (ONE), %xmm8
paddd  (TWO), %xmm9
paddd  (THREE), %xmm10
paddd  (FOUR), %xmm11
movdqa %xmm11, 64(%rbp)

pshufb %xmm2, %xmm8
pshufb %xmm2, %xmm9
pshufb %xmm2, %xmm10
pshufb %xmm2, %xmm11

pxor  (%r15), %xmm8
pxor  (%r15), %xmm9
pxor  (%r15), %xmm10
pxor  (%r15), %xmm11


movdqa     %xmm15, %xmm1
pclmulqdq  $0x00, (%rbp), %xmm1
aesenc  16(%r15), %xmm8
aesenc  16(%r15), %xmm9
aesenc  16(%r15), %xmm10
aesenc  16(%r15), %xmm11
movdqa     %xmm14, %xmm3
pclmulqdq  $0x00, 16(%rbp), %xmm3
aesenc  32(%r15), %xmm8
aesenc  32(%r15), %xmm9
aesenc  32(%r15), %xmm10
aesenc  32(%r15), %xmm11
movdqa     %xmm15, %xmm2
pclmulqdq  $0x11, (%rbp), %xmm2
aesenc  48(%r15), %xmm8
aesenc  48(%r15), %xmm9
aesenc  48(%r15), %xmm10
aesenc  48(%r15), %xmm11
```

```
  movdqa    %xmm14, %xmm4
  pclmulqdq $0x11, 16(%rbp), %xmm4
  aesenc  64(%r15), %xmm8
  aesenc  64(%r15), %xmm9
  aesenc  64(%r15), %xmm10
  aesenc  64(%r15), %xmm11
  movdqa    %xmm13, %xmm5
  pclmulqdq $0x00, 32(%rbp), %xmm5
  aesenc  80(%r15), %xmm8
  aesenc  80(%r15), %xmm9
  aesenc  80(%r15), %xmm10
  aesenc  80(%r15), %xmm11
  movdqa    %xmm13, %xmm6
  pclmulqdq $0x11, 32(%rbp), %xmm6
  aesenc  96(%r15), %xmm8
  aesenc  96(%r15), %xmm9
  aesenc  96(%r15), %xmm10
  aesenc  96(%r15), %xmm11
  movdqa    %xmm0, %xmm7
  pclmulqdq $0x00, 48(%rbp), %xmm7
  aesenc  112(%r15), %xmm8
  aesenc  112(%r15), %xmm9
  movdqa    %xmm0, %xmm12
  pclmulqdq $0x11, 48(%rbp), %xmm12
  aesenc  112(%r15), %xmm10
  aesenc  112(%r15), %xmm11


#holds xor of low products
  pxor  %xmm3, %xmm1
  pxor  %xmm7, %xmm5
  pxor  %xmm5, %xmm1

#holds xor of high products
  pxor  %xmm4, %xmm2
  pxor  %xmm12, %xmm6
  pxor  %xmm6, %xmm2

  pshufd  $78, %xmm15, %xmm3
  pshufd  $78, %xmm14, %xmm4
  pshufd  $78, %xmm13, %xmm5
  pshufd  $78, %xmm0, %xmm6

  pxor %xmm15, %xmm3
  pxor %xmm14, %xmm4
  pxor %xmm13, %xmm5
  pxor %xmm0, %xmm6

  aesenc  128(%r15), %xmm8
  aesenc  128(%r15), %xmm9
  aesenc  128(%r15), %xmm10
  aesenc  128(%r15), %xmm11

  movdqa    %xmm3, %xmm15
  pclmulqdq $0, 80(%rbp), %xmm15
  aesenc  144(%r15), %xmm8
  movdqa    %xmm4, %xmm14
  pclmulqdq $0, 96(%rbp), %xmm14
  aesenc  144(%r15), %xmm9
  movdqa    %xmm5, %xmm13
```

```
pclmulqdq  $0, 112(%rbp), %xmm13
aesenc  144(%r15), %xmm10
movdqa    %xmm6, %xmm0
pclmulqdq  $0, 128(%rbp), %xmm0
aesenc  144(%r15), %xmm11

pxor  %xmm15, %xmm14
pxor  %xmm13, %xmm0

pxor  %xmm1, %xmm14
pxor  %xmm2, %xmm0
pxor  %xmm14, %xmm0

movdqa    %xmm0, %xmm13
psrldq  $8, %xmm13
pslldq  $8, %xmm0

pxor  %xmm0, %xmm1
pxor  %xmm13, %xmm2

movdqa    %xmm1, %xmm4
movdqa    %xmm2, %xmm5
movdqa    %xmm1, %xmm3
movdqa    %xmm2, %xmm6

