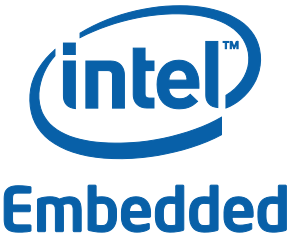


White Paper

**Vinodh Gopal**
**Erdinc Ozturk**
**Wajdi Feghali**
**Jim Guilford**
**Gil Wolrich**
**Martin Dixon**
**IA Architects**

Intel Corporation

# Optimized Galois-Counter-Mode Implementation on Intel® Architecture Processors

August 2010

324194

# Executive Summary

Galois-Counter-Mode (GCM) is a block cipher mode of operation that uses universal hashing over a binary Galois field to provide authenticated encryption. Galois Hash is used for authentication, and the Advanced Encryption Standard (AES) block cipher is used for encryption in counter mode of operation. This paper describes an optimized implementation of GCM benefiting from the PCLMULQDQ instruction and AES-NI set of instructions on Intel® processors based on the 32-nm microarchitecture.

This paper describes an optimized implementation of GCM that combines function stitching with novel polynomial multiplication methods. We are able to achieve performance of ~ **2.8 Cycles / byte** on large buffers, on a single core of an Intel® Core™ i5 650 processor, with Intel® Hyper-Threading Technology. This represents a new record for GCM performance on Intel® processors.

# Contents

# Figures

# Overview

GHASH ("Galois Hash") is used for high performance message authentication, usually in conjunction with AES encryption in Galois counter mode. AES-GCM is an increasingly prominent mode, used for packet processing in applications such as fast networking. AES-GCM mode is defined in the FIPS 800-38D and the IEEE 802.1ae standards and is recommended for high-speed networking. AES-GCM performance can be significantly increased by using the PCLMULQDQ instruction and AES set of instructions in Intel® processors based on the 32-nm microarchitecture. In this paper, we describe new methods to implement GCM for optimized performance using function stitching and novel polynomial multiplication methods and the corresponding performance results. The GCM mode we implement conforms to the IPsec Encapsulating Security Payload (ESP) protocol in tunnel mode as described in RFC 4106 [5]. Though we do not include the code of our implementation in the paper, we describe it a high-level.

For simplicity we have explained the encryption process in this paper, however the decryption approach and performance are almost identical. We assume the reader has familiarity with cryptographic algorithms related to block ciphers and authentication, specifically with GHASH methods.

# GCM

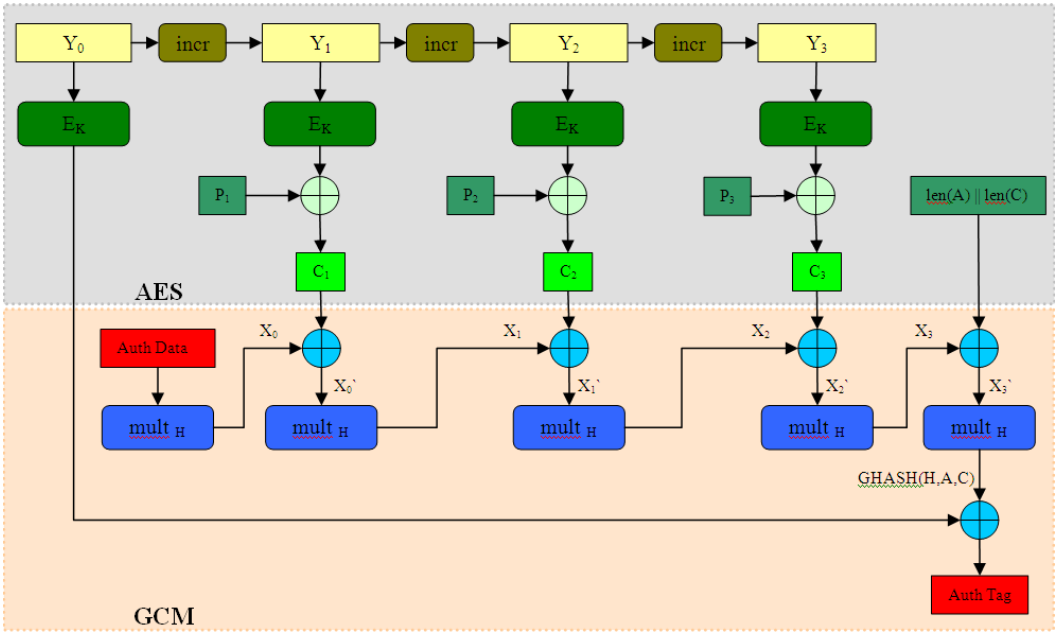
The AES encryption is performed using the counter mode of operation with 128-bit keys. The GHASH hashing is performed on the ciphertext generated by the AES encryption and a 128-bit digest, and generates an updated digest in each step.

In each step of GHASH, the previous GHASH digest is added with an XOR operation (⊕), to the current ciphertext block. The result is then multiplied in the Galois Field GF($2^{128}$) with a hash key value H (which is constant during the session). Multiplication in the Galois Field GF($2^{128}$) consists of the following two steps:

- 128x128 carry-less multiplication resulting in an intermediate 256 bit product
- Reduction modulo the GHASH irreducible pentanomial $g(x) = x^{128} + x^7 + x^2 + x + 1$.

Figure 1 demonstrates the AES-GCM encryption operation for a minimum IPsec packet in tunnel mode (48 byte min sized buffer):

**Figure 1: AES-GCM Encryption Operation for 48 Byte Min Sized Buffer**




In Figure 1, the top portion shows the computations associated with the AES Encryption and the bottom portion shows the GHASH computations. The inputs to the operation consist of the Keys, Plaintext, initialization data, the additional authentication data (AAD) and lengths. The outputs are the ciphertext and the authentication tag.

