White Paper

Intel Mobility Group,
Israel Development
Center, Israel

Shay Gueron

Intel® Advanced Encryption Standard (AES) Instructions Set

Intel® AES instructions are a new set of instructions available beginning with the all new 2010 Intel® Core™ processor family based on the 32nm Intel® microarchitecture codename Westmere. These instructions enable fast and secure data encryption and decryption, using the Advanced Encryption Standard (AES) which is defined by FIPS Publication number 197. Since AES is currently the dominant block cipher, and it is used in various protocols, the new instructions are valuable for a wide range of applications.

The architecture consists of six instructions that offer full hardware support for AES. Four instructions support the AES encryption and decryption, and other two instructions support the AES key expansion.

The AES instructions have the flexibility to support all usages of AES, including all standard key lengths, standard modes of operation, and even some nonstandard or future variants. They offer a significant increase in performance compared to the current pure-software implementations.

Beyond improving performance, the AES instructions provide important security benefits. By running in data-independent time and not using tables, they help in eliminating the major timing and cache-based attacks that threaten table-based software implementations of AES. In addition, they make AES simple to implement, with reduced code size, which helps reducing the risk of inadvertent introduction of security flaws, such as difficult-to-detect side channel leaks.

This paper gives an overview of the AES algorithm and Intel's new AES instructions. It provides guidelines and demonstrations for using these instructions to write secure and high performance AES implementations. This version of the paper also provides a high performance library for implementing AES in the ECB/CBC/CTR modes, and discloses for the first time, the measured performance numbers.

January 2010   (26/1/2010)

Rev. 3.0

Intel Corporation

Contents

Figures

Tables

Introduction

The Advanced Encryption Standard (AES) is the Federal Information Processing Standard for symmetric encryption, and it is defined by FIPS Publication #197 (2001). From the cryptographic perspective, AES is widely believed to be secure and efficient, and is therefore broadly accepted as the standard for both government and industry applications. In fact, almost any new protocol requiring symmetric encryption supports AES, and many existing systems that were originally designed with other symmetric encryption algorithms are being converted to AES. Given the popularity of AES and its expected long term importance, improving AES performance and security has significant benefits for the PC client and server platforms.

Intel is introducing a new set of instructions beginning with the all new 2010 Intel® Core™ processor family based on the 32nm Intel® microarchitecture codename Westmere. (In this document, Westmere refers to the microarchitecture and not to a particular processor product. AES new instructions may not be supported by all processors based on32nm Intel® microarchitecture codename Westmere).

The new architecture has six instructions: four instructions (AESENC, AESENCLAST, AESDEC, and AESDELAST) facilitate high performance AES encryption and decryption, and the other two (AESIMC and AESKEYGENASSIST) support the AES key expansion. Together, these instructions provide full hardware support for AES, offering high performance, enhanced security, and a great deal of software usage flexibility

The new AES instructions can support AES encryption and decryption with each one of the standard key lengths (128, 192, and 256 bits), using the standard block size of 128 bits (and potentially also other block sizes for generalized variants such as the RIJNDAEL algorithms). They are well suited to all common uses of AES, including bulk encryption/decryption using cipher modes such as ECB, CBC and CTR, data authentication using CBC-MACs (e.g., CMAC), random number generation using algorithms such as CTR-DRBG, and authenticated encryption using modes such as GCM. It is believed that these instructions will be useful for a wide range of cryptographic applications.

This paper provides an overview of the AES algorithm and guidelines for utilizing the AES instructions to achieve high performance and secure AES processing. Some special usage models of this architecture are also described. This version of the paper also provides a high performance library for implementing AES in the ECB/CBC/CTR modes of operation, and discloses, for the first time, the performance numbers for the provided code.

Preliminaries: AES and Intel® Architecture

AES Definition and Brief Description

The Advanced Encryption Standard (AES) is the United States Government's Federal Information Processing Standard for symmetric encryption, defined by FIPS Publication #197 (FIPS197 hereafter).

AES is a block cipher that encrypts a 128-bit block (plaintext) to a 128-bit block (ciphertext), or decrypts a 128-bit block (ciphertext) to a 128-bit block (plaintext).

AES uses a key (cipher key) whose length can be 128, 192, or 256 bits. Hereafter encryption/decryption with a cipher key of 128, 192, or 256 bits is denoted AES-128, AES192, AES-256, respectively.

AES-128, AES-192, and AES-256 process the data block in, respectively, 10, 12, or 14 iterations of pre-defined sequences of transformations, which are also called AES rounds ("rounds" for short). The rounds are identical except for the last one, which slightly differs from the others (by skipping one of the transformations).

The rounds operate on two 128-bit inputs: "State" and "Round key". Each round from 1 to 10/12/14 uses a different round key. The 10/12/14 round keys are derived from the cipher key by the "Key Expansion" algorithm. This algorithm is independent of the processed data, and can be therefore carried out independently of the encryption/decryption phase (typically, the key is expanded once and is thereafter used for many data blocks using some cipher mode of operation).

The data block is processed serially as follows: initially, the input data block is XOR-ed with the first 128 bits of the cipher key to generate the "State" (an intermediate cipher result). Subsequently, the State passes, serially, 10/12/14 rounds, each round consisting of a sequence of transformations operating on the State and using a different round key. The result of the last round is the encrypted (decrypted) block.

AES Text Convention in Intel® Architecture Terminology

FIPS197 defines AES in terms of bytes. However, the algorithm is described using a text convention where hexadecimal strings are written with the low-memory byte on the left, and the high-memory byte on the right (this convention is analogous to writing integers in a "Big Endian" convention). This text convention determines the way in which the test vectors are written, and the description of some of the algorithm's transformations. On the other hand, Intel® Architecture (IA) convention is the opposite: hexadecimal strings are written with the low-memory byte on the right and the high-memory byte on the left (this is analogous to writing integers in a "Little Endian" convention).

In either case, the low-memory byte is byte 0, the next is byte 1, and so forth. In the FIPS197 notation, when a 128-bit vector (string) is read from left to right, the bytes are read as [Byte0, Byte1, …, Byte14, Byte15], i.e., byte 0 (denoted "Byte0") is the leftmost one. In an IA notation, when a 128-bit vector is read from left to right, the bytes are read as [Byte15, Byte14, …, Byte1, Byte0], i.e., byte 0 is the rightmost one.

For encoding the bytes, each byte value can be viewed as an integer between 0 and 255, written in binary notation. Under this view, both the FIPS197 and Intel conventions use a Little Endian notation: the leftmost bit of the byte is the most significant bit (e.g., the byte 11000010 corresponds to the integer 194). The byte values are represented as 2-digit (two characters) numbers in hexadecimal notation. For example, the byte 11000010 (which corresponds to 194 in decimal notation) is represented as c2.

We point out that the store/load processor operations are consistent with the way that the AES instructions operate. In other words, the textual convention does not require programmers using the AES architecture to perform any byte reversal in their code,

Hereafter, we use Intel's IA convention to represent the AES State: 128-bit vectors ([127-0]) are split to bytes as [127-120, … 31-24, 23-16, 15-8, 7-0] and the bytes are written in a hexadecimal notation (as two characters).

Example

Consider the vector (State) d4bf5d30e0b452aeb84111f11e2798e5, written in the FIPS197 notation. It consists of 16 bytes, each one represented as 2-digit hexadecimal numbers, as follows: "d4 bf 5d 30 e0 b4 52 ae b8 41 11 f1 1e 27 98 e5". Here, d4 is byte 0.

The equivalent IA-compatible notation is e598271ef11141b8ae52b4e0305dbfd4 (also denoted in hexadecimal notation by 0xe598271ef11141b8ae52b4e0305dbfd4). This corresponds to the 16 bytes (2-digit hexadecimal numbers) "e5 98 27 1e f1 11 41 b8 ae 52 b4 e0 30 5d bf d4" (again, byte 0 is d4). The corresponding 128-bit encoding is:

```
1110010110011000001001110001111011110001000100010100000110111000
1010111001010010101101001110000000110000010111011011111111010100
```

(Where bit 127 equals 1 and bit 0 equals 0).

The AES State in Terms of IA Data Structure

Intel's AES instructions operate on one or two 128-bit inputs, and the typical instruction format is "instruction xmm1 xmm2/m128" (details are provided in the following text). Here, xmm1 and xmm2 are aliases to any two xmm registers, and the result is written into xmm1. The /m128 indicates a register-memory instruction.

When referring to the contents of an xmm register, one may refer to the bits (127-0), to the bytes (15-0), or to the 32-bit double words (3-0). Hereafter, 32-bit data chunks are referred to as "doublewords" to be consistent with the IA terminology (note however that the FIPS197 document calls 32-bits chunks "words"). The bytes are also referred to by the letters P-A and the doublewords are also referred to by X3-X0. Figure 1 illustrates the bit/byte/doublewords positions of the State in an xmm register.

Figure 1. State Bit, Byte, and Doubleword Positions in an xmm Register

127-120	119-112	111-104	103-96	95-88	87-80	79-72	71-64	63-56	55-48	47-40	39-32	31-24	23-16	15-8	7-0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
3 (127-96)				2 (95-64)				1 (63-32)				0 (31-0)			
X3				**X2**				**X1**				**X0**			
P	**O**	**N**	**M**	**L**	**K**	**J**	**I**	**H**	**G**	**F**	**E**	**D**	**C**	**B**	**A**

4x4 Matrix Notation of an xmm Register			
A	E	I	M
B	F	J	N
C	G	K	O
D	H	L	P

Example

The vector e598271ef11141b8ae52b4e0305dbfd4 is split to bytes as follows:

Byte #	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Letter	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A
Value	e5	98	27	1e	f1	11	41	b8	ae	52	b4	e0	30	5d	bf	d4

and its corresponding 4x4 matrix representation is:

4x4 Matrix Format of an xmm Register				**The Vector Arranged in the 4x4 Format**			
A	E	I	M	d4	e0	b8	1e
B	F	J	N	bf	b4	41	27
C	G	K	O	5d	52	11	98
D	H	L	P	30	ae	f1	e5

The AES Algorithm

This chapter describes the functions and the transformations used by the AES algorithm.

Cipher Key

AES is a symmetric key encryption algorithm. It uses a cipher key whose length is 128 bits, 192 bits or 256 bits. The AES algorithm with a cipher key of length 128, 192, 256 bits is denoted AES-128, AES-192, AES-256, respectively.

State

The process of encryption (decryption) of plaintext (ciphertext) to ciphertext (plaintext) generates intermediate 128-bit results. These intermediate results are referred to as the State.

Data Blocks

AES operates on an input data block of 128 bits and its output is also a data block of 128 bits.

Round Keys

AES-128, AES192, and AES-256 algorithms expand the cipher key to 10, 12, and 14 round keys, respectively. The length of each round key is 128 bits. The algorithm for deriving the round keys from the cipher key is the called the AES Key Expansion.

AddRoundKey

AddRoundKey is a (128-bit, 128-bit) → 128-bit transformation, which is defined as the bit-wise xor of its two arguments. In the AES flow, these arguments are the State and the round key. AddRoundKey is its own inverse.

Counting the Rounds and the Round Keys

The AES algorithm starts with a whitening step, implemented by XOR-ing the input data block with the first 128 bits of the cipher key. These 128 bits are the whitening key. The algorithm continues with 10/12/14 rounds, each one using another round key. When counting this way, the rounds and the round keys are counted from 1 to 10/12/14, accordingly. However, sometimes the whitening step is also referred to as "Round 0", and the corresponding 128 bits of the whitening key are referred to as Round Key 0. In that case, the count of the AES rounds and the round keys starts from 0 to 10/12/14. We use these conventions interchangeably.

S-Box and InvS-Sbox

S-Box (Substitution Box) is an 8-bit → 8-bit transformation defined as the affine function $x \rightarrow A\, x^{-1} + b$ where A is an 8x8 binary matrix and b is an 8-bit binary vector, as follows:

$$\begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{pmatrix} \rightarrow \begin{pmatrix} 11111000 \\ 01111100 \\ 00111110 \\ 00011111 \\ 10001111 \\ 11000111 \\ 11100011 \\ 11110001 \end{pmatrix} \begin{pmatrix} x7 \\ x6 \\ x5 \\ x4 \\ x3 \\ x2 \\ x1 \\ x0 \end{pmatrix}^{-1} + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{pmatrix}$$

Here, the notations for "addition" and "multiplication" represent, respectively, bitwise XOR and bitwise AND. Also, $(\ )^{-1}$ denotes inversion in the Galois Field (Finite Field) GF(2^8) defined by the reduction polynomial $x^8+x^4+x^3+x+1$ (0x11b for short). Hereafter, this field is referred to as AES-GF256-Field.

InvS-Box is the inverse of S-Box transformation, defined as $y \rightarrow (A^{-1}\, y + A^{-1}\, b)^{-1}$.

$$\begin{pmatrix}x7\\x6\\x5\\x4\\x3\\x2\\x1\\x0\end{pmatrix}\rightarrow\left(\begin{pmatrix}01010010\\00101001\\10010100\\01001010\\00100101\\10010010\\01001001\\10100100\end{pmatrix}\begin{pmatrix}x7\\x6\\x5\\x4\\x3\\x2\\x1\\x0\end{pmatrix}+\begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\1\end{pmatrix}\right)^{-1}$$

S-Box and InvS-Box Lookup Tables

The S-Box and InvS-Box transformations can also be defined by lookup table as follows. The input to the lookup tables is a byte B [7-0] where x and y denote its low and high nibbles: x [3-0] = B [7-4], y [3-0] = B [3-0].The output byte is encoded in the table as a two digit number in hexadecimal notation. For example, S-Box lookup for the input 85 (x=8; y=5 in hexadecimal notation) yields 97 in hexadecimal notation. InvS-Box lookup for the input 97 yields 85.

Figure 2. S-Box and InvS-Box Lookup Tables

```
S-Box lookup table
               ←------------------ y ----------------------->
                0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
          ^ 0 63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76
          | 1 ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0
          | 2 b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15
          | 3 04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75
          | 4 09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84
          | 5 53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf
            6 d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8
          x 7 51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2
            8 cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73
          | 9 60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db
          | a e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79
          | b e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08
          | c ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a
          | d 70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e
          | e e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df
          V f 8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16

InvS-Box lookup table
              ←------------------ y ----------------------->
               0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
         ^ 0 52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb
         | 1 7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb
         | 2 54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e
         | 3 08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25
         | 4 72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92
         | 5 6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84
           6 90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06
         x 7 d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b
           8 3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73
         | 9 96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e
         | a 47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b
         | b fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4
         | c 1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f
         | d 60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef
         | e a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61
         V f 17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d
```

SubBytes Transformation

SubBytes is the 16-byte → 16-byte transformation defined by applying the S-Box transformation to each one of the 16 bytes of the input, namely:

> [P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [S-Box (P), S-Box (O), S-Box (N), S-Box (M), S-Box (L), S-Box (K), S-Box (J), S-Box (I), S-Box (H), S-Box (G), S-Box (F), S-Box (E), S-Box (D), S-Box (C), S-Box (B), S-Box (A)].

SubBytes Example

SubBytes (73744765635354655d5b56727b746f5d) =
8f92a04dfbed204d4c39b1402192a84c

InvSubBytes Transformation

InvSubBytes is a 16-byte → 16-byte transformation defined by applying the InvS-Box function to each byte of the input, namely:

> [P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [InvS-Box (P), InvS-Box (O), InvS-Box (N), InvS-Box (M), InvS-Box (L), InvS-Box (K), InvS-Box (J), InvS-Box (I), InvS-Box (H), InvS-Box (G), InvS-Box (F), InvS-Box (E), InvS-Box (D), InvS-Box (C), InvS-Box (B), InvS-Box (A)].

InvSubBytes Example

InvSubBytes (5d7456657b536f65735b47726374545d) =
8dcab9bc035006bc8f57161e00cafd8d

ShiftRows Transformation

ShiftRows is the following byte-wise permutation: (15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0) → (11, 6, 1, 12, 7, 2, 13, 8, 3, 14, 9, 4, 15, 10, 5, 0). In the P-A notation is reads [P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [L,G,B,M,H,C,N,I,D,O,J,E,P,K,F,A]. Its name comes from viewing the transformation as an operation on the 4x4 matrix representation of the State. Under this view, the first row is unchanged, the second row is left rotated by one byte position, the third row is left rotated by two byte positions, and the fourth row is left rotated by three byte positions.

ShiftRows Example

ShiftRows (7b5b54657374566563746f725d53475d) = 73744765635354655d5b56727b746f5d

InvShiftRows Transformation

InvShiftRows is the inverse of ShiftRows. It is the following byte-wise permutation: (15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0) → (3, 6, 9, 12, 15, 2, 5, 8, 11, 14, 1, 4, 7, 10, 13, 0). In the P-A notation is reads [P,O,N,M,L,K,J,I,H,G,F,E,D,C,B,A] → [D,G,J,M,P,C,F,I,L,O,B,E,H,K,N,A]

InvShiftRows Example

InvShiftRows (7b5b54657374566563746f725d53475d) = 5d7456657b536f65735b47726374545d

MixColumns Transformation

MixColumns is a 16-byte → 16-byte transformation operating on the columns of the 4x4 matrix representation of the input. The transformation treats each column as a third degree polynomial with coefficients in AES-GF256-Field. Each column of the 4x4 matrix representation of the State is multiplied by polynomial $a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$ and reduced modulo $x^4 + 1$. Here, { } denotes an element in AES-GF256-Field. The equations that define MixColumns are detailed in Figure 3. The transformation is [P – A] → [P' – A']; the symbol • denotes multiplication in AES-GF256-Field (i.e., • is a carry-less multiplication followed by reduction modulo 0x11b); the symbol + denotes XOR.

Figure 3. MixColumns Transformation Equations

```
A' = ({02} • A) + ({03} • B) + C + D
B' = A + ({02} • B) + ({03} • C) + D
C' = A + B + ({02} • C) + ({03} • D)
D' = ({03} • A) + B + C + ({02} • D)
E' = ({02} • E) + ({03} • F) + G + H
F' = E + ({02} • F) + ({03} • G) + H
G' = E + F + ({02} • G) + ({03} • H)
H' = ({03} • E) + F + G + ({02} • H)
I' = ({02} • I) + ({03} • J) + K + L
J' = I + ({02} • J) + ({03} • K) + L
K' = I + J + ({02} • K) + ({03} • L)
L' = ({03} • I) + J + K + ({02} • L)
M' = ({02} • M) + ({03} • N) + O + P
N' = M + ({02} • N) + ({03} • O) + P
O' = M + N + ({02} • O) + ({03} • P)
P' = ({03} • M) + N + O + ({02} • P)
```

MixColumns Example

MixColums (627a6f6644b109c82b18330a81c3b3e5) = 7b5b54657374566563746f725d53475d

InvMixColumns Transformation

InvMixColumns is a 16-byte → 16-byte transformation operating on the columns of the 4x4 matrix representation of the input. It is the inverse of MixColumns**.** The transformation treats each column as a third degree polynomial with coefficients in AES-GF256-Field. Each column of the 4x4 matrix representation of the state is multiplied by polynomial $a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}$ and reduced modulo $x^4 + 1$. The equations that define InvMixColumns are detailed in Figure 4. The transformation is [P – A] → [P' – A']; the symbol • denotes multiplication in AES-GF256-Field (i.e., • is a carry-less multiplication followed by reduction mod 0x11b); the symbol + denotes XOR.