psrld  $31, %xmm4
psrld  $31, %xmm5
pslld  $1, %xmm3
pslld  $1, %xmm6

movdqa    %xmm4, %xmm1
psrldq  $12, %xmm1
pslldq  $4, %xmm4
pslldq  $4, %xmm5
por %xmm4, %xmm3
por %xmm5, %xmm6
por %xmm1, %xmm6

movdqa    %xmm3, %xmm4
movdqa    %xmm3, %xmm5
movdqa    %xmm3, %xmm1
pslld $31, %xmm4
pslld $30, %xmm5
pslld $25, %xmm1

pxor  %xmm5, %xmm4
pxor  %xmm1, %xmm4

movdqa    %xmm4, %xmm5
psrldq  $4, %xmm5
pslldq  $12, %xmm4
pxor  %xmm4, %xmm3

pxor  %xmm3, %xmm6
movdqa    %xmm3, %xmm1
movdqa    %xmm3, %xmm4
psrld  $1, %xmm1
psrld  $2, %xmm4
psrld  $7, %xmm3

pxor  %xmm6, %xmm4
pxor  %xmm3, %xmm1
pxor  %xmm4, %xmm1
```

```
  pxor   %xmm5, %xmm1

  movdqa  160(%r15), %xmm7


  cmpq   $11, %r14
  jb   MESSAGE_ENC_LAST
  cmpq   $13, %r14
  movdqa 192(%r15), %xmm7

  aesenc  160(%r15), %xmm8
  aesenc  160(%r15), %xmm9
  aesenc  160(%r15), %xmm10
  aesenc  160(%r15), %xmm11

  aesenc  176(%r15), %xmm8
  aesenc  176(%r15), %xmm9
  aesenc  176(%r15), %xmm10
  aesenc  176(%r15), %xmm11
  jb   MESSAGE_ENC_LAST
  movdqa  224(%r15), %xmm7

  aesenc  192(%r15), %xmm8
  aesenc  192(%r15), %xmm9
  aesenc  192(%r15), %xmm10
  aesenc  192(%r15), %xmm11

  aesenc  208(%r15), %xmm8
  aesenc  208(%r15), %xmm9
  aesenc  208(%r15), %xmm10
  aesenc  208(%r15), %xmm11
MESSAGE_ENC_LAST:
  aesenclast  %xmm7, %xmm8
  aesenclast  %xmm7, %xmm9
  aesenclast  %xmm7, %xmm10
  aesenclast  %xmm7, %xmm11

  pxor   (%rdi), %xmm8
  pxor   16(%rdi), %xmm9
  pxor   32(%rdi), %xmm10
  pxor   48(%rdi), %xmm11

  movdqa  %xmm8, (%rsi)
  movdqa  %xmm9, 16(%rsi)
  movdqa  %xmm10, 32(%rsi)
  movdqa  %xmm11, 48(%rsi)

  add $64, %rsi
  add $64, %rdi
  dec %r10
  jne  DATA_QUADS

FINAL_REDUCTION:

  pshufb  (BSWAP_MASK), %xmm8
  pshufb  (BSWAP_MASK), %xmm9
  pshufb  (BSWAP_MASK), %xmm10
  pshufb  (BSWAP_MASK), %xmm11

  pxor   %xmm1, %xmm8
```

```
  call   REDUCE_FOUR

DATA_SINGLES:

  movdqa   64(%rbp), %xmm0
  shrq $62, %r11
  je  DATA_REMAINDER

LOOP_DR:
  movq  144(%rsp), %r15
  movq  152(%rsp), %r14
  paddd (ONE), %xmm0
  movdqa    %xmm0, %xmm10
  pshufb  (BSWAP_MASK), %xmm10

  pxor  (%r15), %xmm10
LOOP_DR1:
  addq $16, %r15
  dec   %r14
  aesenc  (%r15), %xmm10
  jne   LOOP_DR1
  aesenclast  16(%r15), %xmm10
  pxor  (%rdi), %xmm10
  movdqa  %xmm10, (%rsi)
  addq  $16, %rsi
  addq  $16, %rdi
  dec   %r11
  pshufb  (BSWAP_MASK), %xmm10
  pxor  %xmm10, %xmm1
  call  GFMUL