The initialization data comprises of a salt from the security association and sequence number from the IPsec Header, concatenated with an initial counter value of 1, to form the value $Y_0$. Subsequent values $Y_i$ are generated by incrementing the counter, shown by the “incr” block. The ciphertext blocks are generated by performing $C_i = E_K(Y_i) \oplus P_i$. The very first counter is encrypted to use as a mask on the final tag.

The AAD is a single padded block that is derived from a sequence number. We perform a first GHASH operation using the AAD to compute the initial digest $X_0$. The GHASH operation is denoted by the “mult H” block. The subsequent GHASH operations are performed on the 3 ciphertext blocks generated by the AES counter mode encryption process. A final GHASH operation is performed on a block generated with the length data, XOR’d with an encrypted mask to produce the final digest. The Authentication Tag is the final digest or a subset of the bytes, depending on the parameters of the protocol. For further details, refer to the RFC 4106 [5]. For larger buffers, the same process is followed, except that there will be more blocks than the 3 blocks shown in this example.

# Our Approach

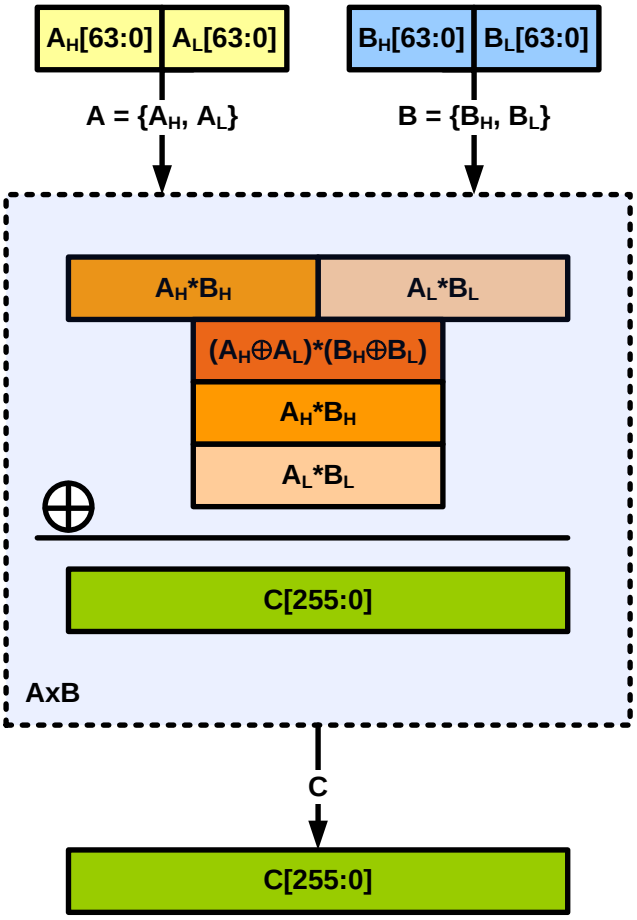
## AES Encryption

We perform AES128 computations efficiently processing 4 blocks in parallel in the counter mode of operation using the AES instructions. This approach and performance characteristics have been described in References [4].

## GHASH

The first fundamental component in GHASH computations is the 128x128 polynomial multiplication for a single block. To realize this multiplication, we use the Karatsuba algorithm as shown in Figure 2. The individual polynomial multiplications in the figure are 64x64 multiplications, which can be computed with the PCLMULQDQ instruction. The specifics of Karatsuba multiplication using PCLMULQDQ instruction, are described in subsequent sections.

**Figure 2: Karatsuba Multiplication method for two 128-bit polynomials for GHASH computations**




To compute the GHASH digest of 4 consecutive blocks, we use a method of parallelization as described in References [3]. The method can be described by the following equations:

**Ciphertext inputs**: $C_0$, $C_1$, $C_2$, $C_3$.

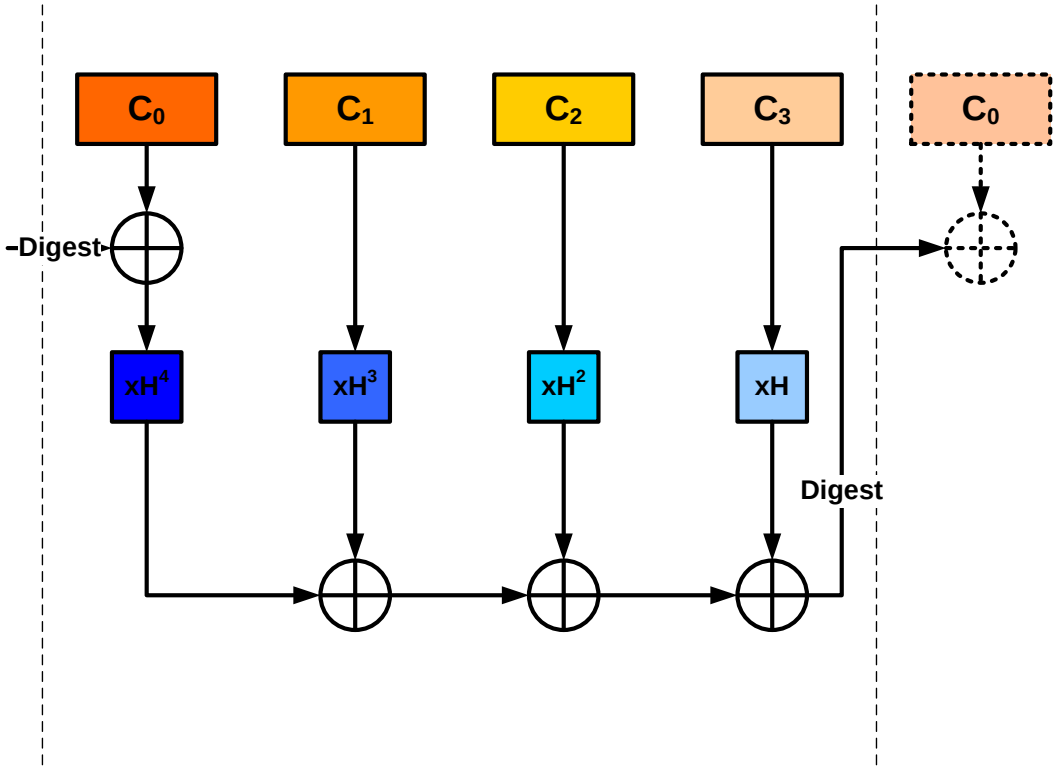
**Digest input/output**: Digest.

$$\text{Digest} = (((((((\text{Digest} \oplus C_0)*H) \oplus C_1)*H) \oplus C_2)*H) \oplus C_3)*H$$

$$= ((\text{Digest} \oplus C_0)*H^4) \oplus (C_1*H^3) \oplus (C_2*H^2) \oplus (C_3*H)$$

This resulting equation is illustrated in Figure 3.