Figure 4. InvMixColumns Transformation Equations

```
A' = ({0e} • A) + ({0b} • B) + ({0d} • C) + ({09} • D)
B' = ({09} • A) + ({0e} • B) + ({0b} • C) + ({0d} • D)
C' = ({0d} • A) + ({09} • B) + ({0e} • C) + ({0b} • D)
D' = ({0b} • A) + ({0d} • B) + ({09} • C) + ({0e} • D)
E' = ({0e} • E) + ({0b} • F) + ({0d} • G) + ({09} • H)
F' = ({09} • E) + ({0e} • F) + ({0b} • G) + ({0d} • H)
G' = ({0d} • E) + ({09} • F) + ({0e} • G) + ({0b} • H)
H' = ({0b} • E) + ({0d} • F) + ({09} • G) + ({0e} • H)
I' = ({0e} • I) + ({0b} • J) + ({0d} • K) + ({09} • L)
J' = ({09} • I) + ({0e} • J) + ({0b} • K) + ({0d} • L)
K' = ({0d} • I) + ({09} • J) + ({0e} • K) + ({0b} • L)
L' = ({0b} • I) + ({0d} • J) + ({09} • K) + ({0e} • L)
M' = ({0e} • M) + ({0b} • N) + ({0d} • O) + ({09} • P)
N' = ({09} • M) + ({0e} • N) + ({0b} • O) + ({0d} • P)
O' = ({0d} • M) + ({09} • N) + ({0e} • O) + ({0b} • P)
P' = ({0b} • M) + ({0d} • N) + ({09} • O) + ({0e} • P)
```

InvMixColumns Example

InvMixColumns (8dcab9dc035006bc8f57161e00cafd8d) = d635a667928b5eaeeec9cc3bc55f5777

SubWord Transformation

SubWord is the doubleword → doubleword transformation defined by applying the S-Box transformation to each one of the 4 bytes of the input, namely:

SubWord (X) = [S-Box(X[31-24]), S-Box(X[23-16]), S-Box(X[15-8]), S-Box(X[7-0])]

SubWord Example

SubWord (73744765) = 8f92a04d

RotWord Transformation

RotWord is the doubleword → doubleword transformation defined by:

RotWord (X [31-0]) = [X[7-0], X [31-24], X [23-16], X [15-8]]

(in C language notation, RotWord(X) = (X >> 8) | (X << 24))

RotWord Example

RotWord (3c4fcf09) = 093c4fcf

Round Constant (RCON)

The AES key expansion procedure uses ten constants called Round Constants (RCON hereafter). The ten RCON values are RCON [i] = $\{02\}^{i-1}$ for i=1, 2, …, 10, where the operations are in AES-GF256-Field.

Each RCON value is an element of AES-GF256-Field, and is encoded here as a byte. The ten RCON values are (in hexadecimal notation):

RCON [1] = 0x01, RCON [2] = 0x02, RCON [3] = 0x04, RCON [4] = 0x08, RCON [5] = 0x10,

RCON [6] = 0x20, RCON [7] = 0x40, RCON [8] = 0x80, RCON [9] = 0x1B, RCON [10] = 0x36.

Remark: in the following RCON values are also viewed, interchangedly, as doublewords where their 24 leftmost bits equal 0. For example, RCON [7] = 0x00000040 (in hexadecimal notation).

Key Expansion

AES uses a cipher key whose length is 128, 192 or 256 bits. This cipher key is expanded into 10, 12, or 14 round keys, respectively, using the "Key Expansion" algorithm, where the length of each round key is 128 bits. This Key Expansion algorithm depends only on the cipher key. Since it is independent of the processed data, it can be (and typically is) executed prior to the encryption/decryption phase. At the heart of the Key Expansion algorithm is the combination of the transformations SubWord(RotWord(tmp)) and SubWord(tmp) and the use of the RCON values. Figure 5 shows the pseudo code for the AES Key Expansion algorithm (as described in FIPS197).

Figure 5. AES Key Expansion Pseudo Code (as Described in FIPS197)

```
Parameters
Nb = 4 (data blocks are of 128 bits)
Nk = number of doublewords in the cipher key
    (4, 6, 8 for AES-128, AES-192, AES-256, resp.)
Nr = number of rounds in the cipher
     (Nr=10, 12, 14 for AES-128, AES-192, AES-256, respectively).

The Key Expansion routine

KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
```

```
word tmp
i = 0
while (i < Nk)
    w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
    i = i+1
end while
i = Nk
while (i < Nb * (Nr+1))
    tmp = w[i-1]
    if (i mod Nk = 0)
        tmp = SubWord(RotWord(tmp)) xor RCON[i/Nk]
    else
    if (Nk = 8)
        tmp = SubWord(tmp)
    end if
    w[i] = w[i-Nk] xor tmp
    i = i + 1
end while
```

AES Encryption and Decryption Flows

The Order of Transformations

SubBytes transformation operates separately on each byte of the State, while the ShiftRows transformation operates on the columns of the State. Therefore, ShiftRows and SubBytes commute with each other. Similarly, InvShiftRows and InvSubBytes commute.

In the following, we use the notation where ShiftRows is applied before SubBytes and InvShiftRows is applied before InvSubBytes (unlike the order that appears in FIPS197).

Decryption with the Equivalent Inverse Cipher

There are two equivalent ways to perform AES decryption, one is called the "Inverse Cipher" and the other is called the "Equivalent Inverse Cipher". They differ in the internal order of the sequence of the (inverse) transformations, and also in the way that the decryption round keys are defined.

Intel architecture uses the Equivalent Inverse Cipher for decryption.

To use the "Equivalent Inverse Cipher", the round keys for the decryption must be properly prepared, as follows. Denote the round key which is used in round number j during the encryption flow, by Round_Key_Encrypt [j], j=0, 1, 2, …, Nr. Here, Nr = 10/12/14 for AES-128, AES-192, AES-256, respectively. Denote the decryption round key, which is used during round j of the decryption flow, by Round_Key_Decrypt [j]. The following figure shows how the decryption round keys can be derived from the encryption round keys.

Figure 6. Preparing the Decryption Round Keys

```
Round_Key_Decrypt [0] = Round_Key_Encrypt [Nr]
for round = 1, 2, to Nr-1
    Round_Key_Decrypt [round] = InvMixColumns (Round_Key_Encrypt [Nr- round])
Round_Key_Decrypt [Nr] = Round_Key_Encrypt [0]
```

Encryption and Decryption Flows

AES encryption and decryption flows consist of a back-to-back sequence of AES transformations, operating on a 128-bit State (data) and a round key. The flows depend on the cipher key length, where

AES-128 encryption/decryption consists of 40 steps (transformations)

AES-192 encryption/decryption consists of 48 steps

AES-256 encryption/decryption consists of 56 steps

The color code is used in Figure 7 and Figure 8 for describing the respective different flows. In the following pseudo code, it is assumed that the **10**, **12,** or **14** round keys are already derived (expanded) from the cipher key and stored, in the proper order, in an array (Round_Key_Encrypt and Round_Key_Decrypt). Here, it is assumed that Round_Key_Encrypt [0] stores the first 128 bits of the cipher key, which is used for the first XOR operation (aka round 0).

Figure 7. The AES Encryption Flow

```
; Data is a 128-bit block to be encrypted.
; The round keys are stored in the array Round_Key_Encrypt

     Tmp = AddRoundKey (Data, Round_Key_Encrypt [0])
     For round =  1-9 or 1-11 or 1-13:
       Tmp = ShiftRows  (Tmp)
       Tmp = SubBytes (Tmp)
       Tmp = MixColumns (Tmp)
       Tmp = AddRoundKey (Tmp, Round_Key_Encrypt [round])
     end loop
     Tmp = ShiftRows (Tmp)
     Tmp = SubBytes (Tmp)
     Tmp = AddRoundKey (Tmp, Round_Key_Encrypt [10 or 12 or 14])
     Result = Tmp
```

Figure 8. The AES Decryption Flow (Using the Equivalent Inverse Cipher)

```
; Data is a 128-bit block to be decrypted.
; The decryption round keys are stored in the array Round_Key_Decrypt
; (it is assumed here that decryption round keys have been properly prepared
; to be used by the Equivalent Inverse Cipher)

     Tmp = AddRoundKey (Data, Round_Key_Decrypt [0])
     For round =  1-9 or 1-11 or 1-13:
       Tmp = InvShiftRows  (Tmp)
       Tmp = InvSubBytes (Tmp)
       Tmp = InvMixColumns (Tmp)
       Tmp = AddRoundKey (Tmp, Round_Key_Decrypt [round])
     end loop
     Tmp = InvShiftRows (Tmp)
     Tmp = InvSubBytes (Tmp)
     Tmp = AddRoundKey (Tmp, Round_Key_Decrypt [10 or 12 or 14])
     Result = Tmp
```

Intel® AES Architecture

The AES instructions set consists of six instructions.

Four instructions, namely AESENC, AESENCLAST, AESDEC, AESDECLAST, are provided for data encryption and decryption (the names are short for AES Encrypt Round, AES Encrypt Last Round, AES Decrypt Round AES Decrypt Last Round). These instructions have both register-register and register-memory variants.

Two other instructions, namely AESIMC and AESKEYGENASSIST are provided in order to assist with AES Key Expansion (the names are short for AES Inverse Mix Columns, and AES Key Generation Assist).

The Four AES Round Instructions

AESENC, AESENCLAST, AESDEC, AESDECLAST are defined by the pseudo code in the following figures (“xmm1” and “xmm2” are aliases to any two xmm registers). These instructions perform a grouped sequence of transformations of the AES encryption/decryption flows (in fact, they perform the longest sequence possible, without introducing a branch in an instruction).

Figure 9. The AESENC and AESENCLAST Instructions

```
 AESENC xmm1, xmm2/m128             AESENCLAST xmm1, xmm2/m128
 Tmp := xmm1                        Tmp := xmm1
 Round Key := xmm2/m128             Round Key := xmm2/m128
 Tmp := ShiftRows (Tmp)             Tmp := Shift Rows (Tmp)
 Tmp := SubBytes (Tmp)              Tmp := SubBytes (Tmp)
 Tmp := MixColumns (Tmp)
 xmm1 := Tmp xor Round Key          xmm1 := Tmp xor Round Key
```

Figure 10. The AESDEC and AESDECLAST Instructions

```
 AESDEC xmm1, xmm2/m128             AESDECLAST xmm1, xmm2/m128
 Tmp := xmm1                        State  := xmm1
 Round Key := xmm2/m128             Round Key  := xmm2/m128
 Tmp := InvShift Rows (Tmp)         Tmp := InvShift Rows (State)
 Tmp := InvSubBytes (Tmp)           Tmp := InvSubBytes (Tmp)
 Tmp := InvMixColumns (Tmp)
 xmm1 := Tmp xor Round Key          xmm1:=  Tmp xor Round Key
```

AESENC Example

Figure 11. AESENC Example

```
; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key).
; The result is delivered in xmm1.
 xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
 AESENC result (in xmm1): a8311c2f9fdba3c58b104b58ded7e595
```

AESENCLAST Example

Figure 12. AESENCLAST Example

```
; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key)
; The result delivered in xmm1
```

```
 xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
 AESENCLAST result (in xmm1): c7fb881e938c5964177ec42553fdc611
```

AESEDEC Example

Figure 13. AESDEC Example

```
; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key).
; The result delivered in xmm1.
 xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
 AESDEC result (in xmm1): 138ac342faea2787b58eb95eb730392a
```

AESDECLAST Example

Figure 14. AESDECLAST Example

```
 ; xmm1 and xmm2 hold two 128-bit inputs (xmm1 = State; xmm2 = Round key).
 ; The result delivered in xmm1.
 xmm1 = 7b5b54657374566563746f725d53475d xmm2 = 48692853686179295b477565726f6e5d
 AESDECLAST result (in xmm1): c5a391ef6b317f95d410637b72a593d0
```

AES Encryption and Decryption Flows Using the AES Round Instructions

From Figures 7 and 8, and the definition of AESENC/AESENCLAST, AESDEC/AESDECLAST instructions (see Figures 9, 10) it is easy to understand how the instructions could be used for AES encryption and decryption. We provide here two outlined code examples, one showing an AES-128 encryption code sequence, and the other showing an AES-192 decryption code sequence.

Figure 15. AES-128 Encryption Outlined Code Sequence

```
 ; AES-128 encryption sequence.
 ; The data block is in xmm15.
 ; Registers xmm0–xmm10 hold the round keys(from 0 to 10 in this order).
 ; In the end, xmm15 holds the encryption result.
         pxor xmm15, xmm0                     ; Whitening step (Round 0)
         aesenc xmm15, xmm1                   ; Round 1
         aesenc xmm15, xmm2                   ; Round 2
         aesenc xmm15, xmm3                   ; Round 3
         aesenc xmm15, xmm4                   ; Round 4
         aesenc xmm15, xmm5                   ; Round 5
         aesenc xmm15, xmm6                   ; Round 6
         aesenc xmm15, xmm7                   ; Round 7
         aesenc xmm15, xmm8                   ; Round 8
         aesenc xmm15, xmm9                   ; Round 9
         aesenclast xmm15, xmm10              ; Round 10
```

Figure 16. AES-192 Decryption: Outlined Code Sequence

```
 ; AES-192 decryption sequence.
 ; The data is in xmm15.
 ; Registers xmm12 – xmm0 hold the decryption round keys.
 ; (the decryption round keys are derived from the encryption round keys by
 ; passing them (except for the first and last) through the
 ; InvMixColumns transformation.)
 ; In the end - xmm15 holds the decryption result
        pxor xmm15, xmm12           ; First xor
        aesdec xmm15, xmm11         ; Round 1 (consuming round keys in reverse order)
        aesdec xmm15, xmm10         ; Round 2
        aesdec xmm15, xmm9          ; Round 3
        aesdec xmm15, xmm8          ; Round 4
        aesdec xmm15, xmm7          ; Round 5
        aesdec xmm15, xmm6          ; Round 6
        aesdec xmm15, xmm5          ; Round 7
        aesdec xmm15, xmm4          ; Round 8
        aesdec xmm15, xmm3          ; Round 9
        aesdec xmm15, xmm2          ; Round 10
        aesdec xmm15, xmm1          ; Round 11
        aesdeclast xmm15, xmm0      ; Round 12
```

The Two Instructions for Supporting AES Key Expansion

AES Key Expansion is supported by two instructions. AESKEYGENASSIST is used for generating the round keys used for encryption. AESIMC is used for converting the encryption round keys to a form usable for decryption using the Equivalent Inverse Cipher.

The AESKEYGENASSIST Instruction

Figure 17. The AESKEYGENASSIST Instruction

```
AESKEYGENASSIST xmm1, xmm2/m128, imm8
Tmp := xmm2/LOAD(m128)
X3[31-0] := Tmp[127-96];
X2[31-0] := Tmp[95-64];
X1[31-0] := Tmp[63-32];
X0[31-0] := Tmp[31-0];
RCON[7-0] := imm8;
RCON [31-8] := 0;
xmm1 := [RotWord (SubWord (X3)) XOR RCON, SubWord (X3),
                       RotWord (SubWord (X1)) XOR RCON, SubWord (X1)]
```

AESKEYGENASSIST Example

Figure 18. AESKEYGENASSIST Example

```
; xmm2 holds a 128-bit input; imm8 holds the RCON value
; result delivered in xmm1
xmm2 = 3c4fcf098815f7aba6d2ae2816157e2b imm8 = 1
AESKEYGENASSIST result (in xmm1): 01eb848beb848a013424b5e524b5e434
```

Key Expansion Using AESKEYGENASSIST

Figure 5 show the AES Key Expansion flow, for the different key sizes (128/192/256 bits). From this software flow, it is clear that the AESKEYGENASSIST instruction is designed to be used for two operations in the expansion sequence, namely

```
tmp = SubWord(RotWord(tmp)) xor RCON[i/Nk]
```

and

```
tmp = SubWord(tmp)
```

where the latter is relevant only for the 256-bits key expansion.

There are several possible ways to expand the key using AESKEYGENASSIST, and full code demonstrations are listed below. We give here the example for AES-128.

Figure 19. AES-128 Key Expansion: Outlined Code Example

```
; Cipher key is stored in “Key”. For example,
;    Key       0x0f0e0d0c0b0a09080706050403020100
; The key scheduled to be stored in the array Key_Schedule.

         movdqu xmm1, XMMWORD PTR Key
         movdqu XMMWORD PTR Key_Schedule, xmm1
         mov rcx, OFFSET Key_Schedule+16

         aeskeygenassist xmm2, xmm1, 0x1
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x2
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x4
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x8
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x10
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x20
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x40
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x80
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x1b
         call key_expansion_128
         aeskeygenassist xmm2, xmm1, 0x36
         call key_expansion_128
         jmp END;

 key_expansion_128:
         pshufd xmm2, xmm2, 0xff
         vpslldq xmm3, xmm1, 0x4
         pxor xmm1, xmm3
         vpslldq xmm3, xmm1, 0x4
         pxor xmm1, xmm3
         vpslldq xmm3, xmm1, 0x4
```

```
        pxor xmm1, xmm3
        pxor xmm1, xmm2
        movdqu XMMWORD PTR [rcx], xmm1
        add rcx, 0x10
        ret
END:
```

Preparing the Decryption Round Keys Using AESIMC

By their definition, AESDEC and AESDECLAST should be used for decryption with the Equivalent Inverse Cipher. To this end, the encryption round keys 1-9/11/13 (for AES-128/AES-192/AES-256, respectively) need to be first passed through the InvMixColumns transformation. This can be easily done by using the AESIMC instruction, which is defined by the following pseudo code.

Figure 20. The AESIMC Instruction

```
AESIMC xmm1, xmm2/m128
RoundKey := xmm2/m128;
xmm1 := InvMixColumns (RoundKey)
```

AESIMC Example

Figure 21. AESIMC Example

```
; xmm2 hold one 128-bit inputs (xmm2 = Round key)
; result delivered in xmm1
xmm2 = 48692853686179295b477565726f6e5d
AESIMC result (in xmm1): 627a6f6644b109c82b18330a81c3b3e5
```

Generating AES Decryption Round Keys

The following Assembly code snippet shows an example for generating an AES-128 key schedule for decryption.

Figure 22. Using AESIMC for AES-128: Outlined Code Example

```
; The array Key_Schedule holds the expanded round keys (round keys 0-10).
; The decryption round keys are to be stored in the array Key_Schedule_Decrypt.
; Transforming the encryption round keys to decryption keys is done by passing round
; keys 1-9 through InMixColumns transformation (using aesimc instruction), to be
; ready to use with the Equivalent Inverse Cipher.

mov rdx, OFFSET Key_Schedule
mov rax, OFFSET Key_Schedule_Decrypt

movdqu xmm1, XMMWORD PTR [rdx]
movdqu XMMWORD PTR [rax], xmm1
add rdx, 0x10
add rax, 0x10

mov ecx, 9                     ; 9 for AES-128, 11 for AES-192, 13 for AES-256
repeat_Nr_minus_one_times:
    movdqu xmm1, XMMWORD PTR [rdx]
    aesimc xmm1, xmm1
    movdqu XMMWORD PTR [rax], xmm1
    add rdx, 0x10
    add rax, 0x10
loop repeat_Nr_minus_one_times

movdqu xmm1, XMMWORD PTR [rdx]
movdqu XMMWORD PTR [rax], xmm1
```

Application Programming Model

The AES extensions follow the same programming model as Intel SSE, Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4 (see IA-32 Intel Architecture Software Developer’s Manual, Volume 1). Operating systems that support handling Intel SSE state will also support applications that use the AES instructions. This is the same requirement for Legacy Intel SSE (i.e., Intel SSE2, Intel SSE3, Intel SSSE3, and Intel SSE4).