  jne   LOOP_DR

DATA_REMAINDER:
  shrq  $60, %r12
  je  DATA_END

  movq  144(%rsp), %r15
  movq  152(%rsp), %r14
  paddd (ONE), %xmm0
  pshufb  (BSWAP_MASK), %xmm0
  pxor  (%r15), %xmm0
LOOP_DR2:
  addq $16, %r15
  dec   %r14
  aesenc  (%r15), %xmm0
  jne   LOOP_DR2
  aesenclast  16(%r15), %xmm0
  pxor  (%rdi), %xmm0
  movdqa  %xmm0, (%rsi)

  movq  (%rsi), %rax
  movq  8(%rsi), %rbx
  cmp   $8, %r12
  jl  DATA_LESS_THEN_8
  jg  DATA_MORE_THEN_8
  xorq  %rbx, %rbx
  jmp   DATA_REMAINDER_END
DATA_LESS_THEN_8:
  movq  $8, %rcx
  subq  %r12, %rcx
  shlq  $3, %rcx
  shlq  %cl, %rax
```

```
   shrq  %cl, %rax
   xorq  %rbx, %rbx
   jmp   DATA_REMAINDER_END
DATA_MORE_THEN_8:
   movq  $16, %rcx
   subq  %r12, %rcx
   shlq  $3, %rcx
   shlq  %cl, %rbx
   shrq  %cl, %rbx
DATA_REMAINDER_END:
   pinsrq  $0, %rax, %xmm3
   pinsrq  $1, %rbx, %xmm3

   pshufb  (BSWAP_MASK), %xmm3
   pxor  %xmm3, %xmm1
   call  GFMUL
DATA_END:

   movq  128(%rsp), %r10
   shlq  $3, %r9
   shlq  $3, %r10

   pinsrq  $0, %r9, %xmm3
   pinsrq  $1, %r10, %xmm3

   pxor   %xmm3, %xmm1
   movdqu  (%rbp), %xmm2
   movdqu  (%r8), %xmm10
   call   GFMUL
   pshufb  (BSWAP_MASK), %xmm1
   pxor  %xmm1, %xmm10
   movdqu  %xmm10, (%r8)

END:
   sub  $0, %rsp
   popq %rdi
   popq %rsi
   popq %rdx
   popq %rcx
   popq %rbx
   popq  %rax
   popq  %r15
   popq  %r14
   popq  %r13
   popq  %r12
   popq  %r11
   popq  %r10
   popq  %r9
   popq  %r8
   popq  %rbp
   ret
###########################
PROCESS_96BIT_IVEC:
   movdqu  (%rcx), %xmm0
# xmm0 will hold the Y
   pinsrd  $3, (INS), %xmm0

   movq  144(%rsp), %r15
# key schedule
   movq  152(%rsp), %r12
```

```
# number of rounds
  dec    %r12
  movq   %r12, 152(%rsp)
  movdqa  (%r15), %xmm1
# in xmm1 zero block is encrypted
  movdqa  %xmm0, %xmm2
  pxor   (%r15), %xmm2
# in xmm2 Y is encrypted

LOOP_PI1:        #parallelize aes encryption
 addq  $16, %r15
 dec   %r12
 aesenc  (%r15), %xmm1
 aesenc  (%r15), %xmm2
 jne   LOOP_PI1

  aesenclast  16(%r15), %xmm1
  aesenclast  16(%r15), %xmm2

  pshufb  (BSWAP_MASK), %xmm1
# swap bytes
  movdqu  %xmm2, (%r8)
# store T at *tag
  movdqu  %xmm1, (%rbp)
# store H on stack

  jmp    CALCULATE_POWERS_OF_H
############################
PROCESS_NON96BIT_IVEC:
  movq  144(%rsp), %r15
# key schedule
  movq  152(%rsp), %r12
# number of rounds
  movq  152(%rsp), %r12
# number of rounds
  dec    %r12
  movq   %r12, 152(%rsp)

  movdqa  (%r15), %xmm2
# in xmm2 zero block is encrypted

LOOP_PNI1:
 addq  $16, %r15
 dec   %r12
 aesenc  (%r15), %xmm2
 jne   LOOP_PNI1

  aesenclast  16(%r15), %xmm2
  pshufb  (BSWAP_MASK), %xmm2
# swap bytes
  movdqu  %xmm2, (%rbp)
# store H on stack
  pxor   %xmm1, %xmm1
# Y is zero at first

  movq  136(%rsp), %r10    # ibytes
  movq  %r10, %r11
  movq  %r10, %r12
  xorq  %r13, %r13
  shlq  $3, %r12
  shlq  $60, %r11
  shrq  $4, %r10
  je  PNI_REMAINDER
```