**Figure 3: Parallelized GHASH computations on 4 blocks**




The constants $H^4$, $H^3$ and $H^2$ can be computed at the beginning of the GCM computation for a given data buffer and re-used for the rest of the blocks. Throughout our GCM implementation and particularly in the main loop, this approach is used to parallelize the GHASH computations.

For the final reduction of a 256-bit partial result modulo the GHASH polynomial, refer to References [3] “Efficient Reduction Algorithm” as we are using the same reduction method.

# Main Innovations in our approach

## Handling Bit-reflection of the multiplication product

The GCM standard References section [1] specifies that the data is bit-reflected for the GHASH operations. Reflecting the bits of all data and intermediate computations is expensive and there are better methods to handle the bit-reflection. Some of these methods and implementation details are explained in the paper [3], section “Bit Reflection Peculiarity of GCM”. We also avoid reflecting the bits of the operands before operating on them, similar to the approach in [3].

The operation:

reflected (A)*reflected (B) =reflected (A*B) >>1

is performed throughout the GHASH computations. Here, it should be noted that the result has to be shifted left by 1 in order to get the exact result: reflected (A*B).

**Our Improvement**

Since one of the operands of this multiplication operation is a constant (H) or precomputed as $H^2$, $H^3$ or $H^4$, we precompute corresponding "modified" constants so that they are "pre-shifted" and the result does not need any shifting operations. The challenge is that these constants are all 128-bit constants and could have their most-significant bits set, making it difficult to generate H<<1, $H^2$<<1, $H^3$ <<1 and $H^4$ <<1 as 128-bit constants. We could generate 129-bit constants, but this would add more overheads to the multiplication and reduction, offsetting potential gains from avoiding shifting the result. We therefore use the fact that we do **not** need the result of the multiplication to be exactly reflected (A*B) – we always compute a reduction of the result modulo the polynomial g(x). Thus if we precompute H<<1, $H^2$<<1, $H^3$ <<1 and $H^4$ <<1 modulo polynomial g(x), we can use them as 128-bit constants and avoid shifting the product, leading to the same result after the final reduction with g(x).

Thus, if we are multiplying A with H mod g(x):

reflected (A)*reflected (H<<1 mod g(x)) = reflected (A*H) mod g(x)

The multiplication operation will return the correct result after the final reduction with g(x). Note that for convenience, we do not show the shift operator explicitly in the rest of the paper, and describe the operations using logical constants such as $H^i$.

## Precomputing inputs to Karatsuba Algorithm

In addition to precomputing H<<1, $H^2$<<1, $H^3$ <<1 and $H^4$ <<1 mod g(x), we also precompute a few corresponding constants that are used in the Karatsuba algorithm.

Referring to Figure 2, during a Karatsuba Multiplication of Ah:Al and Bh:Bl, we need to perform 3 multiplication operations {Ah*Bh, (Ah ⊕ Al)*(Bh ⊕ Bl), Al*Bl}. The middle product term needs some computations to setup the operands. In the context of GHASH, the $2^{nd}$ operand B is always derived from the hashkey H. Thus we precompute (Bh ⊕ Bl) for the 4 derived hashkey values as well.

Thus we save significant amount of instructions in the main loop of the GHASH computations.

## Deferred Recombination of Karatsuba partial products

In the main loop that processes 4 blocks, we use the Karatsuba algorithm to perform multiplications of each block as explained in Figure 2. However, instead of combining the middle parts and having a complete result in 2 XMM registers for each block, we compute only the partial products and XOR them together to be combined at the end of the iteration once for all blocks.

If we name the high product in the Karatsuba multiplication of one block $H_i$, middle product $M_i$ and the low product $L_i$, we have the following equation:

$$M_i' = M_i \oplus H_i \oplus L_i$$

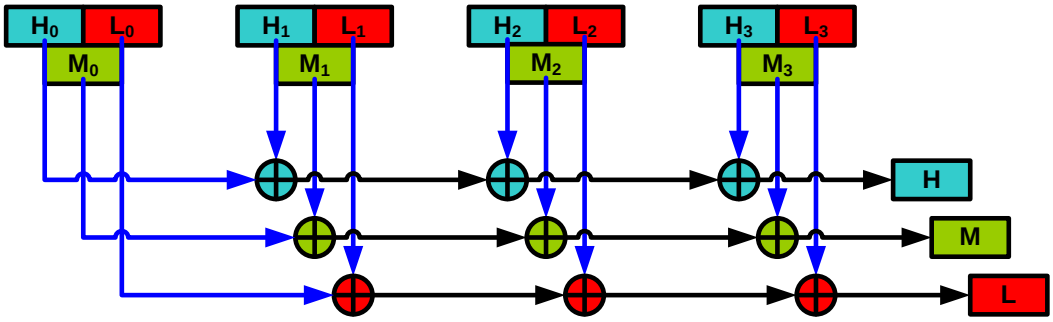
Instead of computing the middle term $M_i'$ for each multiplication, we do this combination just once at the end:

$$M' = M \oplus H \oplus L$$

as shown in Figure 4. The individual products $\{M_i, H_i, L_i\}$ can be computed using a code sequence such as:

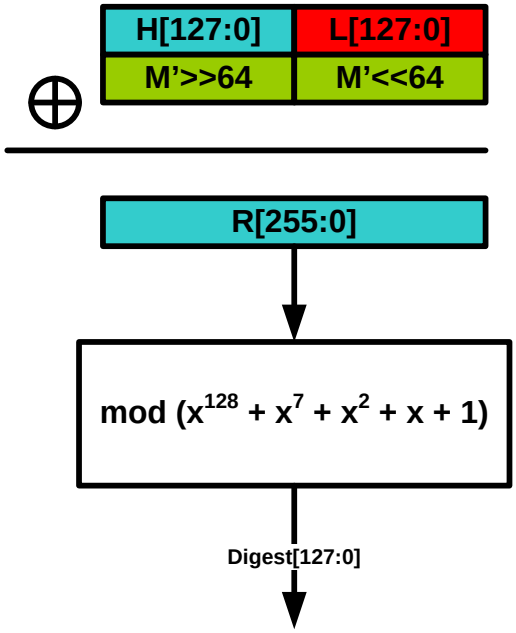
```
   ;; %%XMM6, %%T5 hold the two 128-bit operands (A, B) which are
carry-less multiplied. Polynomial multiplication by Karatsuba
Method:

   movdqa      %%T1, %%XMM6

   pshufd      %%T2, %%XMM6, 01001110b

   pxor        %%T2, %%XMM6 ; %%T2 = (a1+a0)

   movdqa      %%T5, [rsp + HashKey_3] ; B, based on Hash-key

   pclmulqdq   %%T1, %%T5, 0x11     ; %%T1 = a1*b1

   pclmulqdq   %%XMM6, %%T5, 0x00      ; %%XMM6 = a0*b0

   movdqa      %%T5, [rsp + HashKey_3_k] ; pre-computed value

   pclmulqdq   %%T2, %%T5, 0x00     ; %%T2 = (a1+a0)*(b1+b0)

   ;; %%T2, %%T1, %%XMM6 hold the values {Mi, Hi, Li}

   pxor        %%T4, %%T1           ; accumulate results

   pxor        %%XMM5, %%XMM6

   pxor        %%T6, %%T2

     ;; %%T6, %%T4, %%XMM5 hold the values {M, H, L} for 4 blocks
```

**Figure 4: Optimized Karatsuba recombination for the main loop**




Once we have the 3 intermediate results H, M' and L for the 4 blocks, we can combine them and apply the reduction as shown in Figure 5.

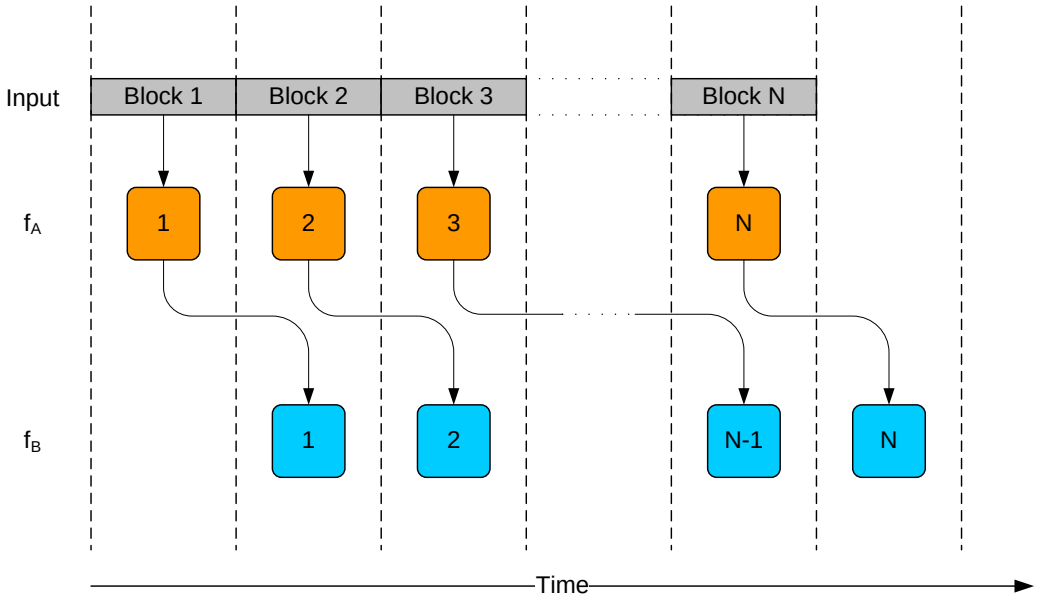
**Figure 5: Final combination and reduction in the main loop**




## Stitching AES with GHASH

We used the function stitching method proposed in References section [2]. The main idea is to interleave instructions from pairs of functions to maximize execution efficiency of the cores. For the purpose of this paper, we stitch the two main functions required for GCM computations {AES, GHASH} and refer to AES as function A and GHASH as function B. The benefits of stitching are widely explained in References section [2]. In the context of GCM encryption, the second stitching method is used as shown in Figure 6:

**Figure 6: Stitching Two Functions When One Operates on a Block of Data Produced by the Other**

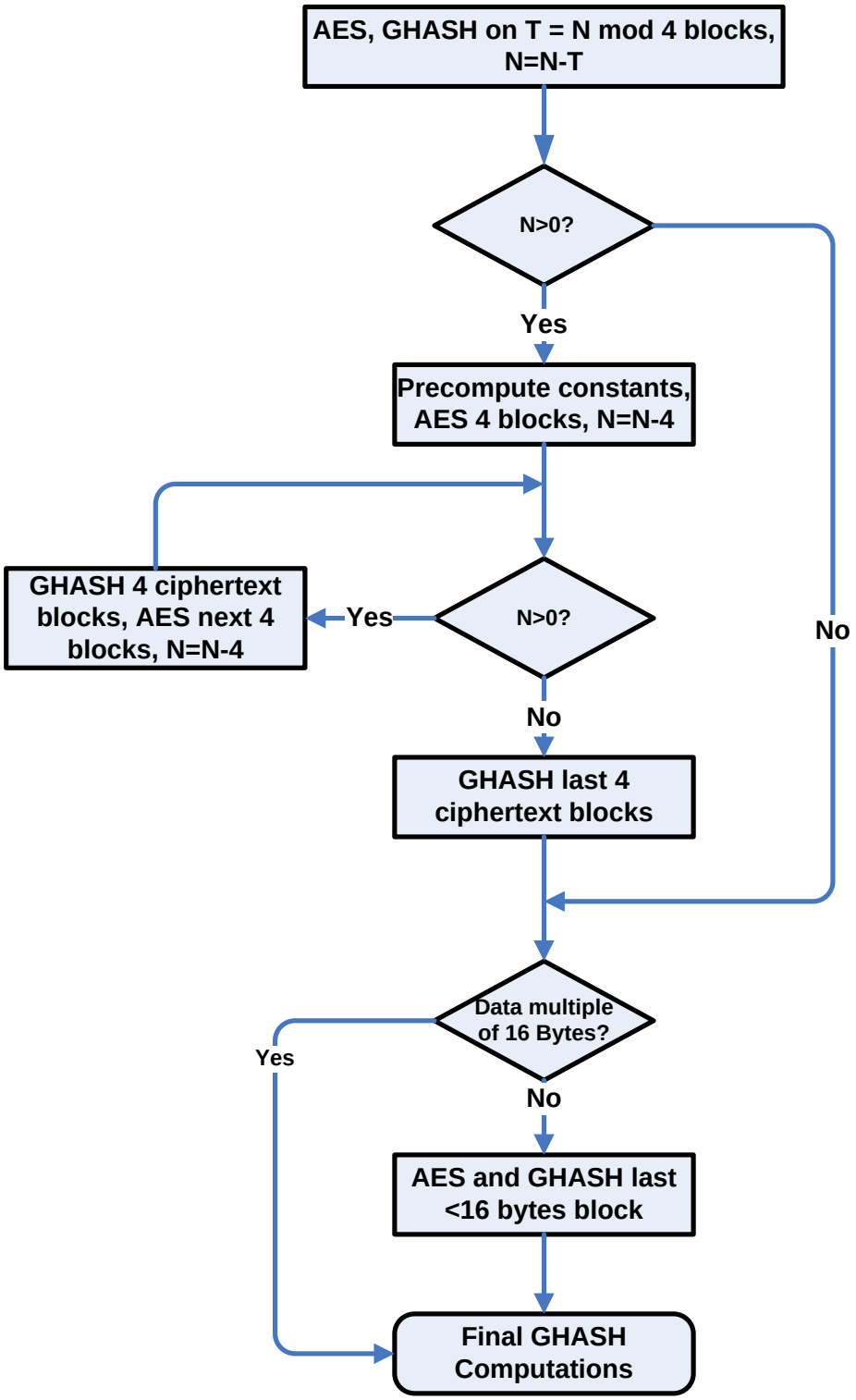



Here, GHASH (function B) works on the ciphertext data that has been produced by AES in the previous iteration and AES (function A) produces the ciphertext for the next iteration. Our GCM implementation works on 4 blocks in parallel, rather than the single block shown in Figure 6. Note that the column on the left corresponds to the loop prolog in our implementation that performs AES operations only, the column on the right corresponds to the loop epilog in our implementation that performs GHASH operations only, and the rest correspond to our main loop of iterations (for suitably large buffers). These concepts will be described in more detail in later sections.

## Overall Organization of the Implementation

The overall flow of our implementation is designed to handle different length buffers efficiently. It is described in detail in Figure 7. N denotes the number of whole blocks. The detailed description of the actual components are described in the section on Implementation Details.

**Figure 7: Overall Code Flow**



# Performance

The performance results provided in this section were measured on an Intel® Core™ i5 650 processor at a frequency of 3.20 GHz, supporting Intel® AES-NI. The tests were run with Intel® Turbo Boost Technology off, and represent the performance with Intel® Hyper-Threading Technology (Intel® HT Technology) on a single core. We present the results with Intel® Hyper-Threading Technology, since our target applications such as IPsec in Linux, are expected to run multiple threads for performance.

When a test is called, it is first run numerous times to **warm up the cache**. The timing is measured using the **rdtsc**() function which returns the processor time stamp counter (TSC). The TSC is the number of clock cycles since the last reset. The 'TSC_initial' is the TSC recorded before the function is called. Then, the function is called for the specified number of times. After the runs are complete, the **rdtsc**() is called again to record the new cycle count 'TSC_final'. The effective cycle count for the called routine is computed using

# of cycles = (TSC_final-TSC_initial)/(number of iterations).

When 2 such identical threads are run simultaneously, the number of cycles measured for the function represents the cycles consumed in the core for 2 data buffers (from each thread) and we calculate the cycles/byte accordingly. Thus, if each thread was computing GCM on buffers of size 1KB, then the net cycles/byte = # of cycles / 2*1KB

***Note:*** Performance results are based on certain tests measured on specific computer systems. Any difference in system hardware, software or configuration will affect actual performance. Configuration: Intel® Core™ i5 650 processor at a frequency of 3.20 GHz, supporting Intel® AES-NI. The tests were run with Intel® Turbo Boost Technology off, and represent the performance with Intel® Hyper-Threading Technology on a single core. For more information go to http://www.intel.com/performance.