Detecting AES Instructions

Before an application attempts to use the AES instructions, it should verify that the processor indeed supports these instructions. This is done by checking that CPUID.01H:ECX.AES[bit 25] = 1.

Software Side Channels and the AES Instructions

This chapter provides a brief description of software side channel attacks and explains why memory access patterns can be used against software implementations of AES that use table lookups.

What are Software Side Channel Attacks?

Software side channels are a set of vulnerabilities targeting modern computing environments. They can be potentially used for attacking cryptographic applications that run on a multi-tasking platform.

These attacks use the fact that practically all current commercial computer platforms run multiple tasks on a single set of hardware resources. Some recent publications showed that multi-tasking operating systems combined with processor’s resource sharing can lead to side channel information leaks, where an unprivileged spy process (“spy” hereafter) running in parallel to some cryptosystem (“crypto” hereafter), at the same privilege level (“Ring 3”) can obtain information on crypto's memory access patterns, or on its execution flow. In some cases (depending on the way that the crypto program is written), this information can be used for compromising secret information (e.g., keys).

The focus of this chapter is software side channels, and how the currently known timing and cache attacks on AES can be mitigated by using the AES instructions.

The CPU Cache and the Basics of Cache Attacks

Cache is a widely used performance optimization technique, employed by virtually all modern processors. The cache is a special (and expensive) type of memory that can be accessed much faster than the main memory. It is used by the CPU for storing the recently read areas of memory. In each memory access, the CPU first checks if the required data is already in the cache. In that case (called cache hit), the memory access is very fast. If the required data is not in the cache (cache miss), it is read from the memory (more slowly) and also stored in the cache for future reads. Obviously, storing new data in the cache requires that the CPU evicts some previously loaded data - typically the least recently used data. The cache helps reducing the average memory access time by a significant amount. However, there is a side effect: the time for reading a particular piece of data depends on whether or not this data is already in the cache. This depends on the specific contents of the cache in the relevant moment, which is the overall result of all the processes that run on the platform. Malicious code could potentially exploit this cache behavior and attack cryptographic applications which involve data-dependent memory access in sensitive steps.

Lookup Tables and the Implied Vulnerability

Currently, many efficient and commonly used AES software implementations on the PC platform use lookup tables (see e.g., Gladman's implementation http://fp.gladman.plus.com/ or the OpenSSL code http://www.openssl.org). These tables are large, and therefore span across several cache lines. As a result, different cache lines may be accesses when different parts of the table are read. Unfortunately, in some critical steps of the AES algorithm, the accessed parts of the tables depend (implicitly) on the secret key, and this fact introduces vulnerabilities.

A potential attack has a spy process executing on the same system as the "victim" crypto process. The spy repeatedly performs data reads, and this way it causes the cache to be filled with its own data. It measures (using the RDTSC instruction) the latency of its own reads, and can therefore identify those cache lines which have been meanwhile evicted by the operation of the crypto (AES) process that runs in parallel. This can be used to deduce which parts of the tables were accessed by the AES operation. Analysis of this information can lead to revealing the secret key (typically, the information on the first and the last AES rounds leak the most sensitive information). We give here two (among many existing) references that provide more details on side channel attacks on software implementation of AES: D. A. Osvik, A. Shamir, and E. Tromer, "Cache Attacks and Countermeasures: The Case of AES", Lecture Notes in Computer Science series, Springer-Verlag, 3860: 1-20, (2006) and also D. J. Bernstein, "Cache-timing attacks on AES", http://people.csail.mit.edu/tromer/papers/cache.pdf (2005).

Software Mitigation for AES Carries a Performance Penalty

There are ways to write AES software in a way that avoids the key-dependent memory access. One example is to permute the lookup tables in order to obscure the undesired data dependency. Details can be found in the paper: E. Brickell, E., G. Graunke, M. Neve, J. P. Seifert, "Software mitigations to hedge AES against cache based software side channel vulnerabilities" http://eprint.iacr.org/2006/052.pdf. However, these mitigation techniques carry significant performance penalty. Another way (called Bit Slicing) is to write the AES software without lookup tables at all, for example, as shown in: M. Matsui and S. Fukuda. "How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors". LNCS, Springer Verlag, 3557: 398–412 (2005). This approach also involves performance penalty. In general, software implementations of AES, that included techniques for avoiding key-dependent memory accesses are slower than the optimized (but vulnerable) implementations based on lookup tables.

Recent results by Bernstein and Schwabe, "New AES Software Speed Records", Proceedings of INDOCRYPT 2008, Lecture Notes in Computer Science 5365:322-336 (2008) show very fast implementations of AES-128 in CTR mode, using bit slicing (without lookup tables). This paper also includes a comprehensive survey of performance results of software implementations of AES.

The AES Instructions Help Protecting Against Side Channels Attacks

The AES instructions are designed to mitigate all of the known timing and cache side channel leakage of sensitive data (from Ring 3 spy processes). Their latency is data-independent, and since all the computations are performed internally by the hardware, no lookup tables are required. Therefore, if the AES instructions are used properly (e.g., as in the following code examples) the AES encryption/decryption, as well as the Key Expansion, would have data-independent timing and would involve only data-independent memory access. Consequently, the AES instructions allow for writing high performance AES software which is, at the same time, protected against the currently known software side channel attacks.

Basic C Code Examples

This chapter provides C code examples illustrating the basic usage of the AES instructions. The examples are provided as function written in C, using compiler intrinsics, and they include functions for AES128/192/256 key expansion, encryption and decryption in ECB, CBC, CTR modes of operation.

The interface of these functions is similar to the interface of OpenSSL, with some slight modifications:

CBC mode: the CBC function behaves exactly as the analogous function of OpenSSL.

ECB mode: OpenSSL does not have ECB mode for encrypting a buffer, but rather provides a function for encrypting a single block. The ECB function provided here, has the same interface, but also receives the buffer length as a parameter (and operates on that buffer).

CTR mode: the OpenSSL function receives a pre-computed counter block as input. Our function receives the IV and the nonce, and builds the counter block.

The functions can be compiled and linked with the test functions that are provided in the "Test Functions" Section (see below), in order to generate a running executable.

The intention is to provide the basic examples, in order to help software writers develop their applications. The emphasis was given here to code clarity, simplicity, and portability (the code samples can be run on Linux/Windows and can be compiled with icc, gcc and the Microsoft compiler). However, it should be noted that these code examples are not necessarily optimized for performance. An optimized library (written in assembly) is provided in a separate chapter.

Using the Code Examples with ICC/gcc and the Software Development Emulator

The code examples presented here were compiled and run on a Linux environment, using both the Intel® C Compiler (icc) and with gcc. The ICC and gcc compilers support the AES instructions from version ICC 11.1 and gcc 4.4, respectively.

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

Detecting AES Instructions

Before an application attempts to use the AES instructions, it should verify that the processor supports these instructions. This should be done by checking that `CPUID.01H:ECX.AES[bit 25] = 1`. The following (assembly) code demonstrates this check.

Figure 23. Checking the CPU Support for the AES Instructions

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

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

AES-128, AES-192, and AES-256 Key Expansion (C code)

Figure 24. AES-128 Key Expansion (C code)

```
#include <wmmintrin.h>

inline __m128i AES_128_ASSIST (__m128i temp1, __m128i temp2)
    {
    __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);
    return temp1;
    }

void AES_128_Key_Expansion (const unsigned char *userkey,
                            unsigned char *key)
    {
    __m128i temp1, temp2;
    __m128i *Key_Schedule = (__m128i*)key;

    temp1 = _mm_loadu_si128((__m128i*)userkey);
    Key_Schedule[0] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1 ,0x1);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[1] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x2);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[2] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x4);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[3] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x8);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[4] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x10);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[5] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x20);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[6] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x40);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[7] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x80);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[8] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x1b);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[9] = temp1;
    temp2 = _mm_aeskeygenassist_si128 (temp1,0x36);
    temp1 = AES_128_ASSIST(temp1, temp2);
    Key_Schedule[10] = temp1;
    }
```

Figure 25. AES-192 Key Expansion (C code)

```
#include <wmmintrin.h>

inline void KEY_192_ASSIST(__m128i* temp1, __m128i * temp2, __m128i * temp3)
    {
    __m128i temp4;
    *temp2 = _mm_shuffle_epi32 (*temp2, 0x55);
    temp4 = _mm_slli_si128 (*temp1, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    *temp1 = _mm_xor_si128 (*temp1, *temp2);
    *temp2 = _mm_shuffle_epi32(*temp1, 0xff);
    temp4 = _mm_slli_si128 (*temp3, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    *temp3 = _mm_xor_si128 (*temp3, *temp2);
    }


void AES_192_Key_Expansion (const unsigned char *userkey,
                            unsigned char *key)
    {
    __m128i temp1, temp2, temp3, temp4;
    __m128i *Key_Schedule = (__m128i*)key;
    temp1 = _mm_loadu_si128((__m128i*)userkey);
    temp3 = _mm_loadu_si128((__m128i*)(userkey+16));

    Key_Schedule[0]=temp1;
    Key_Schedule[1]=temp3;
    temp2=_mm_aeskeygenassist_si128 (temp3,0x1);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[1] = (__m128i)_mm_shuffle_pd((__m128d)Key_Schedule[1],
                                              (__m128d)temp1,0);
    Key_Schedule[2] = (__m128i)_mm_shuffle_pd((__m128d)temp1,(__m128d)temp3,1);
    temp2=_mm_aeskeygenassist_si128 (temp3,0x2);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[3]=temp1;
    Key_Schedule[4]=temp3;
    temp2=_mm_aeskeygenassist_si128 (temp3,0x4);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[4] = (__m128i)_mm_shuffle_pd((__m128d)Key_Schedule[4],
                                              (__m128d)temp1,0);
    Key_Schedule[5] = (__m128i)_mm_shuffle_pd((__m128d)temp1,(__m128d)temp3,1);
    temp2=_mm_aeskeygenassist_si128 (temp3,0x8);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[6]=temp1;
    Key_Schedule[7]=temp3;
    temp2=_mm_aeskeygenassist_si128 (temp3,0x10);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[7] = (__m128i)_mm_shuffle_pd((__m128d)Key_Schedule[7],
                                              (__m128d)temp1,0);
    Key_Schedule[8] = (__m128i)_mm_shuffle_pd((__m128d)temp1,(__m128d)temp3,1);
    temp2=_mm_aeskeygenassist_si128 (temp3,0x20);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[9]=temp1;
    Key_Schedule[10]=temp3;
    temp2=_mm_aeskeygenassist_si128 (temp3,0x40);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[10] = (__m128i)_mm_shuffle_pd((__m128d)Key_Schedule[10],
                                               (__m128d)temp1,0);
    Key_Schedule[11] = (__m128i)_mm_shuffle_pd((__m128d)temp1,(__m128d)temp3,1);
    temp2=_mm_aeskeygenassist_si128 (temp3,0x80);
    KEY_192_ASSIST(&temp1, &temp2, &temp3);
    Key_Schedule[12]=temp1;
    Key_Schedule[13]=temp3;
    }
```

Figure 26. AES-256 Key Expansion (C code)

```
#include <wmmintrin.h>

inline void KEY_256_ASSIST_1(__m128i* temp1, __m128i * temp2)
    {
    __m128i temp4;
    *temp2 = _mm_shuffle_epi32(*temp2, 0xff);
    temp4 = _mm_slli_si128 (*temp1, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp1 = _mm_xor_si128 (*temp1, temp4);
    *temp1 = _mm_xor_si128 (*temp1, *temp2);
    }

inline void KEY_256_ASSIST_2(__m128i* temp1, __m128i * temp3)
    {
    __m128i temp2,temp4;
    temp4 = _mm_aeskeygenassist_si128 (*temp1, 0x0);
    temp2 = _mm_shuffle_epi32(temp4, 0xaa);
    temp4 = _mm_slli_si128 (*temp3, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    temp4 = _mm_slli_si128 (temp4, 0x4);
    *temp3 = _mm_xor_si128 (*temp3, temp4);
    *temp3 = _mm_xor_si128 (*temp3, temp2);
    }

void AES_256_Key_Expansion (const unsigned char *userkey,
                            unsigned char *key)
    {
    __m128i temp1, temp2, temp3;
    __m128i *Key_Schedule = (__m128i*)key;

    temp1 = _mm_loadu_si128((__m128i*)userkey);
    temp3 = _mm_loadu_si128((__m128i*)(userkey+16));
    Key_Schedule[0] = temp1;
    Key_Schedule[1] = temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x01);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[2]=temp1;
    KEY_256_ASSIST_2(&temp1, &temp3);
    Key_Schedule[3]=temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x02);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[4]=temp1;
    KEY_256_ASSIST_2(&temp1, &temp3);
    Key_Schedule[5]=temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x04);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[6]=temp1;
    KEY_256_ASSIST_2(&temp1, &temp3);
    Key_Schedule[7]=temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x08);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[8]=temp1;
    KEY_256_ASSIST_2(&temp1, &temp3);
    Key_Schedule[9]=temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x10);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[10]=temp1;
    KEY_256_ASSIST_2(&temp1, &temp3);
    Key_Schedule[11]=temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x20);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[12]=temp1;
    KEY_256_ASSIST_2(&temp1, &temp3);
    Key_Schedule[13]=temp3;
    temp2 = _mm_aeskeygenassist_si128 (temp3,0x40);
    KEY_256_ASSIST_1(&temp1, &temp2);
    Key_Schedule[14]=temp1;
    }
```

AES Encryption and Decryption in ECB Mode

Figure 27. AES-128, AES-192 and AES-256 Encryption and Decryption in ECB Mode (C code)

```
#include <wmmintrin.h>

/* Note – the length of the output buffer is assumed to be a multiple of 16 bytes */

void AES_ECB_encrypt(const unsigned char *in,  //pointer to the PLAINTEXT
                     unsigned char *out,       //pointer to the CIPHERTEXT buffer
                     unsigned long length,     //text length in bytes
                     const char *key,          //pointer to the expanded key schedule
                     int number_of_rounds)     //number of AES rounds 10,12 or 14
    {
    __m128i tmp;
    int i,j;

    if(length%16)
        length = length/16+1;
    else
        length = length/16;

    for(i=0; i < length; i++){
        tmp = _mm_loadu_si128 (&((__m128i*)in)[i]);
        tmp = _mm_xor_si128 (tmp,((__m128i*)key)[0]);
        for(j=1; j <number_of_rounds; j++){
            tmp = _mm_aesenc_si128 (tmp,((__m128i*)key)[j]);
        }
        tmp = _mm_aesenclast_si128 (tmp,((__m128i*)key)[j]);
        _mm_storeu_si128 (&((__m128i*)out)[i],tmp);
        }
    }

void AES_ECB_decrypt(const unsigned char *in,  //pointer to the CIPHERTEXT
                     unsigned char *out,       //pointer to the DECRYPTED TEXT buffer
                     unsigned long length,     //text length in bytes
                     const char *key,          //pointer to the expanded key schedule
                     int number_of_rounds)     //number of AES rounds 10,12 or 14
    {
    __m128i tmp;
    int i,j;

    if(length%16)
        length = length/16+1;
    else
        length = length/16;

    for(i=0; i < length; i++){
        tmp = _mm_loadu_si128 (&((__m128i*)in)[i]);
        tmp = _mm_xor_si128 (tmp,((__m128i*)key)[0]);
        for(j=1; j <number_of_rounds; j++){
            tmp = _mm_aesdec_si128 (tmp,((__m128i*)key)[j]);
        }
        tmp = _mm_aesdeclast_si128 (tmp,((__m128i*)key)[j]);
        _mm_storeu_si128 (&((__m128i*)out)[i],tmp);
        }
    }
```

AES Encryption and Decryption in CBC Mode

Figure 28. AES-128, AES-192 and AES-256 Encryption and Decryption in CBC Mode (C code)

```
#include <wmmintrin.h>

void AES_CBC_encrypt(const unsigned char *in,
                     unsigned char *out,
                     unsigned char ivec[16],
                     unsigned long length,
                     unsigned char *key,
                     int number_of_rounds)
    {
    __m128i feedback,data;
    int i,j;

    if (length%16)
        length = length/16+1;
    else length /=16;

    feedback=_mm_loadu_si128 ((__m128i*)ivec);
    for(i=0; i < length; i++){
        data = _mm_loadu_si128 (&((__m128i*)in)[i]);
        feedback = _mm_xor_si128 (data,feedback);
        feedback = _mm_xor_si128 (feedback,((__m128i*)key)[0]);
        for(j=1; j <number_of_rounds; j++)
            feedback = _mm_aesenc_si128 (feedback,((__m128i*)key)[j]);
        feedback = _mm_aesenclast_si128 (feedback,((__m128i*)key)[j]);
        _mm_storeu_si128 (&((__m128i*)out)[i],feedback);
        }
    }

void AES_CBC_decrypt(const unsigned char *in,
                     unsigned char *out,
                     unsigned char ivec[16],
                     unsigned long length,
                     unsigned char *key,
                     int number_of_rounds)
    {
    __m128i data,feedback,last_in;
    int i,j;

    if (length%16)
        length = length/16+1;
    else length /=16;

    feedback=_mm_loadu_si128 ((__m128i*)ivec);
    for(i=0; i < length; i++){
        last_in=_mm_loadu_si128 (&((__m128i*)in)[i]);
        data = _mm_xor_si128 (last_in,((__m128i*)key)[0]);
        for(j=1; j <number_of_rounds; j++){
            data = _mm_aesdec_si128 (data,((__m128i*)key)[j]);
        }
        data = _mm_aesdeclast_si128 (data,((__m128i*)key)[j]);
        data = _mm_xor_si128 (data,feedback);
        _mm_storeu_si128 (&((__m128i*)out)[i],data);
        feedback=last_in;
        }
    }
```

AES in CTR Mode

Figure 29. AES-128, AES-192 and AES-256 in CTR Mode (C code)

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

void AES_CTR_encrypt (const unsigned char *in,
                      unsigned char *out,
                      const unsigned char ivec[8],
                      const unsigned char nonce[4],
                      unsigned long length,
                      const unsigned char *key,
                      int number_of_rounds)
    {
    __m128i ctr_block, tmp, ONE, BSWAP_EPI64;
    int i,j;

    if (length%16)
        length = length/16 + 1;
    else length/=16;