```
LOOP_PNI2:      # hash ivec
 movdqu  (%rcx), %xmm3
 add    $16, %rcx
 dec    %r10
 pshufb  (BSWAP_MASK), %xmm3
 pxor  %xmm3, %xmm1
 call  GFMUL
 jne   LOOP_PNI2

PNI_REMAINDER:      # hash ivec remainder
  shrq  $60, %r11
  je  PNI_END

  movq  (%rcx), %rax
  movq  8(%rcx), %rbx

  cmp    $8, %r11
  jl   PNI_LESS_THEN_8
  jg   PNI_MORE_THEN_8
  xorq   %rbx, %rbx
  jmp    PNI_REMAINDER_END
PNI_LESS_THEN_8:
  movq  $8, %rcx
  subq  %r11, %rcx
  shlq  $3, %rcx
  shlq  %cl, %rax
  shrq  %cl, %rax
  xorq  %rbx, %rbx
  jmp    PNI_REMAINDER_END
PNI_MORE_THEN_8:
  movq  $16, %rcx
  subq  %r11, %rcx
  shlq  $3, %rcx
  shlq  %cl, %rbx
  shrq  %cl, %rbx
PNI_REMAINDER_END:
  pinsrq  $0, %rax, %xmm3
  pinsrq  $1, %rbx, %xmm3
  pshufb  (BSWAP_MASK), %xmm3
  pxor  %xmm3, %xmm1
  call  GFMUL

PNI_END:
  pinsrq  $0, %r12, %xmm3
  pinsrq  $1, %r13, %xmm3
  pxor  %xmm3, %xmm1
  call  GFMUL
  pshufb  (BSWAP_MASK), %xmm1
  movdqa  %xmm1, %xmm0

  movq  144(%rsp), %r15
# key schedule
  movq  152(%rsp), %r12
# number of rounds
  pxor  (%r15), %xmm1

LOOP_PNI3:
 addq  $16, %r15
 dec    %r12
```

```
  aesenc  (%r15), %xmm1
  jne    LOOP_PNI3

  aesenclast  16(%r15), %xmm1
  movdqu  %xmm1, (%r8)
# store T at *tag

  jmp    CALCULATE_POWERS_OF_H
#########################
.align 16
REDUCE_FOUR:
  movdqa     %xmm11, %xmm1
  movdqa     %xmm11, %xmm2
  pclmulqdq $0x00, (%rbp), %xmm1
  pclmulqdq $0x11, (%rbp), %xmm2

  movdqa     %xmm10, %xmm3
  movdqa     %xmm10, %xmm4
  pclmulqdq $0x00, 16(%rbp), %xmm3
  pclmulqdq $0x11, 16(%rbp), %xmm4

  movdqa     %xmm9, %xmm5
  movdqa     %xmm9, %xmm6
  pclmulqdq $0x00, 32(%rbp), %xmm5
  pclmulqdq $0x11, 32(%rbp), %xmm6

  movdqa     %xmm8, %xmm7
  movdqa     %xmm8, %xmm12
  pclmulqdq $0x00, 48(%rbp), %xmm7
  pclmulqdq $0x11, 48(%rbp), %xmm12

#holds xor of low products
  pxor  %xmm3, %xmm1
  pxor  %xmm7, %xmm5
  pxor  %xmm5, %xmm1

#holds xor of high products
  pxor  %xmm4, %xmm2
  pxor  %xmm12, %xmm6
  pxor  %xmm6, %xmm2

  pshufd  $78, %xmm11, %xmm3
  pshufd  $78, %xmm10, %xmm4
  pshufd  $78, %xmm9, %xmm5
  pshufd  $78, %xmm8, %xmm6

  pxor %xmm11, %xmm3
  pxor %xmm10, %xmm4
  pxor %xmm9, %xmm5
  pxor %xmm8, %xmm6

  movdqa     %xmm3, %xmm11
  pclmulqdq $0, 80(%rbp), %xmm11
  movdqa     %xmm4, %xmm10
  pclmulqdq $0, 96(%rbp), %xmm10
  movdqa     %xmm5, %xmm9
  pclmulqdq $0, 112(%rbp), %xmm9
  movdqa     %xmm6, %xmm8
  pclmulqdq $0, 128(%rbp), %xmm8