**Figure 8: GCM Performance on a Single core with Intel® Hyper-Threading Technology**

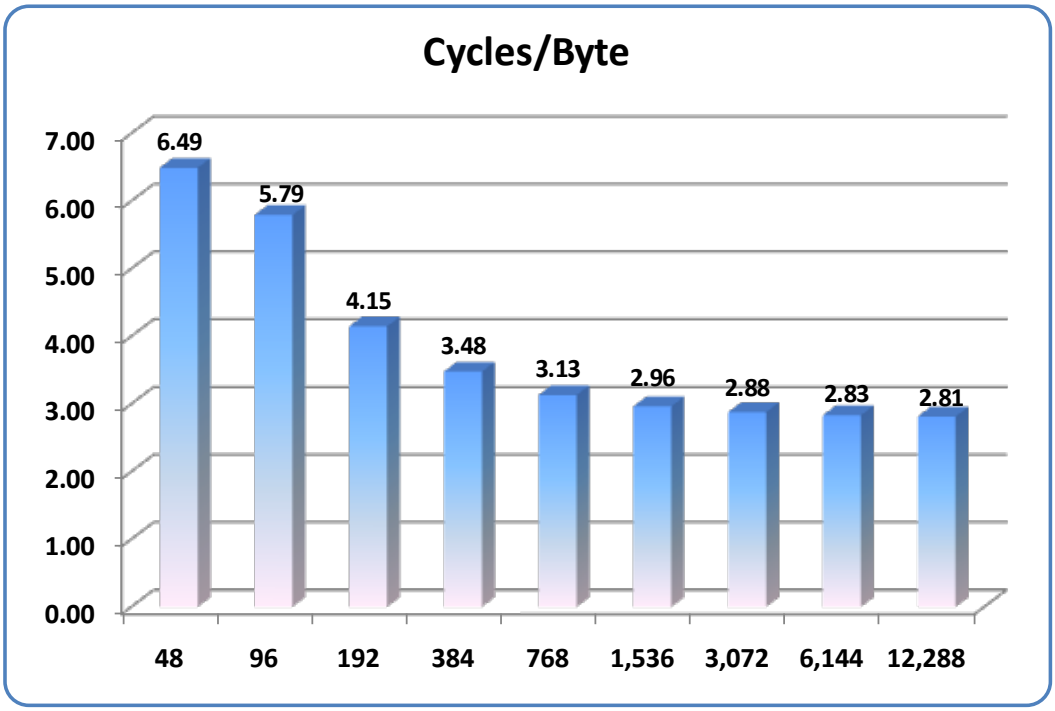



Figure 8 shows the performance in Cycles/byte as a function of the buffer size in bytes.

The performance for large buffers is ~ 2.8 cycles/byte and the performance for small buffers of 48 Bytes is ~6.4 cycles/byte. In terms of throughput, a **single core** can process GCM at the aggregate rate of ~**9 Gigabits/sec** for large buffers.

# Implementation Details

In our implementation, the code consists of 3 main sections: Initial computations, main loop and final computations. Since the main loop works on 4 blocks in each iteration, we compute the number of blocks modulo 4, and consume those many remainder blocks in the initial computations phase, to simplify the rest of the computations. We describe the various components of the implementation assuming that these are used when we process a suitably large data buffer for ease of illustration.

To simplify the explanations let us assume that:

Len = length of the buffer in bytes

N = number of whole blocks = (Len >>4)

T = N modulo 4

Y = number of bytes in partial block at the end of the buffer (Y<16)

## Initial Computations

First we take the length of the buffer in bytes and round it down to the closest multiple of 16, to find the number of whole blocks that we will apply GCM on (N). We then compute N modulo 4 (T) to determine how many remainder blocks to compute in the loop prolog. We handle any partial block (when Len is a non-multiple of 16 Bytes) separately at the end. In the prolog, we also precompute the various values that are derived from the hashkey if they are required in the main processing loop. We also compute the AES operations for the next 4 blocks if they are required at the start of the main loop.

Figure 9, Figure 10, Figure 11 and Figure 12 show the GHASH computations for T = 3, 2, 1 and 0 respectively.