    ONE = _mm_set_epi32(0,1,0,0);
    BSWAP_EPI64 = _mm_setr_epi8(7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8);

    ctr_block = _mm_insert_epi64(ctr_block, *(long long*)ivec, 1);
    ctr_block = _mm_insert_epi32(ctr_block, *(long*)nonce, 1);
    ctr_block = _mm_srli_si128(ctr_block, 4);
    ctr_block = _mm_shuffle_epi8(ctr_block, BSWAP_EPI64);
    ctr_block = _mm_add_epi64(ctr_block, ONE);

    for(i=0; i < length; i++){
        tmp = _mm_shuffle_epi8(ctr_block, BSWAP_EPI64);
        ctr_block = _mm_add_epi64(ctr_block, ONE);
        tmp = _mm_xor_si128(tmp, ((__m128i*)key)[0]);
            for(j=1; j <number_of_rounds; j++) {
                tmp = _mm_aesenc_si128 (tmp, ((__m128i*)key)[j]);
                };
            tmp = _mm_aesenclast_si128 (tmp, ((__m128i*)key)[j]);
            tmp = _mm_xor_si128(tmp,_mm_loadu_si128(&((__m128i*)in)[i]));
            _mm_storeu_si128 (&((__m128i*)out)[i],tmp);
        }
    }
```

Software Flexibility and Miscellaneous Usage Models

The AES instructions are useful for many common cryptographic applications. They support all of the AES variants defined by FIPS197, including encryption and decryption with the three standard key lengths, using the standard block size of 128 bits. The AES instructions can also be used for all common uses of AES, including bulk encryption and decryption using cipher modes such as CBC or CTR, data authentication using CBC-MACs such as CMAC, random number generation using algorithms such as CTR-DRBG, and authenticated encryption using modes such as GCM.

Software has the flexibility to pre-expand the keys and re-use them (which is the typical usage model for the PC platform) or to expand them on-the-fly. Such an example is given in a subsequenct section, below. We also provide examples for using the AES instructions for other (non-AES) Rijndael variants, and to isolating the individual AES transformations from the AES instructions.

Rijndael and other AES variants

We illustrate here the flexibility for software writers to use the AES instructions in some less common usage scenarios, such as Rijndael and related variants, and constructs that use isolated AES components.

We note that modifying standard cryptographic algorithms is likely to introduce severe security problems, and should be avoided. Therefore, the variants described here should be used only if they are standardized (or at least widely agreed to be useful and secure).

According to FIPS197 "This standard explicitly defines the allowed values for the key length (Nk), block size (Nb), and number of rounds (Nr) [5]. However, future reaffirmations of this standard could include changes or additions to the allowed values for those parameters. Therefore, implementers may choose to design their AES implementations with future flexibility in mind."

Here, we consider the applicability of the AES instructions to such variants of AES. Hereafter, we refer to the original Rijndael cipher, on which AES is based, as "RIJNDAEL".

Changing the number of rounds

RIJNDAEL supports any number of rounds between 10 and 14, with the number of rounds chosen as a function of the key length and block size. Intel's AES architecture is completely flexible to support these or any other number of rounds, simply by executing more or fewer AESENC / AESDEC instructions and key scheduling instructions.

Changing the key length or key schedule

RIJNDAEL supports any key length which is a multiple of 32 bits ranging between 128 and 256 bits. The AES instructions can readily support such key lengths (and any others) by simply executing more or fewer key scheduling steps. The AESKEYGENASSIST instruction can take any RCON value as the input's immediate byte. Furthermore, since the instruction set decouples the round keys (and their generation) from their use in the AESENC, AESENCLAST, AESDEC, AESDECLAST instructions, it would be easy to modify or replace the RIJNDAEL key schedule with a different variation, while keeping the benefits of the round instructions.

Changing the block size (e.g., RIJNDAEL-256)

RIJNDAEL supports any block size which is a multiple of 32 bits, from 128 to 256 bits. We explain here how the AES instructions can be used for supporting such block sizes.

To support RIJNDAEL with a block size larger than 128 bits, the RIJNDAEL state needs to be stored in two registers (if the block size is smaller than 256 bits, only part of the second register holds relevant data). For each round, bytes should be swapped between the registers and then shuffled in order to to account for the the appropriate ShiftRows transformation (which is different from the ShiftRows for AES). Following this, the AES round instructions can be applied, independently, to the registers that hold the parts of the state. Due to the high parallelism of this sequence, the result should be efficient.

For example, consider a 256 bits block. Here, the RIJNDAEL state occupies two full xmm registers, and the two halves of the round key need to be stored in two other xmm resgisters. The new AVX instruction VPBLENDVB can be used to swap 8 bytes between these registers (and store the results in two new registers). Then, additional shuffling is required to account for the difference in ShiftRows between the 256 and 128-bit versions of RIJNDAEL. After this is done, the RIJNDAEL round can be computed by applying two AES round instruction, using the appropriate registers that hold the halves of the round key.

Figure 30. Using the AES instructions to compute a 256-bit block size RINJDAEL round

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

void Rijndael256_encrypt (unsigned char *in,
                          unsigned char *out,
                          unsigned char *Key_Schedule,
                          unsigned long long length,
                          int number_of_rounds)
    {
    __m128i tmp1, tmp2, data1 ,data2;
    __m128i RIJNDAEL256_MASK =
                 _mm_set_epi32(0x03020d0c, 0x0f0e0908, 0x0b0a0504, 0x07060100);
    __m128i BLEND_MASK=
                 _mm_set_epi32(0x80000000, 0x80800000, 0x80800000, 0x80808000);
    __m128i *KS = (__m128i*)Key_Schedule;
    int i,j;

    for(i=0; i < length/32; i++) {              /* loop over the data blocks */
        data1 = _mm_loadu_si128(&((__m128i*)in)[i*2+0]);  /* load data block */
        data2 = _mm_loadu_si128(&((__m128i*)in)[i*2+1]);
        data1 = _mm_xor_si128(data1, KS[0]);        /* round 0 (initial xor) */
        data2 = _mm_xor_si128(data2, KS[1]);
        /* Do number_of_rounds-1 AES rounds */
        for(j=1; j < number_of_rounds; j++) {
            /*Blend to compensate for the shift rows shifts bytes between two
              128 bit blocks*/
            tmp1 = _mm_blendv_epi8(data1, data2, BLEND_MASK);
            tmp2 = _mm_blendv_epi8(data2, data1, BLEND_MASK);
            /*Shuffle that compensates for the additional shift in rows 3 and 4
              as opposed to rijndael128 (AES)*/
            tmp1 = _mm_shuffle_epi8(tmp1, RIJNDAEL256_MASK);
            tmp2 = _mm_shuffle_epi8(tmp2, RIJNDAEL256_MASK);
            /*This is the encryption step that includes sub bytes, shift rows,
              mix columns, xor with round key*/
            data1 = _mm_aesenc_si128(tmp1, KS[j*2]);
            data2 = _mm_aesenc_si128(tmp2, KS[j*2+1]);
        }
        tmp1 = _mm_blendv_epi8(data1, data2, BLEND_MASK);
        tmp2 = _mm_blendv_epi8(data2, data1, BLEND_MASK);
        tmp1 = _mm_shuffle_epi8(tmp1, RIJNDAEL256_MASK);
        tmp2 = _mm_shuffle_epi8(tmp2, RIJNDAEL256_MASK);
        tmp1 = _mm_aesenclast_si128(tmp1, KS[j*2+0]);       /*last AES round */
        tmp2 = _mm_aesenclast_si128(tmp2, KS[j*2+1]);
        _mm_storeu_si128(&((__m128i*)out)[i*2+0],tmp1);
        _mm_storeu_si128(&((__m128i*)out)[i*2+1],tmp2);
    }
}
```

Isolating the AES Transformations

Cipher designers may wish to build new cryptographic algorithms using components of AES. Such algorithms could benefit from the performance and side channel benefits of the AES instructions if they are designed to use the AES transformations.

In particular, the AES transformations can be useful building blocks for hash functions. For example, the use the MixColumns transformation provides rapid diffusion and the AES S-box is a good nonlinear mixer. Operating on large block sizes could be useful in constructing hash functions with a long digest size.

This concept is already being used for constructing some of the new Secure Hash Function algorithms that have been recently submitted to the NIST cryptographic hash Algorithm Competition. Two candidate algorithms (namely, SHAvite-3, and ECHO) that use the AES instrcution for achieving high performance survived in Round 2 of the competition. For details on the impact of the AES instructions on such hash algorithms, see "The Intel AES Instructions Set and the SHA-3 Candidates" (by R. Benadjila, O. Billet, S. Gueron. M. Robshaw) in Lecture Notes in Computer Science - Proceedings of Asiacrypt 2009, 5665:51-66 (2009).

We show here how combinations of the AES instructions can isolate the AES transformations.

Figure 31. Isolating the AES Transformations with Combinations of AES Instructions

```
Isolating ShiftRows
        PSHUFB xmm0, 0x0b06010c07020d08030e09040f0a0500
Isolating InvShiftRows
         PSHUFB xmm0, 0x0306090c0f0205080b0e0104070a0d00
Isolating MixColumns
        AESDECLAST xmm0, 0x00000000000000000000000000000000
        AESENC xmm0, 0x00000000000000000000000000000000
Isolating InvMixColumns
         AESENCLAST xmm0, 0x00000000000000000000000000000000
         AESDEC xmm0, 0x00000000000000000000000000000000
Isolating SubBytes
         PSHUFB xmm0, 0x0306090c0f0205080b0e0104070a0d00
         AESENCLAST xmm0, 0x00000000000000000000000000000000
Isolating InvSubBytes
         PSHUFB xmm0, 0x0b06010c07020d08030e09040f0a0500
         AESDECLAST xmm0, 0x00000000000000000000000000000000
```

The following code demonstrates show how the AES transformations can be isolated.

Figure 32. Isolating the AES Transformations (C Code)

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

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");
    }

/*****************************************************************************/
int main ()
    {
    __m128i ZERO = _mm_setzero_si128();
    __m128i ISOLATE_SROWS_MASK =
                _mm_set_epi32(0x0B06010C, 0x07020D08, 0x030E0904, 0x0F0A0500);
    __m128i ISOLATE_SBOX_MASK =
                _mm_set_epi32(0x0306090C, 0x0F020508, 0x0B0E0104, 0x070A0D00);
    __m128i Round_Key_0 =
                _mm_set_epi32(0x0f0e0d0c, 0x0b0a0908, 0x07060504, 0x03020100);
    __m128i Round_Key_1 =
                _mm_set_epi32(0xfe76abd6, 0xf178a6da, 0xfa72afd2, 0xfd74aad6);
    __m128i DATA =
                _mm_set_epi32(0xffeeddcc, 0xbbaa9988, 0x77665544, 0x33221100);
    __m128i temp1,temp2;

    printf ("Demonstrating the exposed transformations: \n");
    print_m128i_with_string("DATA:", DATA);
    print_m128i_with_string("Round Key 0:", Round_Key_0);

    temp1 = _mm_xor_si128(DATA, Round_Key_0);                 /* Round 0*/

    print_m128i_with_string("After Round 0:", temp1);
    print_m128i_with_string("Round Key 1:", Round_Key_1);

/* A "decomposed" encryption round, built from the individual transformations*/
    temp2 = _mm_shuffle_epi8(temp1, ISOLATE_SROWS_MASK);/* isolate ShiftRows */

    print_m128i_with_string("After ShiftRows:", temp2);

    temp2 = _mm_shuffle_epi8(temp2, ISOLATE_SBOX_MASK);  /* isolate SubBytes */
    temp2 = _mm_aesenclast_si128(temp2, ZERO);

    print_m128i_with_string("After SubBytes:", temp2);

    temp2 = _mm_aesdeclast_si128(temp2, ZERO);         /* isolate MixColumns */
    temp2 = _mm_aesenc_si128(temp2, ZERO);

    print_m128i_with_string("After MixColumns:", temp2);

    temp2 = _mm_xor_si128(temp2, Round_Key_1);        /* isolate AddRoundKey */

    print_m128i_with_string("After AddRoundKey:", temp2);

    temp1 = _mm_aesenc_si128(temp1, Round_Key_1);/* round 1 using instruction*/

    printf("\n");
    print_m128i_with_string("AES Round using exposed transformations:", temp2);
    print_m128i_with_string("AES round using AESENC instruction:", temp1);
    printf("\n");
    printf ("Going backwards using exposed inverse transformations: \n");

    temp2 = _mm_xor_si128(temp2, Round_Key_1);             /* Going Bakwards */

    print_m128i_with_string("After InvAddRoundKey:", temp2);
    temp2 = _mm_aesenclast_si128(temp2, ZERO);
    temp2 = _mm_aesdec_si128(temp2, ZERO);

    print_m128i_with_string("After InvMixColumns:", temp2);

    temp2 = _mm_shuffle_epi8(temp2, ISOLATE_SROWS_MASK);
    temp2 = _mm_aesdeclast_si128(temp2, ZERO);

    print_m128i_with_string("After InvSubBytes:", temp2);

    temp2 = _mm_shuffle_epi8(temp2, ISOLATE_SBOX_MASK);

    print_m128i_with_string("After InvShiftRows:", temp2);

    temp2 = _mm_xor_si128(temp2, Round_Key_0);

    print_m128i_with_string("Final:", temp2);
    printf ("Returned to initial state. \n");
    }
```

Figure 33. Isolating the AES Transformations – Code Results

```
Demonstrating the exposed transformations:
DATA:                                   [0x00112233445566778899aabbccddeeff]
Round Key 0:                            [0x000102030405060708090a0b0c0d0e0f]
After Round 0:                          [0x00102030405060708090a0b0c0d0e0f0]
Round Key 1:                            [0xd6aa74fdd2af72fadaa678f1d6ab76fe]

After ShiftRows:                        [0x0050a0f04090e03080d02070c01060b0]
After SubBytes:                         [0x6353e08c0960e104cd70b751bacad0e7]
After MixColumns:                       [0x5f72641557f5bc92f7be3b291db9f91a]
After AddRoundKey:                      [0x89d810e8855ace682d1843d8cb128fe4]

AES Round using exposed transformations:[0x89d810e8855ace682d1843d8cb128fe4]
AES round using AESENC instruction:     [0x89d810e8855ace682d1843d8cb128fe4]

Going backwards using exposed inverse transformations:
After InvAddRoundKey:                   [0x5f72641557f5bc92f7be3b291db9f91a]
After InvMixColumns:                    [0x6353e08c0960e104cd70b751bacad0e7]
After InvSubBytes:                      [0x0050a0f04090e03080d02070c01060b0]
After InvShiftRows:                     [0x00102030405060708090a0b0c0d0e0f0]
Final:                                  [0x00112233445566778899aabbccddeeff]
Returned to initial state.
```

On-the-Fly Key Expansion

The following two examples illustrate AES-128 encryption and decryption with on-the-fly key expansion. The input key for the encryption is the cipher key. For the decryption, the input key is the last round key (number 10 in this example). This example also demonstrates the use of a combination of instrcutions to isolate AES transofrmations. Here, the AES SBox is isolated and the key expansion is carried out without using the AESKEYGENASSIST instrcution.

Figure 34. AES128-ECB Encryption with On-the-Fly Key Expansion

```
#include <wmmintrin.h>
#include <smmintrin.h>
//unsigned char *userkey points to the cipher key
//unsigned char *data points to 16 bytes of data to be encrypted
void AES_128_ENCRYPT_on_the_fly (const unsigned char *userkey,
                                 const unsigned char *data)
    {

    __m128i temp1, temp2, temp3;
    __m128i block;
    __m128i shuffle_mask =
        _mm_set_epi32(0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d);
    __m128i rcon;
    int i;

    block = _mm_loadu_si128((__m128i*)&data[0]);
    temp1 = _mm_loadu_si128((__m128i*)userkey);
    rcon = _mm_set_epi32(1,1,1,1);

    block = _mm_xor_si128(block, temp1);

    for (i=1; i<=8; i++){
        temp2 = _mm_shuffle_epi8(temp1, shuffle_mask);
        temp2 = _mm_aesenclast_si128 (temp2,rcon);
        rcon = _mm_slli_epi32(rcon,1);
        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);
        block = _mm_aesenc_si128 (block, temp1);
        }
    rcon = _mm_set_epi32(0x1b,0x1b,0x1b,0x1b);
    temp2 = _mm_shuffle_epi8(temp1, shuffle_mask);
    temp2 = _mm_aesenclast_si128 (temp2,rcon);
    rcon = _mm_slli_epi32(rcon,1);
    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);
    block = _mm_aesenc_si128 (block, temp1);

    temp2 = _mm_shuffle_epi8(temp1, shuffle_mask);
    temp2 = _mm_aesenclast_si128 (temp2,rcon);
    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);
    block = _mm_aesenclast_si128 (block, temp1);

    _mm_storeu_si128((__m128i*)&data[0] ,block);
    }
```

Figure 35. AES128-ECB Decryption with On-the-Fly Key Expansion

```
#include <wmmintrin.h>
#include <smmintrin.h>
//unsigned char *userkey points to the last key of the encrypt key schedule
//unsigned char *data points to 16 bytes of data to be encrypted
void AES_128_DECRYPT_on_the_fly (const unsigned char *userkey,
                                 const unsigned char *data)
    {
    __m128i temp1, temp2, temp3, temp4;
    int mask = 0x0c0f0e0d;
    int con1 = 0x80, con2 = 0x36;
    __m128i shuffle_mask =
        _mm_set_epi32(0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d,0x0c0f0e0d);
    __m128i rcon;
    __m128i block;
    int i;

    rcon = _mm_set_epi32(0x1b,0x1b,0x1b,0x1b);

    temp1 = _mm_loadu_si128((__m128i*)userkey);
    block = _mm_loadu_si128((__m128i*)&data[0]);

    block = _mm_xor_si128(block, temp1);

    for (i=1; i<=2; i++){
        temp2 = _mm_slli_si128(temp1 , 4);
        temp2 = _mm_xor_si128(temp1, temp2);
        temp3 = _mm_shuffle_epi8(temp2, shuffle_mask);
        temp3 = _mm_aesenclast_si128 (temp3,rcon);
        temp1 = _mm_xor_si128(temp1,temp3);
        temp1 = (__m128i)_mm_blend_ps((__m128)temp1, (__m128)temp2, 14);
        temp2 = _mm_aesimc_si128(temp1);
        rcon = _mm_srli_epi32(rcon,1);
        block = _mm_aesdec_si128 (block, temp2);
        }
    rcon = _mm_set_epi32(1,1,1,1);
    for (i=3; i<10; i++){
        temp2 = _mm_slli_si128(temp1 , 4);
        temp2 = _mm_xor_si128(temp1, temp2);
        temp3 = _mm_shuffle_epi8(temp2, shuffle_mask);
        temp3 = _mm_aesenclast_si128 (temp3,rcon);
        temp1 = _mm_xor_si128(temp1,temp3);
        temp1 = (__m128i)_mm_blend_ps((__m128)temp1, (__m128)temp2, 14);
        temp2 = _mm_aesimc_si128(temp1);
        rcon = _mm_srli_epi32(rcon,1);
        block = _mm_aesdec_si128 (block, temp2);
        }
    temp2 = _mm_slli_si128(temp1 , 4);
    temp2 = _mm_xor_si128(temp1, temp2);
    temp3 = _mm_shuffle_epi8(temp2, shuffle_mask);
    temp3 = _mm_aesenclast_si128 (temp3,rcon);
    temp1 = _mm_xor_si128(temp1,temp3);
    temp1 = (__m128i)_mm_blend_ps((__m128)temp1, (__m128)temp2, 14);
    block = _mm_aesdeclast_si128 (block, temp1);