  pxor  %xmm11, %xmm10
  pxor  %xmm9, %xmm8
```

```
  pxor   %xmm1, %xmm10
  pxor   %xmm2, %xmm8
  pxor   %xmm10, %xmm8

  movdqa    %xmm8, %xmm9
  psrldq  $8, %xmm9
  pslldq  $8, %xmm8

  pxor   %xmm8, %xmm1
  pxor   %xmm9, %xmm2

  movdqa    %xmm1, %xmm4
  movdqa    %xmm2, %xmm5
  psrld  $31, %xmm4
  psrld  $31, %xmm5
  movdqa    %xmm1, %xmm3
  movdqa    %xmm2, %xmm6
  pslld  $1, %xmm3
  pslld  $1, %xmm6

  movdqa    %xmm4, %xmm1
  psrldq  $12, %xmm1
  pslldq  $4, %xmm4
  pslldq  $4, %xmm5

  por %xmm4, %xmm3
  por %xmm5, %xmm6
  por %xmm1, %xmm6

  movdqa    %xmm3, %xmm4
  movdqa    %xmm3, %xmm5
  movdqa    %xmm3, %xmm1
  pslld $31, %xmm4
  pslld $30, %xmm5
  pslld $25, %xmm1

  pxor   %xmm5, %xmm4
  pxor   %xmm1, %xmm4

  movdqa    %xmm4, %xmm5
  psrldq  $4, %xmm5
  pslldq  $12, %xmm4
  pxor   %xmm4, %xmm3

  pxor   %xmm3, %xmm6
  movdqa    %xmm3, %xmm1
  movdqa    %xmm3, %xmm4
  psrld  $1, %xmm1
  psrld  $2, %xmm4
  psrld  $7, %xmm3
  pxor   %xmm6, %xmm4
  pxor   %xmm3, %xmm1
  pxor   %xmm4, %xmm1
  pxor   %xmm5, %xmm1

  ret
#########################
# a = xmm1
# b = xmm2 - remains unchanged
# res = xmm1
```

```
# uses also xmm3,xmm4,xmm5,xmm6
GFMUL:
  movdqa     %xmm1, %xmm3
  movdqa     %xmm1, %xmm6
  pclmulqdq $0x00, %xmm2, %xmm3
  pclmulqdq $0x11, %xmm2, %xmm6

  pshufd $78, %xmm1, %xmm4
  pshufd $78, %xmm2, %xmm5
  pxor    %xmm1, %xmm4
  pxor    %xmm2, %xmm5
  ##
  pclmulqdq $0x00, %xmm5, %xmm4
  pxor    %xmm3, %xmm4
  pxor    %xmm6, %xmm4

  movdqa     %xmm4, %xmm5
  pslldq $8, %xmm5
  psrldq $8, %xmm4

  pxor    %xmm5, %xmm3
  pxor    %xmm4, %xmm6

  movdqa     %xmm3, %xmm4
  movdqa     %xmm6, %xmm5
  psrld  $31, %xmm4
  psrld  $31, %xmm5
  pslld  $1, %xmm3
  pslld  $1, %xmm6

  movdqa     %xmm4, %xmm1
  psrldq $12, %xmm1
  pslldq $4, %xmm4
  pslldq $4, %xmm5

  por %xmm4, %xmm3
  por %xmm5, %xmm6
  por %xmm1, %xmm6

  movdqa     %xmm3, %xmm4
  movdqa     %xmm3, %xmm5
  movdqa     %xmm3, %xmm1
  pslld $31, %xmm4
  pslld $30, %xmm5
  pslld $25, %xmm1

  pxor   %xmm5, %xmm4
  pxor   %xmm1, %xmm4

  movdqa     %xmm4, %xmm5
  psrldq  $4, %xmm5
  pslldq  $12, %xmm4
  pxor   %xmm4, %xmm3

  pxor   %xmm3, %xmm6
  movdqa     %xmm3, %xmm1
  movdqa     %xmm3, %xmm4
  psrld  $1, %xmm1
  psrld  $2, %xmm4
  psrld  $7, %xmm3
  pxor   %xmm6, %xmm4
  pxor   %xmm3, %xmm1
  pxor   %xmm4, %xmm1
```

```
pxor    %xmm5, %xmm1

ret
```

Code Outputs

Figure 16. Test Vector 1: Code Output

```
CPU check passed. AES instructions are supported.

The Key:                          [0x00000000000000000000000000000000]

The IV:                           [0x000000000000000000000000]

NONE

NONE

NONE

The tag:                          [0x58e2fccefa7e3061367f1d57a4e7455a]

The tag is equal to the expected tag.
The cipher text is equal to the expected cipher text.
Decryption succeeded.
The decrypted text is equal to the original plaintext.
```

Figure 17. Test Vector 2: Code Output

```
CPU check passed. AES instructions are supported.