**Figure 9: Number of blocks modulo 4 (T) = 3**

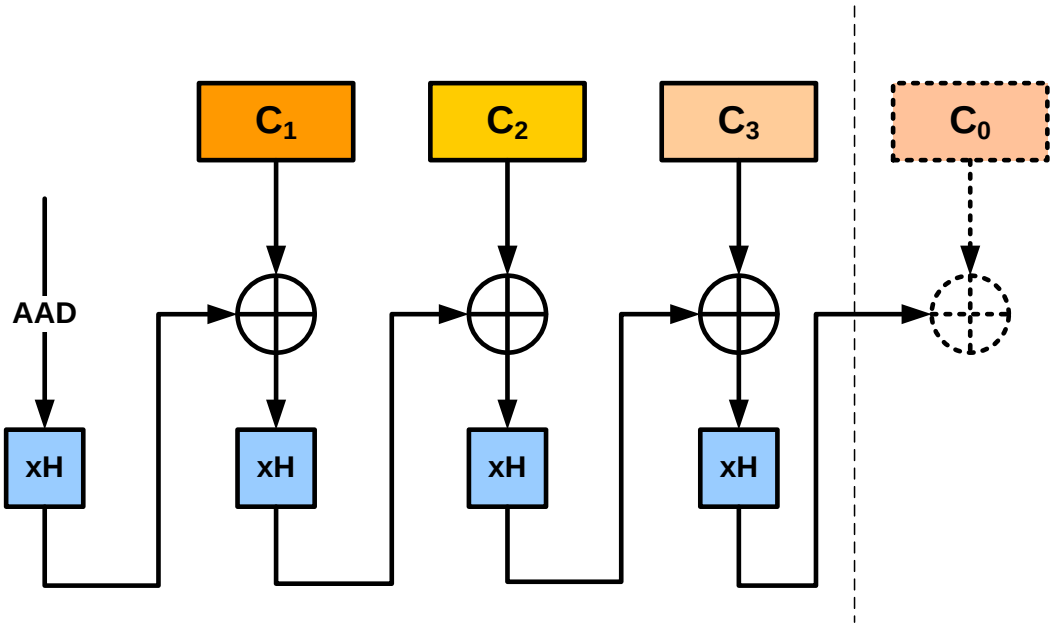


**Figure 10: Number of blocks modulo 4 (T) = 2**

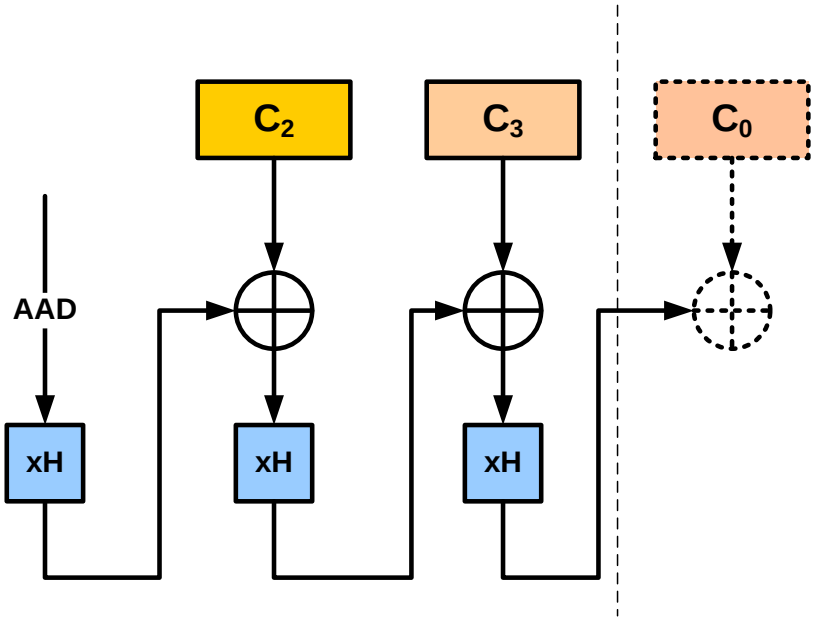
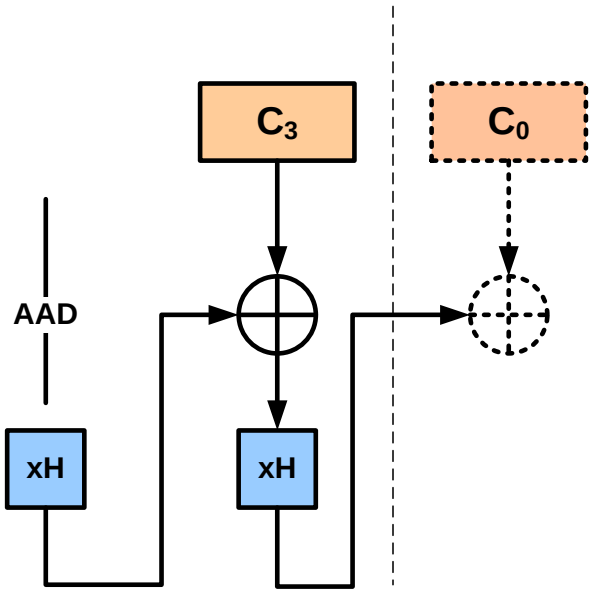