    _mm_storeu_si128((__m128i*)&data[0] ,block);
    }
```

Performance and Performance Optimization Guidelines

Expected Performance for Encryption and Decryption

The AES instructions provide a substantial performance speedup to bulk data encryption and decryption. When using parallelizable modes of operation, such as CBC decryption, CTR, and CTR-derived modes (GCM), XTS. The performance speedup could exceed an order of magnitude over software-only, lookup tables based AES implementations. In scenarios where pipelined operation is impossible, for example in CBC encryption, the performance speedup would still be significant, around 2 to 3 times over (unprotected) software implementation.

The Relative Cost of the Key Expansion

The AES architecture is optimized for security and performance in applications where many block encryptions are performed with the same key (e.g., disk or network encryption). For example, Microsoft's Bitlocker disk encryption application uses a single key for the whole volume. In these applications, the cost of the key expansion is amortized over many blocks, making the overhead of the key expansion marginal from the performance perspective.

Some less frequent applications require frequent key scheduling. For example, some random number generators may rekey frequently to achieve forward secrecy. One extreme example is a Davies-Meyer hashing construction, which uses a block cipher primitive as a compression function, and the cipher is re-keyed for each processed data block.

Although these are not the mainstream usage models of the AES instructions, we point out that the AESKEYGENASSIST and AESIMC instructions facilitate Key Expansion procedure which is lookup tables free, and faster than software only key expansion. In addition, we point out that unrolling of the key expansion code, which is provided in the previous sections, improves the key expansion performance. The AES256 case can also utilize the instruction AESENCLAST, for the sbox transformation, that is faster than using AESKEYGENASSIST.

Optimizing AES Software for Enhanced Performance in Parallel Modes of Operation

Perhaps the most significant performance optimization for encryptio/decryption using the AES instructions can be achieved by re-ordering the computations. This helps take better advantage of parallelism in parallel modes of operation such as ECB, CTR, and CBC-Decrypt (with the CBC-Encrypt serial mode being an exception). This section explains how it can be done.

The hardware that supports the four AES round instructions is pipelined. This allows independent AES instructions to be dispatched theoretically every 1-2 CPU clock cycle (depending on the micro architectural implementation), if data can be provided sufficiently fast. As a result, the AES throughput can be significantly enhanced for parallel modes of operation, if the "order of the loop" is reversed: instead of completing the encryption of one data block and then continuing to the subsequent block, it is preferable to write software sequences that compute one AES round on multiple blocks, using one Round Key, and only then continue to computing the subsequent round on for multiple blocks (using another round key). For such software optimization, one needs to choose the number of blocks that will be processed in parallel. This optimal parallelization parameter value depends on the scenario, for example on how many registers are available, and how many data blocks are to be (typically) processed.

Excessive pipelining does not provide performance benefit, and it also consumes registers that can be used for other purposes. Therefore, the tradeoff should be assessed by the developers depending on the application and the optimization targets.

In general, we recommend processing 4 or 8 blocks in parallel, to for optimized throughput. The speedup that can be gained is significant. For cases where the size, in blocks, of the processed buffer is not divisible by 4 (or by 8), the remainder blocks need to be handled separately.

In the following, we offer two examples. The first one is a function (C code snippet) that illustrates CBC decryption of 4 data blocks (the complete code was provided in the previous examples, and can be run with the proper choice of the `#define PARALLEL`). The second example is a C code function that encrypts 8 blocks in parallel, using ECB mode (the complete code was provided in the previous examples, and can be run with the proper choice of `#define EIGHT_BLOCKS`).

Figure 36. Parallelizing CBC Decrypt Function 4 Blocks at a Time

```
#include <wmmintrin.h>

void AES_CBC_decrypt_parallelize_4_blocks(const unsigned char *in,
                                          unsigned char *out,
                                          unsigned char ivec[16],
                                          unsigned long length,
                                          unsigned char *key_schedule,
                                          unsigned int nr)
    {
    __m128i data1,data2,data3,data4;
    __m128i feedback1,feedback2,feedback3,feedback4,last_in;
    int i,j;

    if (length%16)
        length = length/16 + 1;
    else length/=16;

    feedback1=_mm_loadu_si128 ((__m128i*)ivec);

    for(i=0; i < length/4; i++){
        data1=_mm_loadu_si128 (&((__m128i*)in)[i*4+0]);
        data2=_mm_loadu_si128 (&((__m128i*)in)[i*4+1]);
        data3=_mm_loadu_si128 (&((__m128i*)in)[i*4+2]);
        data4=_mm_loadu_si128 (&((__m128i*)in)[i*4+3]);

        feedback2=data1;
        feedback3=data2;
        feedback4=data3;
        last_in=data4;

        data1 = _mm_xor_si128 (data1,((__m128i*)key_schedule)[0]);
        data2 = _mm_xor_si128 (data2,((__m128i*)key_schedule)[0]);
        data3 = _mm_xor_si128 (data3,((__m128i*)key_schedule)[0]);
        data4 = _mm_xor_si128 (data4,((__m128i*)key_schedule)[0]);

        for(j=1; j < nr; j++){
            data1 = _mm_aesdec_si128 (data1,((__m128i*)key_schedule)[j]);
            data2 = _mm_aesdec_si128 (data2,((__m128i*)key_schedule)[j]);
            data3 = _mm_aesdec_si128 (data3,((__m128i*)key_schedule)[j]);
            data4 = _mm_aesdec_si128 (data4,((__m128i*)key_schedule)[j]);
        }

        data1 = _mm_aesdeclast_si128 (data1,((__m128i*)key_schedule)[j]);
        data2 = _mm_aesdeclast_si128 (data2,((__m128i*)key_schedule)[j]);
        data3 = _mm_aesdeclast_si128 (data3,((__m128i*)key_schedule)[j]);
        data4 = _mm_aesdeclast_si128 (data4,((__m128i*)key_schedule)[j]);

        data1 = _mm_xor_si128 (data1,feedback1);
        data2 = _mm_xor_si128 (data2,feedback2);
        data3 = _mm_xor_si128 (data3,feedback3);
        data4 = _mm_xor_si128 (data4,feedback4);

        _mm_storeu_si128 (&((__m128i*)out)[i*4+0],data1);
        _mm_storeu_si128 (&((__m128i*)out)[i*4+1],data2);
        _mm_storeu_si128 (&((__m128i*)out)[i*4+2],data3);
        _mm_storeu_si128 (&((__m128i*)out)[i*4+3],data4);

        feedback1=last_in;
        }

    for(j=i*4; j < length; j++){
        data1=_mm_loadu_si128 (&((__m128i*)in)[j]);
        last_in=data1;
        data1 = _mm_xor_si128 (data1,((__m128i*)key_schedule)[0]);
        for(i=1; i < nr; i++){
            data1 = _mm_aesdec_si128 (data1,((__m128i*)key_schedule)[i]);
        }
        data1 = _mm_aesdeclast_si128 (data1,((__m128i*)key_schedule)[i]);
        data1 = _mm_xor_si128 (data1,feedback1);
        _mm_storeu_si128 (&((__m128i*)out)[j],data1);
        feedback1=last_in;
        }
    }
```

Figure 37.CBC Encrypt Four Buffers in Parallel – C function

```
#include <wmmintrin.h>

void AES_CBC_encrypt_parallelize_4_blocks(const unsigned char *in,
                                          unsigned char *out,
                                          unsigned char ivec1[16],
                                          unsigned char ivec2[16],
                                          unsigned char ivec3[16],
                                          unsigned char ivec4[16],
                                          unsigned long length,
                                          const unsigned char *key,
                                          int nr)
    {
    __m128i feedback1,feedback2,feedback3,feedback4;
    __m128i data1,data2,data3,data4;
    int i,j;

    feedback1=_mm_loadu_si128 ((__m128i*)ivec1);
    feedback2=_mm_loadu_si128 ((__m128i*)ivec2);
    feedback3=_mm_loadu_si128 ((__m128i*)ivec3);
    feedback4=_mm_loadu_si128 ((__m128i*)ivec4);
    for(i=0; i < length/16/4; i++){
        data1 = _mm_loadu_si128 (&((__m128i*)in)[i*4+0]);
        data2 = _mm_loadu_si128 (&((__m128i*)in)[i*4+1]);
        data3 = _mm_loadu_si128 (&((__m128i*)in)[i*4+2]);
        data4 = _mm_loadu_si128 (&((__m128i*)in)[i*4+3]);

        feedback1 = _mm_xor_si128 (data1,feedback1);
        feedback2 = _mm_xor_si128 (data2,feedback2);
        feedback3 = _mm_xor_si128 (data3,feedback3);
        feedback4 = _mm_xor_si128 (data4,feedback4);

        feedback1 = _mm_xor_si128 (feedback1,((__m128i*)key)[0]);
        feedback2 = _mm_xor_si128 (feedback2,((__m128i*)key)[0]);
        feedback3 = _mm_xor_si128 (feedback3,((__m128i*)key)[0]);
        feedback4 = _mm_xor_si128 (feedback4,((__m128i*)key)[0]);

        for(j=1; j <nr; j++){
            feedback1 = _mm_aesenc_si128 (feedback1,((__m128i*)key)[j]);
            feedback2 = _mm_aesenc_si128 (feedback2,((__m128i*)key)[j]);
            feedback3 = _mm_aesenc_si128 (feedback3,((__m128i*)key)[j]);
            feedback4 = _mm_aesenc_si128 (feedback4,((__m128i*)key)[j]);
        }

        feedback1 = _mm_aesenclast_si128 (feedback1,((__m128i*)key)[j]);
        feedback2 = _mm_aesenclast_si128 (feedback2,((__m128i*)key)[j]);
        feedback3 = _mm_aesenclast_si128 (feedback3,((__m128i*)key)[j]);
        feedback4 = _mm_aesenclast_si128 (feedback4,((__m128i*)key)[j]);

        _mm_storeu_si128 (&((__m128i*)out)[i*4+0],feedback1);
        _mm_storeu_si128 (&((__m128i*)out)[i*4+1],feedback2);
        _mm_storeu_si128 (&((__m128i*)out)[i*4+2],feedback3);
        _mm_storeu_si128 (&((__m128i*)out)[i*4+3],feedback4);
        }
    }
```

An AES Library

This chapter provides high performance functions, written in assembly (AT&T syntax), implementing AES-128, AES-192, AES-256 in ECB, CBC, and CTR modes. A separate section provides test functions that can be used for generating executables that can be run and measured for performance. The outputs of the test runs are also provided below. These functions have a very similar or identical interface to that of OpenSSL (as explained above).

Key Expansion

Figure 38. Unrolled Key Expansion Decrypt using InvMixColumns

```
//void AES_Key_Expansion_Decrypt(const unsigned char *encrypt_schedule,
//                               unsigned char *decrypt_schedule,
//                               int number_of_rounds)
.align    16,0x90
.globl AES_Key_Expansion_Decrypt
```

```
AES_Key_Expansion_Decrypt:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %edx

        movslq    %edx, %rdx
        movq      %rdx, %rax
        shlq      $4, %rax
        cmpq      $10, %rdx
        movdqa    (%rax,%rdi), %xmm0
        movdqa    %xmm0, (%rsi)

        aesimc    -16(%rax,%rdi), %xmm1
        aesimc    -32(%rax,%rdi), %xmm2
        aesimc    -48(%rax,%rdi), %xmm3
        aesimc    -64(%rax,%rdi), %xmm4
        movdqa    %xmm1, 16(%rsi)
        movdqa    %xmm2, 32(%rsi)
        movdqa    %xmm3, 48(%rsi)
        movdqa    %xmm4, 64(%rsi)

        aesimc    -80(%rax,%rdi),  %xmm5
        aesimc    -96(%rax,%rdi),  %xmm6
        aesimc    -112(%rax,%rdi), %xmm7
        aesimc    -128(%rax,%rdi), %xmm8
        movdqa    %xmm5, 80(%rsi)
        movdqa    %xmm6, 96(%rsi)
        movdqa    %xmm7, 112(%rsi)
        movdqa    %xmm8, 128(%rsi)

        aesimc    -144(%rax,%rdi), %xmm9
        movdqa    %xmm9, 144(%rsi)

        jle       END_DEC
        cmpq      $12, %rdx

        aesimc    -160(%rax,%rdi), %xmm0
        aesimc    -176(%rax,%rdi), %xmm1
        movdqa    %xmm0, 160(%rsi)
        movdqa    %xmm1, 176(%rsi)
        jle       END_DEC

        aesimc    -192(%rax,%rdi), %xmm0
        aesimc    -208(%rax,%rdi), %xmm1
        movdqa    %xmm0, 192(%rsi)
        movdqa    %xmm1, 208(%rsi)

END_DEC:
        movdqa    (%rdi), %xmm0
        movdqa    %xmm0, (%rax,%rsi)
        ret
```

Figure 39. AES-128 Key Expansion: Assembly Code

```
//void AES_128_Key_Expansion(const unsigned char* userkey,
//                           unsigned char* key_schedule);

.align    16,0x90
.globl AES_128_Key_Expansion
AES_128_Key_Expansion:
# parameter 1: %rdi
# parameter 2: %rsi
        movl      $10, 240(%rsi)
        movdqu    (%rdi), %xmm1
        movdqa    %xmm1, (%rsi)

ASSISTS:
        aeskeygenassist $1, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 16(%rsi)
        aeskeygenassist $2, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 32(%rsi)
        aeskeygenassist $4, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 48(%rsi)
        aeskeygenassist $8, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 64(%rsi)
        aeskeygenassist $16, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 80(%rsi)
        aeskeygenassist $32, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 96(%rsi)
        aeskeygenassist $64, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 112(%rsi)
        aeskeygenassist $0x80, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 128(%rsi)
        aeskeygenassist $0x1b, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 144(%rsi)
        aeskeygenassist $0x36, %xmm1, %xmm2
        call PREPARE_ROUNDKEY_128
        movdqa %xmm1, 160(%rsi)
        ret

PREPARE_ROUNDKEY_128:
             pshufd $255, %xmm2, %xmm2
             movdqa %xmm1, %xmm3
             pslldq $4, %xmm3
             pxor %xmm3, %xmm1
             pslldq $4, %xmm3
             pxor %xmm3, %xmm1
             pslldq $4, %xmm3
             pxor %xmm3, %xmm1
             pxor %xmm2, %xmm1
             ret
```

Figure 40. AES-192 Key Expansion: Assembly Code

```
//void AES_192_Key_Expansion (const unsigned char *userkey,
//                            unsigned char *key)
.globl AES_192_Key_Expansion
AES_192_Key_Expansion:
# parameter 1: %rdi
# parameter 2: %rsi
```

```
        movdqu (%rdi), %xmm1
        movdqu 16(%rdi), %xmm3
        movdqa %xmm1, (%rsi)
        movdqa %xmm3, %xmm5

        aeskeygenassist $0x1, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        shufpd $0, %xmm1, %xmm5
        movdqa %xmm5, 16(%rsi)
        movdqa %xmm1, %xmm6
        shufpd $1, %xmm3, %xmm6
        movdqa %xmm6, 32(%rsi)

        aeskeygenassist $0x2, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        movdqa %xmm1, 48(%rsi)
        movdqa %xmm3, %xmm5

        aeskeygenassist $0x4, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        shufpd $0, %xmm1, %xmm5
        movdqa %xmm5, 64(%rsi)
        movdqa %xmm1, %xmm6
        shufpd $1, %xmm3, %xmm6
        movdqa %xmm6, 80(%rsi)

        aeskeygenassist $0x8, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        movdqa %xmm1, 96(%rsi)
        movdqa %xmm3, %xmm5

        aeskeygenassist $0x10, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        shufpd $0, %xmm1, %xmm5
        movdqa %xmm5, 112(%rsi)
        movdqa %xmm1, %xmm6
        shufpd $1, %xmm3, %xmm6
        movdqa %xmm6, 128(%rsi)

        aeskeygenassist $0x20, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        movdqa %xmm1, 144(%rsi)
        movdqa %xmm3, %xmm5

        aeskeygenassist $0x40, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        shufpd $0, %xmm1, %xmm5
        movdqa %xmm5, 160(%rsi)
        movdqa %xmm1, %xmm6
        shufpd $1, %xmm3, %xmm6
        movdqa %xmm6, 176(%rsi)

        aeskeygenassist $0x80, %xmm3, %xmm2
        call PREPARE_ROUNDKEY_192
        movdqa %xmm1, 192(%rsi)
        movdqa %xmm3, 208(%rsi)
        ret

PREPARE_ROUNDKEY_192:
        pshufd $0x55, %xmm2, %xmm2
        movdqu %xmm1, %xmm4
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm1
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm1
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm1
        pxor   %xmm2, %xmm1
        pshufd $0xff, %xmm1, %xmm2
        movdqu %xmm3, %xmm4
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm3
        pxor   %xmm2, %xmm3
        ret
```

Figure 41. AES-256 Key Expansion: Assembly Code

```
//void AES_256_Key_Expansion (const unsigned char *userkey,
//                             unsigned char *key)
.globl AES_256_Key_Expansion
AES_256_Key_Expansion:
# parameter 1: %rdi
# parameter 2: %rsi

        movdqu (%rdi), %xmm1
        movdqu 16(%rdi), %xmm3
        movdqa %xmm1, (%rsi)
        movdqa %xmm3, 16(%rsi)

        aeskeygenassist $0x1, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 32(%rsi)
        aeskeygenassist $0x0, %xmm1, %xmm2
        call MAKE_RK256_b
        movdqa %xmm3, 48(%rsi)
        aeskeygenassist $0x2, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 64(%rsi)
        aeskeygenassist $0x0, %xmm1, %xmm2
        call MAKE_RK256_b
        movdqa %xmm3, 80(%rsi)
        aeskeygenassist $0x4, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 96(%rsi)
        aeskeygenassist $0x0, %xmm1, %xmm2
        call MAKE_RK256_b
        movdqa %xmm3, 112(%rsi)
        aeskeygenassist $0x8, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 128(%rsi)
        aeskeygenassist $0x0, %xmm1, %xmm2
        call MAKE_RK256_b
        movdqa %xmm3, 144(%rsi)
        aeskeygenassist $0x10, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 160(%rsi)
        aeskeygenassist $0x0, %xmm1, %xmm2
        call MAKE_RK256_b
        movdqa %xmm3, 176(%rsi)
        aeskeygenassist $0x20, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 192(%rsi)
        aeskeygenassist $0x0, %xmm1, %xmm2
        call MAKE_RK256_b
        movdqa %xmm3, 208(%rsi)
        aeskeygenassist $0x40, %xmm3, %xmm2
        call MAKE_RK256_a
        movdqa %xmm1, 224(%rsi)

        ret

MAKE_RK256_a:
        pshufd $0xff, %xmm2, %xmm2
        movdqa %xmm1, %xmm4
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm1
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm1
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm1
        pxor   %xmm2, %xmm1
        ret

MAKE_RK256_b:
        pshufd $0xaa, %xmm2, %xmm2
        movdqa %xmm3, %xmm4
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm3
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm3
        pslldq $4, %xmm4
        pxor   %xmm4, %xmm3
        pxor   %xmm2, %xmm3
        ret
```

Figure 42. A Universal Key Expansion(C code)

```
/*
A function with OpenSSL interface (using AES_KEY struct), to call the other key-
length specific key expansion functions
*/
#include <wmmintrin.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;


int AES_set_encrypt_key (const unsigned char *userKey,
                         const int bits,
                         AES_KEY *key)
    {
    if (!userKey || !key)
             return -1;
       if (bits == 128)
        {
```