The Key:                          [0x00000000000000000000000000000000]

The IV:                           [0x000000000000000000000000]

NONE

The PLAINTEXT:                    [0x00000000000000000000000000000000]

The CIPHERTEXT:                   [0x0388dace60b6a392f328c2b971b2fe78]

The tag:                          [0xab6e47d42cec13bdf53a67b21257bddf]

The tag is equal to the expected tag.
The cipher text is equal to the expected cipher text.
Decryption succeeded.
The decrypted text is equal to the original plaintext.
```

Figure 18. Test Vector 3: Code Output

```
CPU check passed. AES instructions are supported.

The Key:                            [0xfeffe9928665731c6d6a8f9467308308]

The IV:                             [0xcafebabefacedbaddecaf888]

NONE

The PLAINTEXT:                      [0xd9313225f88406e5a55909c5aff5269a]
The PLAINTEXT:                      [0x86a7a9531534f7da2e4c303d8a318a72]
The PLAINTEXT:                      [0x1c3c0c95956809532fcf0e2449a6b525]
The PLAINTEXT:                      [0xb16aedf5aa0de657ba637b391aafd255]

The CIPHERTEXT:                     [0x42831ec2217774244b7221b784d0d49c]
The CIPHERTEXT:                     [0xe3aa212f2c02a4e035c17e2329aca12e]
The CIPHERTEXT:                     [0x21d514b25466931c7d8f6a5aac84aa05]
The CIPHERTEXT:                     [0x1ba30b396a0aac973d58e091473f5985]

The tag:                            [0x4d5c2af327cd64a62cf35abd2ba6fab4]

The tag is equal to the expected tag.
The cipher text is equal to the expected cipher text.
Decryption succeeded.
The decrypted text is equal to the original plaintext.
```

Figure 19. Test Vector 4: Code Output

```
CPU check passed. AES instructions are supported.

The Key:                            [0xfeffe9928665731c6d6a8f9467308308]

The IV:                             [0xcafebabefacedbaddecaf888]

The header buffer:                  [0xfeedfacedeadbeeffeedfacedeadbeef]
The header buffer:                  [0xabaddad2]

The PLAINTEXT:                      [0xd9313225f88406e5a55909c5aff5269a]
The PLAINTEXT:                      [0x86a7a9531534f7da2e4c303d8a318a72]
The PLAINTEXT:                      [0x1c3c0c95956809532fcf0e2449a6b525]
The PLAINTEXT:                      [0xb16aedf5aa0de657ba637b39]

The CIPHERTEXT:                     [0x42831ec2217774244b7221b784d0d49c]
The CIPHERTEXT:                     [0xe3aa212f2c02a4e035c17e2329aca12e]
The CIPHERTEXT:                     [0x21d514b25466931c7d8f6a5aac84aa05]
The CIPHERTEXT:                     [0x1ba30b396a0aac973d58e091]

The tag:                            [0x5bc94fbc3221a5db94fae95ae7121a47]

The tag is equal to the expected tag.
The cipher text is equal to the expected cipher text.
Decryption succeeded.
The decrypted text is equal to the original plaintext.
```

Figure 20. Test Vector 5: Code Output

```
CPU check passed. AES instructions are supported.

The Key:                            [0xfeffe9928665731c6d6a8f9467308308]

The IV:                             [0xcafebabefacedbad]

The header buffer:                  [0xfeedfacedeadbeeffeedfacedeadbeef]
The header buffer:                  [0xabaddad2]

The PLAINTEXT:                      [0xd9313225f88406e5a55909c5aff5269a]
The PLAINTEXT:                      [0x86a7a9531534f7da2e4c303d8a318a72]
The PLAINTEXT:                      [0x1c3c0c95956809532fcf0e2449a6b525]
The PLAINTEXT:                      [0xb16aedf5aa0de657ba637b39]

The CIPHERTEXT:                     [0x61353b4c2806934a777ff51fa22a4755]
The CIPHERTEXT:                     [0x699b2a714fcdc6f83766e5f97b6c7423]
The CIPHERTEXT:                     [0x73806900e49f24b22b097544d4896b42]
The CIPHERTEXT:                     [0x4989b5e1ebac0f07c23f4598]

The tag:                            [0x3612d2e79e3b0785561be14aaca2fccb]

The tag is equal to the expected tag.
The cipher text is equal to the expected cipher text.
Decryption succeeded.
The decrypted text is equal to the original plaintext.
```

Figure 21. Test Vector 6: Code Output

```
CPU check passed. AES instructions are supported.