**Figure 11: Number of blocks modulo 4 (T) = 1**



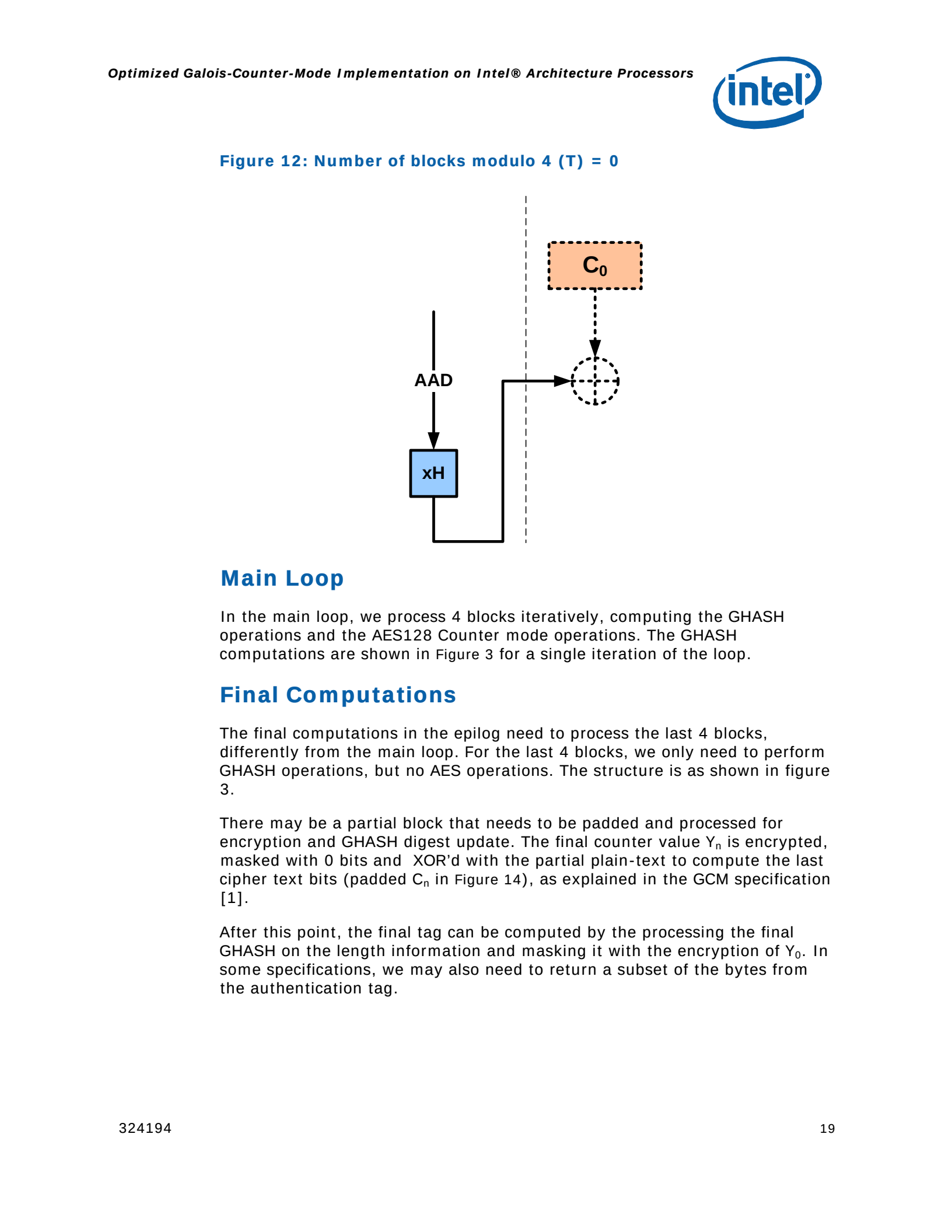


Figure 12: Number of blocks modulo 4 (T) = 0




## Main Loop

In the main loop, we process 4 blocks iteratively, computing the GHASH operations and the AES128 Counter mode operations. The GHASH computations are shown in Figure 3 for a single iteration of the loop.

## Final Computations

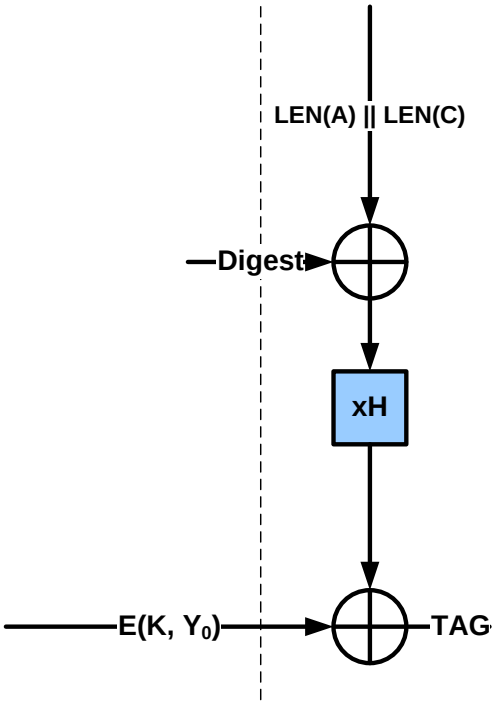
The final computations in the epilog need to process the last 4 blocks, differently from the main loop. For the last 4 blocks, we only need to perform GHASH operations, but no AES operations. The structure is as shown in figure 3.

There may be a partial block that needs to be padded and processed for encryption and GHASH digest update. The final counter value $Y_n$ is encrypted, masked with 0 bits and XOR'd with the partial plain-text to compute the last cipher text bits (padded $C_n$ in Figure 14), as explained in the GCM specification [1].

After this point, the final tag can be computed by the processing the final GHASH on the length information and masking it with the encryption of $Y_0$. In some specifications, we may also need to return a subset of the bytes from the authentication tag.

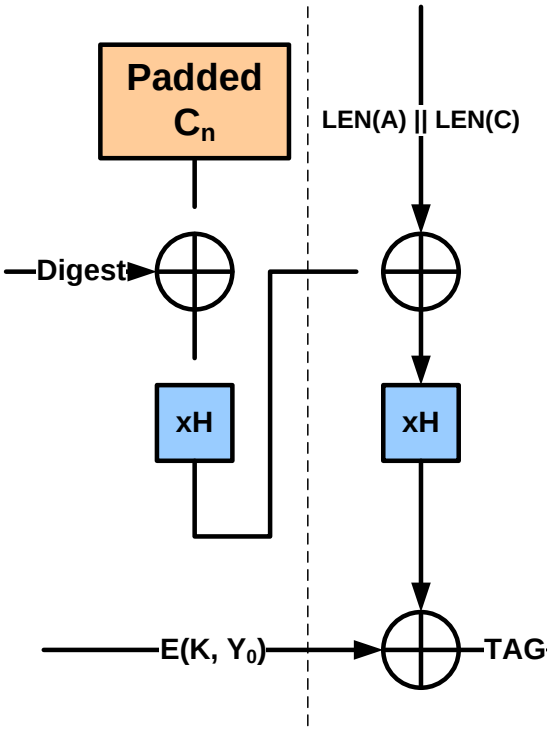
### Multiple of 16 Bytes

**Figure 13: Final combination for multiples of 16 Bytes**



### Non-Multiple of 16 Bytes

**Figure 14: Final computations for non-multiples of 16 Bytes**




# Conclusion

The paper describes the best-known implementation of AES-GCM on Intel® processors using the PCLMULQDQ instruction and AES set of instructions in Intel® processors based on the 32-nm microarchitecture. By combining function stitching with novel polynomial multiplication methods, we are able to achieve performance of ~ **2.8 Cycles / byte** on large buffers on a single core with Intel® Hyper-Threading Technology.

# References

[1] The Galois/Counter Mode of Operation (GCM)
http://www.cryptobarn.com/papers/gcm-spec.pdf

[2] Fast Cryptographic Computation on Intel® Architecture Processors Via Function Stitching
http://download.intel.com/design/intarch/PAPERS/323686.pdf

[3] Intel® Carry-less Multiplication Instruction and its Usage for Computing the GCM Mode http://software.intel.com/file/24918

[4] Breakthrough AES Performance with Intel® AES New Instructions
http://software.intel.com/file/26898

[5] The Use of Galois/Counter Mode (GCM) in IPsec Encapsulating Security Payload (ESP) http://tools.ietf.org/html/rfc4106



## Authors

**Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Wajdi Feghali** and **Martin Dixon** are IA Architects with the IAG Group at Intel Corporation.

## Acronyms

| | |
|---|---|
| IA | Intel® Architecture |
| ILP | Instruction level parallelism |
| SIMD | Single Instruction Multiple Data |
| SSE | Streaming SIMD Extensions |



§