```
        AES_128_Key_Expansion (userKey,key);
        key->nr = 10;
        return 0;
        }
    else if (bits == 192)
        {
        AES_192_Key_Expansion (userKey,key);
        key->nr = 12;
        return 0;
        }
    else if (bits == 256)
        {
        AES_256_Key_Expansion (userKey,key);
        key->nr = 14;
        return 0;
        }
    return -2;
    }

int AES_set_decrypt_key (const unsigned char *userKey,
                         const int bits,
                         AES_KEY *key)
    {
    int i,nr;;
    AES_KEY temp_key;
    __m128i *Key_Schedule = (__m128i*)key->KEY;
    __m128i *Temp_Key_Schedule = (__m128i*)temp_key.KEY;
    if (!userKey || !key)
             return -1;

    if (AES_set_encrypt_key(userKey,bits,&temp_key) == -2)
        return -2;

    nr = temp_key.nr;
    key->nr = nr;

    Key_Schedule[nr] = Temp_Key_Schedule[0];
    Key_Schedule[nr-1] = _mm_aesimc_si128(Temp_Key_Schedule[1]);
    Key_Schedule[nr-2] = _mm_aesimc_si128(Temp_Key_Schedule[2]);
    Key_Schedule[nr-3] = _mm_aesimc_si128(Temp_Key_Schedule[3]);
    Key_Schedule[nr-4] = _mm_aesimc_si128(Temp_Key_Schedule[4]);
    Key_Schedule[nr-5] = _mm_aesimc_si128(Temp_Key_Schedule[5]);
    Key_Schedule[nr-6] = _mm_aesimc_si128(Temp_Key_Schedule[6]);
    Key_Schedule[nr-7] = _mm_aesimc_si128(Temp_Key_Schedule[7]);
    Key_Schedule[nr-8] = _mm_aesimc_si128(Temp_Key_Schedule[8]);
    Key_Schedule[nr-9] = _mm_aesimc_si128(Temp_Key_Schedule[9]);
    if(nr>10){
        Key_Schedule[nr-10] = _mm_aesimc_si128(Temp_Key_Schedule[10]);
        Key_Schedule[nr-11] = _mm_aesimc_si128(Temp_Key_Schedule[11]);
        }
    if(nr>12){
        Key_Schedule[nr-12] = _mm_aesimc_si128(Temp_Key_Schedule[12]);
        Key_Schedule[nr-13] = _mm_aesimc_si128(Temp_Key_Schedule[13]);
        }
    Key_Schedule[0] = Temp_Key_Schedule[nr];
    return 0;
    }
```

ECB MODE

Figure 43. The AES Encryption Parallelizing 4 Blocks (AT&T Assembly Function)

```
//void AES_ECB_encrypt (const unsigned char *in,
//                       unsigned char *out,
//                       unsigned long length,
//                       const unsigned char *KS,
//                       int nr)

.globl AES_ECB_encrypt
AES_ECB_encrypt:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
# parameter 4: %rcx
# parameter 5: %r8d
        movq      %rdx, %r10
        shrq      $4, %rdx
        shlq      $60, %r10
        je        NO_PARTS_4
        addq      $1, %rdx
NO_PARTS_4:
        movq      %rdx, %r10
        shlq      $62, %r10
        shrq      $62, %r10
        shrq      $2, %rdx
        je        REMAINDER_4
        subq      $64, %rsi
LOOP_4:
        movdqu    (%rdi), %xmm1
        movdqu    16(%rdi), %xmm2
        movdqu    32(%rdi), %xmm3
        movdqu    48(%rdi), %xmm4
        movdqa    (%rcx), %xmm9
        movdqa    16(%rcx), %xmm10
        movdqa    32(%rcx), %xmm11
        movdqa    48(%rcx), %xmm12
        pxor      %xmm9, %xmm1
        pxor      %xmm9, %xmm2
        pxor      %xmm9, %xmm3
        pxor      %xmm9, %xmm4
        aesenc    %xmm10, %xmm1
        aesenc    %xmm10, %xmm2
        aesenc    %xmm10, %xmm3
        aesenc    %xmm10, %xmm4
        aesenc    %xmm11, %xmm1
        aesenc    %xmm11, %xmm2
        aesenc    %xmm11, %xmm3
        aesenc    %xmm11, %xmm4
        aesenc    %xmm12, %xmm1
        aesenc    %xmm12, %xmm2
        aesenc    %xmm12, %xmm3
        aesenc    %xmm12, %xmm4
        movdqa    64(%rcx), %xmm9
        movdqa    80(%rcx), %xmm10
        movdqa    96(%rcx), %xmm11
        movdqa    112(%rcx), %xmm12
        aesenc    %xmm9, %xmm1
        aesenc    %xmm9, %xmm2
        aesenc    %xmm9, %xmm3
        aesenc    %xmm9, %xmm4
        aesenc    %xmm10, %xmm1
        aesenc    %xmm10, %xmm2
        aesenc    %xmm10, %xmm3
        aesenc    %xmm10, %xmm4
        aesenc    %xmm11, %xmm1
        aesenc    %xmm11, %xmm2
        aesenc    %xmm11, %xmm3
        aesenc    %xmm11, %xmm4
        aesenc    %xmm12, %xmm1
        aesenc    %xmm12, %xmm2
        aesenc    %xmm12, %xmm3
        aesenc    %xmm12, %xmm4
        movdqa    128(%rcx), %xmm9
        movdqa    144(%rcx), %xmm10
        movdqa    160(%rcx), %xmm11
        cmpl      $12, %r8d
        aesenc    %xmm9, %xmm1
        aesenc    %xmm9, %xmm2
        aesenc    %xmm9, %xmm3
        aesenc    %xmm9, %xmm4
        aesenc    %xmm10, %xmm1
        aesenc    %xmm10, %xmm2
        aesenc    %xmm10, %xmm3
        aesenc    %xmm10, %xmm4
        jb        LAST_4
        movdqa    160(%rcx), %xmm9
        movdqa    176(%rcx), %xmm10
        movdqa    192(%rcx), %xmm11
        cmpl      $14, %r8d
        aesenc    %xmm9, %xmm1
        aesenc    %xmm9, %xmm2
        aesenc    %xmm9, %xmm3
        aesenc    %xmm9, %xmm4
        aesenc    %xmm10, %xmm1
        aesenc    %xmm10, %xmm2
        aesenc    %xmm10, %xmm3
        aesenc    %xmm10, %xmm4
        jb        LAST_4
        movdqa    192(%rcx), %xmm9
        movdqa    208(%rcx), %xmm10
        movdqa    224(%rcx), %xmm11
        aesenc    %xmm9, %xmm1
        aesenc    %xmm9, %xmm2
        aesenc    %xmm9, %xmm3
        aesenc    %xmm9, %xmm4
        aesenc    %xmm10, %xmm1
        aesenc    %xmm10, %xmm2
        aesenc    %xmm10, %xmm3
        aesenc    %xmm10, %xmm4
LAST_4:
        addq      $64, %rdi
        addq      $64, %rsi
        decq      %rdx
        aesenclast %xmm11, %xmm1
        aesenclast %xmm11, %xmm2
        aesenclast %xmm11, %xmm3
        aesenclast %xmm11, %xmm4
        movdqu    %xmm1, (%rsi)
        movdqu    %xmm2, 16(%rsi)
        movdqu    %xmm3, 32(%rsi)
        movdqu    %xmm4, 48(%rsi)
        jne       LOOP_4
        addq      $64, %rsi
REMAINDER_4:
        cmpq      $0, %r10
        je        END_4
LOOP_4_2:
        movdqu    (%rdi), %xmm1
        addq      $16, %rdi
        pxor      (%rcx), %xmm1
        movdqu    160(%rcx), %xmm2
        aesenc    16(%rcx), %xmm1
        aesenc    32(%rcx), %xmm1
        aesenc    48(%rcx), %xmm1
        aesenc    64(%rcx), %xmm1
        aesenc    80(%rcx), %xmm1
        aesenc    96(%rcx), %xmm1
        aesenc    112(%rcx), %xmm1
        aesenc    128(%rcx), %xmm1
        aesenc    144(%rcx), %xmm1
        cmpl      $12, %r8d
        jb        LAST_4_2
        movdqu    192(%rcx), %xmm2
        aesenc    160(%rcx), %xmm1
        aesenc    176(%rcx), %xmm1
        cmpl      $14, %r8d
        jb        LAST_4_2
        movdqu    224(%rcx), %xmm2
        aesenc    192(%rcx), %xmm1
        aesenc    208(%rcx), %xmm1
LAST_4_2:
        aesenclast %xmm2, %xmm1
        movdqu    %xmm1, (%rsi)
        addq      $16, %rsi
        decq      %r10
        jne       LOOP_4_2
END_4:
        ret
```

Figure 44. The AES Decryption Parallelizing 4 Blocks (AT&T Assembly Function)

```
//void AES_ECB_decrypt (const unsigned char *in,
//                      unsigned char *out,
//                      unsigned long length,
//                      const unsigned char *KS,
//                      int nr)

.globl AES_ECB_decrypt
AES_ECB_decrypt:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
# parameter 4: %rcx
# parameter 5: %r8d
        movq      %rdx, %r10
        shrq   $4, %rdx
        shlq      $60, %r10
        je            DNO_PARTS_4
        addq      $1, %rdx
DNO_PARTS_4:
        movq      %rdx, %r10
        shlq      $62, %r10
        shrq      $62, %r10
        shrq      $2, %rdx
        je        DREMAINDER_4
        subq      $64, %rsi
DLOOP_4:
        movdqu    (%rdi), %xmm1
        movdqu    16(%rdi), %xmm2
        movdqu    32(%rdi), %xmm3
        movdqu    48(%rdi), %xmm4
        movdqa    (%rcx), %xmm9
        movdqa    16(%rcx), %xmm10
        movdqa    32(%rcx), %xmm11
        movdqa    48(%rcx), %xmm12
        pxor      %xmm9, %xmm1
        pxor      %xmm9, %xmm2
        pxor      %xmm9, %xmm3
        pxor      %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        aesdec    %xmm11, %xmm1
        aesdec    %xmm11, %xmm2
        aesdec    %xmm11, %xmm3
        aesdec    %xmm11, %xmm4
        aesdec    %xmm12, %xmm1
        aesdec    %xmm12, %xmm2
        aesdec    %xmm12, %xmm3
        aesdec    %xmm12, %xmm4
        movdqa    64(%rcx), %xmm9
        movdqa    80(%rcx), %xmm10
        movdqa    96(%rcx), %xmm11
        movdqa    112(%rcx), %xmm12
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        aesdec    %xmm11, %xmm1
        aesdec    %xmm11, %xmm2
        aesdec    %xmm11, %xmm3
        aesdec    %xmm11, %xmm4
        aesdec    %xmm12, %xmm1
        aesdec    %xmm12, %xmm2
        aesdec    %xmm12, %xmm3
        aesdec    %xmm12, %xmm4
        movdqa    128(%rcx), %xmm9
        movdqa    144(%rcx), %xmm10
        movdqa    160(%rcx), %xmm11
        cmpl      $12, %r8d
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        jb        DLAST_4
        movdqa    160(%rcx), %xmm9
        movdqa    176(%rcx), %xmm10
        movdqa    192(%rcx), %xmm11
        cmpl      $14, %r8d
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        jb        DLAST_4
        movdqa    192(%rcx), %xmm9
        movdqa    208(%rcx), %xmm10
        movdqa    224(%rcx), %xmm11
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
DLAST_4:
        addq      $64, %rdi
        addq      $64, %rsi
        decq      %rdx
        aesdeclast %xmm11, %xmm1
        aesdeclast %xmm11, %xmm2
        aesdeclast %xmm11, %xmm3
        aesdeclast %xmm11, %xmm4
        movdqu    %xmm1, (%rsi)
        movdqu    %xmm2, 16(%rsi)
        movdqu    %xmm3, 32(%rsi)
        movdqu    %xmm4, 48(%rsi)
        jne       DLOOP_4
        addq      $64, %rsi
DREMAINDER_4:
        cmpq      $0, %r10
        je        DEND_4
DLOOP_4_2:
        movdqu    (%rdi), %xmm1
        addq      $16, %rdi
        pxor      (%rcx), %xmm1
        movdqu    160(%rcx), %xmm2
        cmpl      $12, %r8d
        aesdec    16(%rcx), %xmm1
        aesdec    32(%rcx), %xmm1
        aesdec    48(%rcx), %xmm1
        aesdec    64(%rcx), %xmm1
        aesdec    80(%rcx), %xmm1
        aesdec    96(%rcx), %xmm1
        aesdec    112(%rcx), %xmm1
        aesdec    128(%rcx), %xmm1
        aesdec    144(%rcx), %xmm1
        jb        DLAST_4_2
        cmpl      $14, %r8d
        movdqu    192(%rcx), %xmm2
        aesdec    160(%rcx), %xmm1
        aesdec    176(%rcx), %xmm1
        jb        DLAST_4_2
        movdqu    224(%rcx), %xmm2
        aesdec    192(%rcx), %xmm1
        aesdec    208(%rcx), %xmm1
DLAST_4_2:
        aesdeclast %xmm2, %xmm1
        movdqu    %xmm1, (%rsi)
        addq      $16, %rsi
        decq      %r10
        jne       DLOOP_4_2
DEND_4:
        ret
```

CBC MODE

Figure 45. CBC Encryption of 1 Block at a Time (AT&T Assembly Function)

```
//AES_CBC_encrypt (const unsigned char *in,
//                 unsigned char *out,
//                 unsigned char ivec[16],
//                 unsigned long length,
//                 const unsigned char *KS,
//                 int nr)

.globl AES_CBC_encrypt
AES_CBC_encrypt:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
# parameter 4: %rcx
# parameter 5: %r8
# parameter 6: %r9d

        movq      %rcx, %r10
        shrq      $4, %rcx
        shlq      $60, %r10
        je        NO_PARTS
        addq      $1, %rcx
NO_PARTS:
        subq      $16, %rsi
        movdqa    (%rdx), %xmm1
LOOP:
        pxor      (%rdi), %xmm1
        pxor      (%r8),  %xmm1
        addq      $16,%rsi
        addq      $16,%rdi
        cmpl      $12, %r9d
        aesenc    16(%r8),%xmm1
        aesenc    32(%r8),%xmm1
        aesenc    48(%r8),%xmm1
        aesenc    64(%r8),%xmm1
        aesenc    80(%r8),%xmm1
        aesenc    96(%r8),%xmm1
        aesenc    112(%r8),%xmm1
        aesenc    128(%r8),%xmm1
        aesenc    144(%r8),%xmm1
        movdqa    160(%r8),%xmm2
        jb        LAST
        cmpl      $14, %r9d
        aesenc    160(%r8),%xmm1
        aesenc    176(%r8),%xmm1
        movdqa    192(%r8),%xmm2
        jb        LAST
        aesenc    192(%r8),%xmm1
        aesenc    208(%r8),%xmm1
        movdqa    224(%r8),%xmm2
LAST:
        decq       %rcx
        aesenclast %xmm2,%xmm1
        movdqu     %xmm1,(%rsi)

        jne        LOOP
        ret
```

Figure 46. CBC Decryption Parallelizing 4 Blocks (AT&T Assembly Function)

```
//AES_CBC_decrypt (const unsigned char *in,
//                 unsigned char *out,
//                 unsigned char ivec[16],
//                 unsigned long length,
//                 const unsigned char *KS,
//                 int nr)

.globl AES_CBC_decrypt
AES_CBC_decrypt:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
# parameter 4: %rcx
# parameter 5: %r8
# parameter 6: %r9d

        movq      %rcx, %r10
        shrq   $4, %rcx
        shlq      $60, %r10
        je            DNO_PARTS_4
        addq      $1, %rcx
DNO_PARTS_4:
        movq      %rcx, %r10
        shlq      $62, %r10
        shrq      $62, %r10
        shrq      $2, %rcx
        movdqu    (%rdx),%xmm5
        je        DREMAINDER_4
        subq      $64, %rsi
DLOOP_4:
        movdqu    (%rdi), %xmm1
        movdqu    16(%rdi), %xmm2
        movdqu    32(%rdi), %xmm3
        movdqu    48(%rdi), %xmm4
        movdqa    %xmm1, %xmm6
        movdqa    %xmm2, %xmm7
        movdqa    %xmm3, %xmm8
        movdqa    %xmm4, %xmm15
        movdqa    (%r8), %xmm9
        movdqa    16(%r8), %xmm10
        movdqa    32(%r8), %xmm11
        movdqa    48(%r8), %xmm12
        pxor      %xmm9, %xmm1
        pxor      %xmm9, %xmm2
        pxor      %xmm9, %xmm3
        pxor      %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        aesdec    %xmm11, %xmm1
        aesdec    %xmm11, %xmm2
        aesdec    %xmm11, %xmm3
        aesdec    %xmm11, %xmm4
        aesdec    %xmm12, %xmm1
        aesdec    %xmm12, %xmm2
        aesdec    %xmm12, %xmm3
        aesdec    %xmm12, %xmm4
        movdqa    64(%r8), %xmm9
        movdqa    80(%r8), %xmm10
        movdqa    96(%r8), %xmm11
        movdqa    112(%r8), %xmm12
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        aesdec    %xmm11, %xmm1
        aesdec    %xmm11, %xmm2
        aesdec    %xmm11, %xmm3
        aesdec    %xmm11, %xmm4
        aesdec    %xmm12, %xmm1
        aesdec    %xmm12, %xmm2
        aesdec    %xmm12, %xmm3
        aesdec    %xmm12, %xmm4
        movdqa    128(%r8), %xmm9
        movdqa    144(%r8), %xmm10
        movdqa    160(%r8), %xmm11
        cmpl      $12, %r9d
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        jb        DLAST_4
        movdqa    160(%r8), %xmm9
        movdqa    176(%r8), %xmm10
        movdqa    192(%r8), %xmm11
        cmpl      $14, %r9d
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
        jb        DLAST_4
        movdqa    192(%r8), %xmm9
        movdqa    208(%r8), %xmm10
        movdqa    224(%r8), %xmm11
        aesdec    %xmm9, %xmm1
        aesdec    %xmm9, %xmm2
        aesdec    %xmm9, %xmm3
        aesdec    %xmm9, %xmm4
        aesdec    %xmm10, %xmm1
        aesdec    %xmm10, %xmm2
        aesdec    %xmm10, %xmm3
        aesdec    %xmm10, %xmm4
DLAST_4:
        addq      $64, %rdi
        addq      $64, %rsi
        decq      %rcx
        aesdeclast %xmm11, %xmm1
        aesdeclast %xmm11, %xmm2
        aesdeclast %xmm11, %xmm3
        aesdeclast %xmm11, %xmm4
        pxor    %xmm5 ,%xmm1
        pxor    %xmm6 ,%xmm2
        pxor    %xmm7 ,%xmm3
        pxor    %xmm8 ,%xmm4
        movdqu    %xmm1, (%rsi)
        movdqu    %xmm2, 16(%rsi)
        movdqu    %xmm3, 32(%rsi)
        movdqu    %xmm4, 48(%rsi)
        movdqa    %xmm15,%xmm5
        jne       DLOOP_4
        addq      $64, %rsi
DREMAINDER_4:
        cmpq      $0, %r10
        je        DEND_4
DLOOP_4_2:
        movdqu    (%rdi), %xmm1
        movdqa    %xmm1 ,%xmm15
        addq      $16, %rdi
        pxor      (%r8), %xmm1
        movdqu    160(%r8), %xmm2
        cmpl      $12, %r9d
        aesdec    16(%r8), %xmm1
        aesdec    32(%r8), %xmm1
        aesdec    48(%r8), %xmm1
        aesdec    64(%r8), %xmm1
        aesdec    80(%r8), %xmm1
        aesdec    96(%r8), %xmm1
        aesdec    112(%r8), %xmm1
        aesdec    128(%r8), %xmm1
        aesdec    144(%r8), %xmm1
        jb        DLAST_4_2
        movdqu    192(%r8), %xmm2
        cmpl      $14, %r9d
        aesdec    160(%r8), %xmm1
        aesdec    176(%r8), %xmm1
        jb        DLAST_4_2
        movdqu    224(%r8), %xmm2
        aesdec    192(%r8), %xmm1
        aesdec    208(%r8), %xmm1
DLAST_4_2:
        aesdeclast %xmm2, %xmm1
        pxor      %xmm5, %xmm1
        movdqa    %xmm15, %xmm5
        movdqu    %xmm1, (%rsi)
        addq      $16, %rsi
        decq      %r10
        jne       DLOOP_4_2
DEND_4:
        ret
```

CTR MODE

Figure 47. CTR Encryption Parallelizing 4 Blocks (AT&T Assembly Function)