The Key:                            [0xfeffe9928665731c6d6a8f9467308308]

The IV:                             [0x9313225df88406e555909c5aff5269aa]
The IV:                             [0x6a7a9538534f7da1e4c303d2a318a728]
The IV:                             [0xc3c0c95156809539fcf0e2429a6b5254]
The IV:                             [0x16aedbf5a0de6a57a637b39b]

The header buffer:                  [0xfeedfacedeadbeeffeedfacedeadbeef]
The header buffer:                  [0xabaddad2]

The PLAINTEXT:                      [0xd9313225f88406e5a55909c5aff5269a]
The PLAINTEXT:                      [0x86a7a9531534f7da2e4c303d8a318a72]
The PLAINTEXT:                      [0x1c3c0c95956809532fcf0e2449a6b525]
The PLAINTEXT:                      [0xb16aedf5aa0de657ba637b39]

The CIPHERTEXT:                     [0x8ce24998625615b603a033aca13fb894]
The CIPHERTEXT:                     [0xbe9112a5c3a211a8ba262a3cca7e2ca7]
The CIPHERTEXT:                     [0x01e4a9a4fba43c90ccdcb281d48c7c6f]
The CIPHERTEXT:                     [0xd62875d2aca417034c34aee5]

The tag:                            [0x619cc5aefffe0bfa462af43c1699d050]

The tag is equal to the expected tag.
The cipher text is equal to the expected cipher text.
Decryption succeeded.
The decrypted text is equal to the original plaintext.
```

PCLMULQDQ and GFMUL Test Vectors

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

PCLMULQDQ Test Vectors

The following first 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 = 7b5b54657374566563746f725d53475d
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


Test Vector 5:
xmm1 = 00000000000000008000000000000000
xmm2 = 00000000000000008000000000000000
immediate = 0x00
PCLMULQDQ result = 40000000000000000000000000000000
```

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

```
a =                0x7b5b54657374566563746f725d53475d
b =                0x48692853686179295b477565726f6e5d
GFMUL128 (a, b) =    0x40229a09a5ed12e7e4e10da323506d2
```

(GFMUL128 (a, b) is the multiplication results of a and b, 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
```

Explanation: For a 128 bits quantity X, Reflect (X) is defined as the 128 bits quantity Y whose i-th bit equal to the (127-i)-th bit of X, (for $0 \le 0 \le 127$).

If
```
A = 0x952b2a56a5604ac0b32b6656a05b40b6
B = 0xdfa6bf4ded81db03ffcaff95f830f061
```

And,

```
C = Reflect (A) * Reflect (B)
```

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

Then
```
Reflect (A) = 0x6d02da056a66d4cd035206a56a54d4a9
Reflect (B) = 0x860f0c1fa9ff53ffc0db81b7b2fd65fb
C = 0x65B7FC3340123F26DDAA34B50D7CA5B
```

Finally,