```
.align    16
ONE:
.quad 0x00000000,0x00000001
.align    16
FOUR:
.quad 0x00000004,0x00000004
.align    16
EIGHT:
.quad 0x00000008,0x00000008
.align    16
TWO_N_ONE:
.quad 0x00000002,0x00000001
.align    16
TWO_N_TWO:
.quad 0x00000002,0x00000002
.align    16
LOAD_HIGH_BROADCAST_AND_BSWAP:
.byte 15,14,13,12,11,10,9,8,15,14,13,12,11,10,9,8
.align    16
BSWAP_EPI_64:
.byte 7,6,5,4,3,2,1,0,15,14,13,12,11,10,9,8

//AES_CTR_encrypt (const unsigned char *in,
//                 unsigned char *out,
//                 const unsigned char ivec[8],
//                 const unsigned char nonce[4],
//                 unsigned long length,
//                 const unsigned char *key,
 //                 int nr)

.globl AES_CTR_encrypt

AES_CTR_encrypt:
# parameter 1: %rdi
# parameter 2: %rsi
# parameter 3: %rdx
# parameter 4: %rcx
# parameter 5: %r8
# parameter 6: %r9
# parameter 7: 8 + %rsp

        movq      %r8, %r10
        movl      8(%rsp), %r12d
        shrq      $4, %r8
        shlq      $60, %r10
        je          NO_PARTS_4
        addq      $1, %r8
NO_PARTS_4:
        movq      %r8, %r10
        shlq      $62, %r10
        shrq      $62, %r10

        pinsrq    $1, (%rdx), %xmm0
        pinsrd    $1, (%rcx), %xmm0
        psrldq    $4, %xmm0
        movdqa    %xmm0, %xmm2
        pshufb    (LOAD_HIGH_BROADCAST_AND_BSWAP), %xmm2
        paddq     (TWO_N_ONE), %xmm2
        movdqa    %xmm2, %xmm1
        paddq     (TWO_N_TWO), %xmm2
        pshufb    (BSWAP_EPI_64), %xmm1
        pshufb    (BSWAP_EPI_64), %xmm2

        shrq      $2, %r8
        je        REMAINDER_4
        subq      $64, %rsi
        subq      $64, %rdi
LOOP_4:
        addq      $64, %rsi
        addq      $64, %rdi

        movdqa    %xmm0, %xmm11
        movdqa    %xmm0, %xmm12
        movdqa    %xmm0, %xmm13
        movdqa    %xmm0, %xmm14

        shufpd    $2, %xmm1, %xmm11
        shufpd    $0, %xmm1, %xmm12
        shufpd    $2, %xmm2, %xmm13
        shufpd    $0, %xmm2, %xmm14

        pshufb    (BSWAP_EPI_64), %xmm1
        pshufb    (BSWAP_EPI_64), %xmm2

        movdqa      (%r9), %xmm8
        movdqa      16(%r9), %xmm9
        movdqa      32(%r9), %xmm10
        movdqa      48(%r9), %xmm7

        paddq     (FOUR), %xmm1
        paddq     (FOUR), %xmm2

        pxor      %xmm8, %xmm11
        pxor      %xmm8, %xmm12
        pxor      %xmm8, %xmm13
        pxor      %xmm8, %xmm14

        pshufb    (BSWAP_EPI_64), %xmm1
        pshufb    (BSWAP_EPI_64), %xmm2

        aesenc    %xmm9, %xmm11
        aesenc    %xmm9, %xmm12
        aesenc    %xmm9, %xmm13
        aesenc    %xmm9, %xmm14
        aesenc    %xmm10, %xmm11
        aesenc    %xmm10, %xmm12
        aesenc    %xmm10, %xmm13
        aesenc    %xmm10, %xmm14
        aesenc    %xmm7, %xmm11
        aesenc    %xmm7, %xmm12
        aesenc    %xmm7, %xmm13
```

```
         aesenc    %xmm7, %xmm14

         movdqa      64(%r9), %xmm8
         movdqa      80(%r9), %xmm9
         movdqa      96(%r9), %xmm10
         movdqa      112(%r9), %xmm7

         aesenc    %xmm8, %xmm11
         aesenc    %xmm8, %xmm12
         aesenc    %xmm8, %xmm13
         aesenc    %xmm8, %xmm14
         aesenc    %xmm9, %xmm11
         aesenc    %xmm9, %xmm12
         aesenc    %xmm9, %xmm13
         aesenc    %xmm9, %xmm14
         aesenc    %xmm10, %xmm11
         aesenc    %xmm10, %xmm12
         aesenc    %xmm10, %xmm13
         aesenc    %xmm10, %xmm14
         aesenc    %xmm7, %xmm11
         aesenc    %xmm7, %xmm12
         aesenc    %xmm7, %xmm13
         aesenc    %xmm7, %xmm14

         movdqa      128(%r9), %xmm8
         movdqa      144(%r9), %xmm9
         movdqa      160(%r9), %xmm10
         cmp          $12, %r12d

         aesenc    %xmm8, %xmm11
         aesenc    %xmm8, %xmm12
         aesenc    %xmm8, %xmm13
         aesenc    %xmm8, %xmm14
         aesenc    %xmm9, %xmm11
         aesenc    %xmm9, %xmm12
         aesenc    %xmm9, %xmm13
         aesenc    %xmm9, %xmm14
         jb          LAST_4
         movdqa      160(%r9), %xmm8
         movdqa      176(%r9), %xmm9
         movdqa      192(%r9), %xmm10
         cmp          $14, %r12d

         aesenc    %xmm8, %xmm11
         aesenc    %xmm8, %xmm12
         aesenc    %xmm8, %xmm13
         aesenc    %xmm8, %xmm14
         aesenc    %xmm9, %xmm11
         aesenc    %xmm9, %xmm12
         aesenc    %xmm9, %xmm13
         aesenc    %xmm9, %xmm14
         jb          LAST_4

         movdqa      192(%r9), %xmm8
         movdqa      208(%r9), %xmm9
         movdqa      224(%r9), %xmm10

         aesenc    %xmm8, %xmm11
         aesenc    %xmm8, %xmm12
```

```
        aesenc    %xmm8, %xmm13
        aesenc    %xmm8, %xmm14
        aesenc    %xmm9, %xmm11
        aesenc    %xmm9, %xmm12
        aesenc    %xmm9, %xmm13
        aesenc    %xmm9, %xmm14
LAST_4:

        aesenclast %xmm10, %xmm11
        aesenclast %xmm10, %xmm12
        aesenclast %xmm10, %xmm13
        aesenclast %xmm10, %xmm14

        pxor    (%rdi), %xmm11
        pxor    16(%rdi), %xmm12
        pxor    32(%rdi), %xmm13
        pxor    48(%rdi), %xmm14

        movdqu    %xmm11, (%rsi)
        movdqu    %xmm12, 16(%rsi)
        movdqu    %xmm13, 32(%rsi)
        movdqu    %xmm14, 48(%rsi)
        dec %r8
        jne LOOP_4

        addq  $64,%rsi
        addq  $64,%rdi

REMAINDER_4:
        cmp $0, %r10
        je END_4
        shufpd  $2, %xmm1, %xmm0
IN_LOOP_4:
        movdqa %xmm0, %xmm11
        pshufb (BSWAP_EPI_64), %xmm0
        pxor   (%r9), %xmm11
        paddq  (ONE), %xmm0
        aesenc 16(%r9), %xmm11
        aesenc 32(%r9), %xmm11
        pshufb (BSWAP_EPI_64), %xmm0
        aesenc 48(%r9), %xmm11
        aesenc 64(%r9), %xmm11
        aesenc 80(%r9), %xmm11
        aesenc 96(%r9), %xmm11
        aesenc 112(%r9), %xmm11
        aesenc 128(%r9), %xmm11
        aesenc 144(%r9), %xmm11
        movdqa 160(%r9), %xmm2
        cmp       $12, %r12d
        jb     IN_LAST_4
        aesenc 160(%r9), %xmm11
        aesenc 176(%r9), %xmm11
        movdqa 192(%r9), %xmm2
        cmp       $14, %r12d
        jb     IN_LAST_4
        aesenc 192(%r9), %xmm11
        aesenc 208(%r9), %xmm11
        movdqa 224(%r9), %xmm2
IN_LAST_4:
        aesenclast %xmm2, %xmm11
        pxor       (%rdi) ,%xmm11
        movdqu     %xmm11, (%rsi)
        addq       $16,%rdi
        addq       $16,%rsi
        dec        %r10
        jne        IN_LOOP_4
END_4:
        ret
```

TEST FUNCTIONS

Figure 48. The ECB Main Function

```
//#define AES128
//#define AES192
//#define AES256

#ifndef LENGTH
#define LENGTH 64
#endif

#include <stdint.h>
#include <stdio.h>
#include <wmmintrin.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;

/*test vectors were taken from http://csrc.nist.gov/publications/nistpubs/800-
38a/sp800-38a.pdf*/
ALIGN16 uint8_t AES128_TEST_KEY[] = {0x2b,0x7e,0x15,0x16,0x28,0xae,0xd2,0xa6,
                                     0xab,0xf7,0x15,0x88,0x09,0xcf,0x4f,0x3c};
ALIGN16 uint8_t AES192_TEST_KEY[] = {0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
                                     0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
                                     0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
ALIGN16 uint8_t AES256_TEST_KEY[] = {0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
                                     0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
                                     0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
                                     0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
ALIGN16 uint8_t AES_TEST_VECTOR[] = {0x6b,0xc1,0xbe,0xe2,0x2e,0x40,0x9f,0x96,
                                     0xe9,0x3d,0x7e,0x11,0x73,0x93,0x17,0x2a,
                                     0xae,0x2d,0x8a,0x57,0x1e,0x03,0xac,0x9c,
                                     0x9e,0xb7,0x6f,0xac,0x45,0xaf,0x8e,0x51,
                                     0x30,0xc8,0x1c,0x46,0xa3,0x5c,0xe4,0x11,
                                     0xe5,0xfb,0xc1,0x19,0x1a,0x0a,0x52,0xef,
                                     0xf6,0x9f,0x24,0x45,0xdf,0x4f,0x9b,0x17,
                                     0xad,0x2b,0x41,0x7b,0xe6,0x6c,0x37,0x10};

ALIGN16 uint8_t ECB128_EXPECTED[] = {0x3a,0xd7,0x7b,0xb4,0x0d,0x7a,0x36,0x60,
                                     0xa8,0x9e,0xca,0xf3,0x24,0x66,0xef,0x97,
                                     0xf5,0xd3,0xd5,0x85,0x03,0xb9,0x69,0x9d,
                                     0xe7,0x85,0x89,0x5a,0x96,0xfd,0xba,0xaf,
                                     0x43,0xb1,0xcd,0x7f,0x59,0x8e,0xce,0x23,
                                     0x88,0x1b,0x00,0xe3,0xed,0x03,0x06,0x88,
                                     0x7b,0x0c,0x78,0x5e,0x27,0xe8,0xad,0x3f,
                                     0x82,0x23,0x20,0x71,0x04,0x72,0x5d,0xd4};

ALIGN16 uint8_t ECB192_EXPECTED[] = {0xbd,0x33,0x4f,0x1d,0x6e,0x45,0xf2,0x5f,
                                     0xf7,0x12,0xa2,0x14,0x57,0x1f,0xa5,0xcc,
                                     0x97,0x41,0x04,0x84,0x6d,0x0a,0xd3,0xad,
                                     0x77,0x34,0xec,0xb3,0xec,0xee,0x4e,0xef,
                                     0xef,0x7a,0xfd,0x22,0x70,0xe2,0xe6,0x0a,
                                     0xdc,0xe0,0xba,0x2f,0xac,0xe6,0x44,0x4e,
                                     0x9a,0x4b,0x41,0xba,0x73,0x8d,0x6c,0x72,
                                     0xfb,0x16,0x69,0x16,0x03,0xc1,0x8e,0x0e};

ALIGN16 uint8_t ECB256_EXPECTED[] = {0xf3,0xee,0xd1,0xbd,0xb5,0xd2,0xa0,0x3c,
                                     0x06,0x4b,0x5a,0x7e,0x3d,0xb1,0x81,0xf8,
                                     0x59,0x1c,0xcb,0x10,0xd4,0x10,0xed,0x26,
                                     0xdc,0x5b,0xa7,0x4a,0x31,0x36,0x28,0x70,
                                     0xb6,0xed,0x21,0xb9,0x9c,0xa6,0xf4,0xf9,
                                     0xf1,0x53,0xe7,0xb1,0xbe,0xaf,0xed,0x1d,
                                     0x23,0x30,0x4b,0x7a,0x39,0xf9,0xf3,0xff,
                                     0x06,0x7d,0x8d,0x8f,0x9e,0x24,0xec,0xc7};
/*****************************************************************************/
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(){
    AES_KEY key;
    AES_KEY decrypt_key;
    uint8_t *PLAINTEXT;
    uint8_t *CIPHERTEXT;
    uint8_t *DECRYPTEDTEXT;
    uint8_t *EXPECTED_CIPHERTEXT;
    uint8_t *CIPHER_KEY;
    int i,j;
    int key_length;

    if (!Check_CPU_support_AES()){
        printf("Cpu does not support AES instruction set. Bailing out.\n");
        return 1;
        }
    printf("CPU support AES instruction set.\n\n");

#ifdef AES128
#define STR "Performing AES128 ECB.\n"
```

```
    CIPHER_KEY = AES128_TEST_KEY;
    EXPECTED_CIPHERTEXT = ECB128_EXPECTED;
    key_length = 128;
#elif defined AES192
#define STR "Performing AES192 ECB.\n"
    CIPHER_KEY = AES192_TEST_KEY;
    EXPECTED_CIPHERTEXT = ECB192_EXPECTED;
    key_length = 192;
#elif defined AES256
#define STR "Performing AES256 ECB.\n"
    CIPHER_KEY = AES256_TEST_KEY;
    EXPECTED_CIPHERTEXT = ECB256_EXPECTED;
    key_length = 256;
#endif

    PLAINTEXT = (uint8_t*)malloc(LENGTH);
    CIPHERTEXT = (uint8_t*)malloc(LENGTH);
    DECRYPTEDTEXT = (uint8_t*)malloc(LENGTH);

    for(i=0 ;i<LENGTH/16/4; i++){
        for(j=0; j<4; j++){
            _mm_storeu_si128(&((__m128i*)PLAINTEXT)[i*4+j],
                             ((__m128i*)AES_TEST_VECTOR)[j]);
            }
        }
    for(j=i*4 ; j<LENGTH/16; j++){
        _mm_storeu_si128(&((__m128i*)PLAINTEXT)[j],
                         ((__m128i*)AES_TEST_VECTOR)[j%4]);
        }
    if (LENGTH%16){
        _mm_storeu_si128(&((__m128i*)PLAINTEXT)[j],
                          ((__m128i*)AES_TEST_VECTOR)[j%4]);
        }

    AES_set_encrypt_key(CIPHER_KEY, key_length, &key);
    AES_set_decrypt_key(CIPHER_KEY, key_length, &decrypt_key);

    AES_ECB_encrypt(PLAINTEXT,
                    CIPHERTEXT,
                    LENGTH,
                    key.KEY,
                    key.nr);

    AES_ECB_decrypt(CIPHERTEXT,
                    DECRYPTEDTEXT,
                    LENGTH,
                    decrypt_key.KEY,
                    decrypt_key.nr);

    printf("%s\n",STR);
    printf("The Cipher Key:\n");
    print_m128i_with_string("",((__m128i*)CIPHER_KEY)[0]);
    if (key_length > 128)
      print_m128i_with_string_short("",((__m128i*)CIPHER_KEY)[1],(key_length/8) -16);

    printf("The Key Schedule:\n");
    for (i=0; i< key.nr; i++)
        print_m128i_with_string("",((__m128i*)key.KEY)[i]);
    printf("The PLAINTEXT:\n");
    for (i=0; i< LENGTH/16; i++)
        print_m128i_with_string("",((__m128i*)PLAINTEXT)[i]);
    if (LENGTH%16)
        print_m128i_with_string_short("",((__m128i*)PLAINTEXT)[i],LENGTH%16);

    printf("\n\nThe CIPHERTEXT:\n");
    for (i=0; i< LENGTH/16; i++)
        print_m128i_with_string("",((__m128i*)CIPHERTEXT)[i]);
    if (LENGTH%16)
        print_m128i_with_string_short("",((__m128i*)CIPHERTEXT)[i],LENGTH%16);

    for(i=0; i<LENGTH; i++){
        if (CIPHERTEXT[i] != EXPECTED_CIPHERTEXT[i%(16*4)]){
          printf("The CIPHERTEXT is not equal to the EXPECTED CIHERTEXT.\n\n");
          return 1;
          }
        }
    printf("The CIPHERTEXT equals to the EXPECTED CIHERTEXT.\n\n");

    for(i=0; i<LENGTH; i++){
        if (DECRYPTEDTEXT[i] != PLAINTEXT[i%(16*4)]){
            printf("The DECRYPTED TEXT isn't equal to the original PLAINTEXT!");
            printf("\n\n");
            return 1;
            }
        }
    printf("The DECRYPTED TEXT equals to the original PLAINTEXT.\n\n");

    }
```

Figure 49. CBC Main Function

```
//#define AES128
//#define AES192
//#define AES256

#ifndef LENGTH
#define LENGTH 64
#endif

#include <stdint.h>
#include <stdio.h>
#include <wmmintrin.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;