```
Reflect (C) = 0xda53eb0ad2c55bb64fc4802cc3feda60
```

Which is the "Multiplication Result" obtained above.

Performance

This chapter provides the performance numbers of the code examples which were given above, and, for comparison, the performance of code that does not use Intel AES New Instructions / PCLMULQDQ instruction (running on the same platform).

The experiments were carried out on a processor based on Intel microarchitecture codename Westmere running at 2.67 GHz. The system was run with Intel® Turbo Boost Technology, Intel® Hyper-Threading Technology, and Enhanced Intel Speedstep® Technology disabled, and no X server and no network daemon running. The operating system was Linux (OpenSuse 11.1 64 bits).

The C code parts were compiled using icc version 11.0 with the –O3 flag. For the AES-GCM, the key was pre-expanded, and the key expansion overhead was not counted. Table 1, below, provides performance numbers for three buffer sizes.

The performance results are summarized in Table 1. The top part of Table 1 shows several implementations of AES-GCM, using the algorithms which were provided in the paper. The bottom part of Table 1 shows the performance of code that does not use the PCLMULQDQ and the Intel AES New Instructions.

	Code Reference	Comments	Performance in CPU cycles per byte		
			Buffer Size: 1 KB	Buffer Size: 4 KB	Buffer Size: 16 KB
1	Fig. 5 + Fig. 9	AES-CTR parallelizing 4 blocks at a time; ghash operates on one block at a time. GFmul implemented using schoolbook and Algorithm 5.	5.49	5.36	5.33
2	Fig.7 + Fig. 11	AES-CTR parallelizing 4 blocks at a time; ghash operates on one block at a time. GFmul implemented using Karatsuba and Algorithm 4.	5.08	4.94	4.91
3	Fig. 8 + Fig. 12 +Fig. 5	AES-CTR parallelizing 4 blocks at a time; ghash operates on 4 blocks at a time. GFmul implemented using Schoolbook and Algorithm 5.	4.16	3.88	3.70
4	Fig. 15 + Fig. 6	ASM AES-CTR parallelizing 4 blocks and Ghashing 4 blocks at a time (assembly code)	3.85	3.60	3.54
	Performance of implementations that do not use PCLMULQDQ and AES instructions				
5	Fig. 2 + Fig.9 (Solaris)	AES-CTR parallelize 4 blocks, ghash one block at a time. GFmul using Solaris function. (*)	188.9	187.69	187.40
6	Gladman	Brian Gladman's GCM, taken from his website, using fastest method of 64K lookup tables (**)	22.69	22.27	21.96

Table 1. The Performance of AES128-GCM (on Intel Architecture Codename Westmere)

(*) The code was downloaded from http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/common/crypto/modes/gcm.c#46 (on September, 2009)

(**) The code was downloaded from http://gladman.plushost.co.uk/oldsite/AES/index.php (July 2009)

> **Note**: *Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should*

consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, Go to:
http://www.intel.com/performance/resources/benchmark_limitations.htm

Discussion

The significant speedup gained from using the PCLMULQDQ instruction and the Intel AES New Instructions, together with the proposed algorithms is evident.

For the cases where the GHASH is computed one block at a time, Table 1 shows that bit-reflecting the input it faster than the shift-by-one method (Algorithm 5). However, the picture changes when the GHASH is aggregated and computed only once per four (or more) blocks at a time. Then, the shift-by-one method allows for processing the input data (the ciphertext from the AES-CTR part of the algorithm) without bit-reflection, and the reduction modulo $x^{128} + x^7 + x^2 + x + 1$ occurs less frequently. In this case, the aggregated reduction method outperforms the other approaches, achieving performance of 3.54 Cycles/Byte (for long buffers). Note that in CTR mode the counter is encrypted (rather than the data). Therefore, the number of blocks which are processed in parallel, and the number of blocks which are aggregated before computing a reduction, is an implementation preference.

Table 1 also shows that performance improved with longer buffers. This is due to the fact the overhead for the initialization of the H value (s), and for the CTR mode is amortized over a larger chunk of data. For 4K buffer (and longer), the fastest implementation performs is more than 6 times faster than the existing implementations that use lookup tables.

Summary

This paper presented Intel's, new instruction, namely PCLMULQDQ, that performs carry-less multiplication of two 64-bit operands. We provided information on using PCLMULQDQ for computing the Galois Hash portion of the Galois Counter Mode. We also detailed several methods for using PCLMULQDQ and Intel AES New Instructions in order to implement AES-GCM. The reported performance numbers show significant speedup in AES-GCM, compared lookup tables based sowtware implementations.

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.

[4] 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.

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

[6] 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.

[7] 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, accepted for publication.

> Paper has already appeared as: S. Gueron and M. E. Kounavis, Efficient Implementation of the Galois Counter Mode Using a Carry-less Multiplier and a Fast Reduction Algorithm, Information Processing Letters 110:549–553 (2010)

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

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

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

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

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

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

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

[15] 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.

[16] 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/.

[17] S. Gueron, Intel's Advanced Encryption Standard (AES) New Instructions Set, available at: http://software.intel.com/en-us/articles/advanced-encryption-standard-aes-instructions-set/

[18] K. Jankowski and P. Laurent, *Packed AES-GCM algorithm suitable for AES / PCLMULQDQ instructions, manuscript*

Paper has already appeared as: K. Jankowski, P. Laurent: Packed AES-GCM Algorithm Suitable for AES/PCLMULQDQ Instructions, IEEE Transactions on Computers, 60:135-138 (2011).

Acknowledgements

About the Authors

Shay Gueron is an Intel Principal Engineer. He is the security architect of the CPU Architecture Department in the Mobility Group, at the Israel Development Center. His interests include applied security, cryptography, and algorithms. He holds a Ph.D. degree in applied mathematics from Technion—Israel Institute of Technology. He is also an Associate Professor at the Department of Mathematics of the Faculty of Science at the University of Haifa, in 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. Michael has co-invented the CRC32, PCLMULQDQ instruction, and the Intel AES New Instructions for the Intel® microarchitecture codename Nehalem and Intel® microarchitecture codename Westmere, and was a co-receipient of an Intel Achievement award for the AES work, in 2008. Prior to joining Intel, he worked on programming network architectures at Columbia University, where he also received a Ph.D degree in 2004.