/*test vectors were taken from http://csrc.nist.gov/publications/nistpubs/800-
38a/sp800-38a.pdf*/
ALIGN16 uint8_t AES128_TEST_KEY[] = {0x2b,0x7e,0x15,0x16,0x28,0xae,0xd2,0xa6,
                                     0xab,0xf7,0x15,0x88,0x09,0xcf,0x4f,0x3c};
ALIGN16 uint8_t AES192_TEST_KEY[] = {0x8e,0x73,0xb0,0xf7,0xda,0x0e,0x64,0x52,
                                     0xc8,0x10,0xf3,0x2b,0x80,0x90,0x79,0xe5,
                                     0x62,0xf8,0xea,0xd2,0x52,0x2c,0x6b,0x7b};
ALIGN16 uint8_t AES256_TEST_KEY[] = {0x60,0x3d,0xeb,0x10,0x15,0xca,0x71,0xbe,
                                     0x2b,0x73,0xae,0xf0,0x85,0x7d,0x77,0x81,
                                     0x1f,0x35,0x2c,0x07,0x3b,0x61,0x08,0xd7,
                                     0x2d,0x98,0x10,0xa3,0x09,0x14,0xdf,0xf4};
ALIGN16 uint8_t AES_TEST_VECTOR[] = {0x6b,0xc1,0xbe,0xe2,0x2e,0x40,0x9f,0x96,
                                     0xe9,0x3d,0x7e,0x11,0x73,0x93,0x17,0x2a,
                                     0xae,0x2d,0x8a,0x57,0x1e,0x03,0xac,0x9c,
                                     0x9e,0xb7,0x6f,0xac,0x45,0xaf,0x8e,0x51,
                                     0x30,0xc8,0x1c,0x46,0xa3,0x5c,0xe4,0x11,
                                     0xe5,0xfb,0xc1,0x19,0x1a,0x0a,0x52,0xef,
                                     0xf6,0x9f,0x24,0x45,0xdf,0x4f,0x9b,0x17,
                                     0xad,0x2b,0x41,0x7b,0xe6,0x6c,0x37,0x10};

ALIGN16 uint8_t CBC_IV[] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                            0x08,0x09,0x0a,0x0b,0x0c,0x0d,0x0e,0x0f};
ALIGN16 uint8_t CBC128_EXPECTED[] = {0x76,0x49,0xab,0xac,0x81,0x19,0xb2,0x46,
                                     0xce,0xe9,0x8e,0x9b,0x12,0xe9,0x19,0x7d,
                                     0x50,0x86,0xcb,0x9b,0x50,0x72,0x19,0xee,
                                     0x95,0xdb,0x11,0x3a,0x91,0x76,0x78,0xb2,
                                     0x73,0xbe,0xd6,0xb8,0xe3,0xc1,0x74,0x3b,
                                     0x71,0x16,0xe6,0x9e,0x22,0x22,0x95,0x16,
                                     0x3f,0xf1,0xca,0xa1,0x68,0x1f,0xac,0x09,
                                     0x12,0x0e,0xca,0x30,0x75,0x86,0xe1,0xa7};
ALIGN16 uint8_t CBC192_EXPECTED[] = {0x4f,0x02,0x1d,0xb2,0x43,0xbc,0x63,0x3d,
                                     0x71,0x78,0x18,0x3a,0x9f,0xa0,0x71,0xe8,
                                     0xb4,0xd9,0xad,0xa9,0xad,0x7d,0xed,0xf4,
                                     0xe5,0xe7,0x38,0x76,0x3f,0x69,0x14,0x5a,
                                     0x57,0x1b,0x24,0x20,0x12,0xfb,0x7a,0xe0,
                                     0x7f,0xa9,0xba,0xac,0x3d,0xf1,0x02,0xe0,
                                     0x08,0xb0,0xe2,0x79,0x88,0x59,0x88,0x81,
                                     0xd9,0x20,0xa9,0xe6,0x4f,0x56,0x15,0xcd};
ALIGN16 uint8_t CBC256_EXPECTED[] = {0xf5,0x8c,0x4c,0x04,0xd6,0xe5,0xf1,0xba,
                                     0x77,0x9e,0xab,0xfb,0x5f,0x7b,0xfb,0xd6,
                                     0x9c,0xfc,0x4e,0x96,0x7e,0xdb,0x80,0x8d,
                                     0x67,0x9f,0x77,0x7b,0xc6,0x70,0x2c,0x7d,
                                     0x39,0xf2,0x33,0x69,0xa9,0xd9,0xba,0xcf,
                                     0xa5,0x30,0xe2,0x63,0x04,0x23,0x14,0x61,
                                     0xb2,0xeb,0x05,0xe2,0xc3,0x9b,0xe9,0xfc,
                                     0xda,0x6c,0x19,0x07,0x8c,0x6a,0x9d,0x1b};
/*****************************************************************************/
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(){
    AES_KEY key;
    AES_KEY decrypt_key;
    uint8_t *PLAINTEXT;
    uint8_t *CIPHERTEXT;
    uint8_t *DECRYPTEDTEXT;
    uint8_t *EXPECTED_CIPHERTEXT;
    uint8_t *CIPHER_KEY;
    int i,j;
    int key_length;

    if (!Check_CPU_support_AES()){
        printf("Cpu does not support AES instruction set. Bailing out.\n");
        return 1;
        }
    printf("CPU support AES instruction set.\n\n");

#ifdef AES128
#define STR "Performing AES128 CBC.\n"
    CIPHER_KEY = AES128_TEST_KEY;
    EXPECTED_CIPHERTEXT = CBC128_EXPECTED;
    key_length = 128;
#elif defined AES192
#define STR "Performing AES192 CBC.\n"
    CIPHER_KEY = AES192_TEST_KEY;
    EXPECTED_CIPHERTEXT = CBC192_EXPECTED;
    key_length = 192;
#elif defined AES256
#define STR "Performing AES256 CBC.\n"
    CIPHER_KEY = AES256_TEST_KEY;
    EXPECTED_CIPHERTEXT = CBC256_EXPECTED;
    key_length = 256;
#endif

    PLAINTEXT = (uint8_t*)malloc(LENGTH);
    CIPHERTEXT = (uint8_t*)malloc(LENGTH);
    DECRYPTEDTEXT = (uint8_t*)malloc(LENGTH);

    for(i=0 ;i<LENGTH/16/4; i++){
        for(j=0; j<4; j++){
            _mm_storeu_si128(&((__m128i*)PLAINTEXT)[i*4+j],
                               ((__m128i*)AES_TEST_VECTOR)[j]);
            }
        }
    for(j=i*4 ; j<LENGTH/16; j++){
        _mm_storeu_si128(&((__m128i*)PLAINTEXT)[j],
                           ((__m128i*)AES_TEST_VECTOR)[j%4]);
        }
    if (LENGTH%16){
        _mm_storeu_si128(&((__m128i*)PLAINTEXT)[j],
                           ((__m128i*)AES_TEST_VECTOR)[j%4]);
        }

    AES_set_encrypt_key(CIPHER_KEY, key_length, &key);
    AES_set_decrypt_key(CIPHER_KEY, key_length, &decrypt_key);

    AES_CBC_encrypt(PLAINTEXT,
                    CIPHERTEXT,
                    CBC_IV,
                    LENGTH,
                    key.KEY,
                    key.nr);
    AES_CBC_decrypt(CIPHERTEXT,
                    DECRYPTEDTEXT,
                    CBC_IV,
                    LENGTH,
                    decrypt_key.KEY,
                    decrypt_key.nr);

    printf("%s\n",STR);
    printf("The Cipher Key:\n");
    print_m128i_with_string("",((__m128i*)CIPHER_KEY)[0]);
    if (key_length > 128)
      print_m128i_with_string_short("",((__m128i*)CIPHER_KEY)[1],(key_length/8) -16);

    printf("The Key Schedule:\n");
    for (i=0; i< key.nr; i++)
        print_m128i_with_string("",((__m128i*)key.KEY)[i]);

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

    printf("\n\nThe CIPHERTEXT:\n");
    for (i=0; i< LENGTH/16; i++)
        print_m128i_with_string("",((__m128i*)CIPHERTEXT)[i]);
    if (LENGTH%16)
        print_m128i_with_string_short("",((__m128i*)CIPHERTEXT)[i],LENGTH%16);

    for(i=0; i<((64<LENGTH)? 64 : LENGTH); i++){
        if (CIPHERTEXT[i] != EXPECTED_CIPHERTEXT[i%64]){
            printf("The ciphertext is not equal to the expected ciphertext.\n\n");
            return 1;
            }
        }
    printf("The CIPHERTEXT equals to the EXPECTED CIHERTEXT"
           " for bytes where expected text was entered.\n\n");

    for(i=0; i<LENGTH; i++){
        if (DECRYPTEDTEXT[i] != PLAINTEXT[i%(16*4)]){
            printf("%x",DECRYPTEDTEXT[i]);
            printf("The DECRYPTED TEXT is not equal to the original"
                   "PLAINTEXT.\n\n");
            return 1;
            }
        }
    printf("The DECRYPTED TEXT equals to the original PLAINTEXT.\n\n");

    }
```

Figure 50. CTR Main Function

```
#ifndef LENGTH
#define LENGTH 64
#endif

#include <stdint.h>
```

```
#include <stdio.h>
#include <wmmintrin.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;

/*test vectors were taken from http://w3.antd.nist.gov/iip_pubs/rfc3602.txt*/

ALIGN16 uint8_t AES128_TEST_KEY[] = {0x7E,0x24,0x06,0x78,0x17,0xFA,0xE0,0xD7,
                                     0x43,0xD6,0xCE,0x1F,0x32,0x53,0x91,0x63};

ALIGN16 uint8_t AES192_TEST_KEY[] = {0x7C,0x5C,0xB2,0x40,0x1B,0x3D,0xC3,0x3C,
                                     0x19,0xE7,0x34,0x08,0x19,0xE0,0xF6,0x9C,
                                     0x67,0x8C,0x3D,0xB8,0xE6,0xF6,0xA9,0x1A};
ALIGN16 uint8_t AES256_TEST_KEY[] = {0xF6,0xD6,0x6D,0x6B,0xD5,0x2D,0x59,0xBB,
                                     0x07,0x96,0x36,0x58,0x79,0xEF,0xF8,0x86,
                                     0xC6,0x6D,0xD5,0x1A,0x5B,0x6A,0x99,0x74,
                                     0x4B,0x50,0x59,0x0C,0x87,0xA2,0x38,0x84};

ALIGN16 uint8_t AES_TEST_VECTOR[] = {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,
                                     0x08,0x09,0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,
                                     0x10,0x11,0x12,0x13,0x14,0x15,0x16,0x17,
                                     0x18,0x19,0x1A,0x1B,0x1C,0x1D,0x1E,0x1F};

ALIGN16 uint8_t CTR128_IV[] = {0xC0,0x54,0x3B,0x59,0xDA,0x48,0xD9,0x0B};
ALIGN16 uint8_t CTR192_IV[] = {0x02,0x0C,0x6E,0xAD,0xC2,0xCB,0x50,0x0D};
ALIGN16 uint8_t CTR256_IV[] = {0xC1,0x58,0x5E,0xF1,0x5A,0x43,0xD8,0x75};

ALIGN16 uint8_t CTR128_NONCE[] = {0x00,0x6C,0xB6,0xDB};
ALIGN16 uint8_t CTR192_NONCE[] = {0x00,0x96,0xB0,0x3B};
ALIGN16 uint8_t CTR256_NONCE[] = {0x00,0xFA,0xAC,0x24};

ALIGN16 uint8_t CTR128_EXPECTED[] = {0x51,0x04,0xA1,0x06,0x16,0x8A,0x72,0xD9,
                                     0x79,0x0D,0x41,0xEE,0x8E,0xDA,0xD3,0x88,
                                     0xEB,0x2E,0x1E,0xFC,0x46,0xDA,0x57,0xC8,
                                     0xFC,0xE6,0x30,0xDF,0x91,0x41,0xBE,0x28};

ALIGN16 uint8_t CTR192_EXPECTED[] = {0x45,0x32,0x43,0xFC,0x60,0x9B,0x23,0x32,
                                     0x7E,0xDF,0xAA,0xFA,0x71,0x31,0xCD,0x9F,
                                     0x84,0x90,0x70,0x1C,0x5A,0xD4,0xA7,0x9C,
                                     0xFC,0x1F,0xE0,0xFF,0x42,0xF4,0xFB,0x00};

ALIGN16 uint8_t CTR256_EXPECTED[] = {0xF0,0x5E,0x23,0x1B,0x38,0x94,0x61,0x2C,
                                     0x49,0xEE,0x00,0x0B,0x80,0x4E,0xB2,0xA9,
                                     0xB8,0x30,0x6B,0x50,0x8F,0x83,0x9D,0x6A,
                                     0x55,0x30,0x83,0x1D,0x93,0x44,0xAF,0x1C};
/*****************************************************************************/
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(){
    AES_KEY key;
    uint8_t *PLAINTEXT;
    uint8_t *CIPHERTEXT;
    uint8_t *DECRYPTEDTEXT;
    uint8_t *EXPECTED_CIPHERTEXT;
    uint8_t *CIPHER_KEY;
    uint8_t *NONCE;
    uint8_t *IV;
    int i,j;
    int key_length;

    if (!Check_CPU_support_AES()){
        printf("Cpu does not support AES instruction set. Bailing out.\n");
        return 1;
        }
    printf("CPU support AES instruction set.\n\n");


#ifdef AES128
#define STR "Performing AES128 CTR.\n"
    CIPHER_KEY = AES128_TEST_KEY;
    EXPECTED_CIPHERTEXT = CTR128_EXPECTED;
    IV = CTR128_IV;
    NONCE = CTR128_NONCE;
    key_length = 128;
#elif defined AES192
#define STR "Performing AES192 CTR.\n"
    CIPHER_KEY = AES192_TEST_KEY;
    EXPECTED_CIPHERTEXT = CTR192_EXPECTED;
    IV = CTR192_IV;
    NONCE = CTR192_NONCE;
    key_length = 192;
#elif defined AES256
#define STR "Performing AES256 CTR.\n"
    CIPHER_KEY = AES256_TEST_KEY;
    EXPECTED_CIPHERTEXT = CTR256_EXPECTED;
    IV = CTR256_IV;
    NONCE = CTR256_NONCE;
    key_length = 256;
#endif

    PLAINTEXT = (uint8_t*)malloc(LENGTH);
    CIPHERTEXT = (uint8_t*)malloc(LENGTH);
    DECRYPTEDTEXT = (uint8_t*)malloc(LENGTH);
    for(i=0 ;i<LENGTH/16/2; i++){
        for(j=0; j<2; j++){
            _mm_storeu_si128(&((__m128i*)PLAINTEXT)[i*2+j],
                              ((__m128i*)AES_TEST_VECTOR)[j]);
            }
        }
    for(j=i*2 ; j<LENGTH/16; j++){
        _mm_storeu_si128(&((__m128i*)PLAINTEXT)[j],
                          ((__m128i*)AES_TEST_VECTOR)[j%4]);
        }
    if (LENGTH%16){
        _mm_storeu_si128(&((__m128i*)PLAINTEXT)[j],
                          ((__m128i*)AES_TEST_VECTOR)[j%4]);
        }

    AES_set_encrypt_key(CIPHER_KEY, key_length, &key);

    AES_CTR_encrypt(PLAINTEXT,
                    CIPHERTEXT,
                    IV,
                    NONCE,
                    LENGTH,
                    key.KEY,
                    key.nr);

    AES_CTR_encrypt(CIPHERTEXT,
                    DECRYPTEDTEXT,
                    IV,
                    NONCE,
                    LENGTH,
                    key.KEY,
                    key.nr);

    printf("%s\n",STR);
    printf("The Cipher Key:\n");
    print_m128i_with_string("",((__m128i*)CIPHER_KEY)[0]);
    if (key_length > 128)
      print_m128i_with_string_short("",((__m128i*)CIPHER_KEY)[1],(key_length/8) -16);

    printf("The Key Schedule:\n");
    for (i=0; i< key.nr; i++)
        print_m128i_with_string("",((__m128i*)key.KEY)[i]);

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

    printf("\n\nThe CIPHERTEXT:\n");
    for (i=0; i< LENGTH/16; i++)
        print_m128i_with_string("",((__m128i*)CIPHERTEXT)[i]);
    if (LENGTH%16)
        print_m128i_with_string_short("",((__m128i*)CIPHERTEXT)[i],LENGTH%16);

    for(i=0; i< ((32<LENGTH)? 32 : LENGTH); i++){
        if (CIPHERTEXT[i] != EXPECTED_CIPHERTEXT[i%(16*2)]){
            printf("The ciphertext is not equal to the expected ciphertext.\n\n");
            return 1;
            }
        }
    printf("The CIPHERTEXT equals to the EXPECTED CIHERTEXT"
           " for bytes where expected text was entered.\n\n");
    for(i=0; i<LENGTH; i++){
        if (DECRYPTEDTEXT[i] != PLAINTEXT[i]){
            printf("The DECRYPTED TEXT is not equal to the original"
                   "PLAINTEXT.\n\n");
            return 1;
            }
        }
    printf("The DECRYPTED TEXT equals to the original PLAINTEXT.\n\n");
    }
```

How to Use the Library

The functions provided above can be copied into files, compiled and linked into working executables, for example, as follows:

Save the functions from Figure 39, 40, 41 into a single file (key_expansion.s)

Save the code from Figure 42 (aes.c)

Save the code from Figures 43, 44 (ecb.s), Figures 45,46 (cbc.s), and Figure 47 (ctr.s).

Compile the .s files by using

`gcc -maes -msse4 *.s` (use gcc version 4.4.2 and above).

Save the test functions from Figures 48, 49, 50 (ecb_main.c, cbc_main.c and ctr_main.c).

Link the required files with gcc, and generate the desired executable.

CODE OUTPUTS

For ECB use:

```
icc ecb_main.c ecb.o key_expansion.o aes.c -D[AES128/192/256] -o ecb_exe
```

(or `gcc -maes -msse4`)

To define parameter length, use `-DLENGTH=xxxx` during compilation

Figure 51. ECB Output Example

```
CPU support AES instruction set.

Performing AES192 ECB.

The Cipher Key:
                                        [0x8e73b0f7da0e6452c810f32b809079e5]
                                        [0x62f8ead2522c6b7b]
The Key Schedule:
                                        [0x8e73b0f7da0e6452c810f32b809079e5]
                                        [0x62f8ead2522c6b7bfe0c91f72402f5a5]
                                        [0xec12068e6c827f6b0e7a95b95c56fec2]
                                        [0x4db7b4bd69b5411885a74796e92538fd]
                                        [0xe75fad44bb095386485af05721efb14f]
                                        [0xa448f6d94d6dce24aa326360113b30e6]
                                        [0xa25e7ed583b1cf9a27f939436a94f767]
                                        [0xc0a69407d19da4e1ec1786eb6fa64971]
                                        [0x485f703222cb8755e26d135233f0b7b3]
                                        [0x40beeb282f18a2596747d26b458c553e]
                                        [0xa7e1466c9411f1df821f750aad07d753]
                                        [0xca4005388fcc5006282d166abc3ce7b5]
The PLAINTEXT:
                                        [0x6bc1bee22e409f96e93d7e117393172a]
                                        [0xae2d8a571e03ac9c9eb76fac45af8e51]
                                        [0x30c81c46a35ce411e5fbc1191a0a52ef]
                                        [0xf69f2445df4f9b17ad2b417be66c3710]


The CIPHERTEXT:
                                        [0xbd334f1d6e45f25ff712a214571fa5cc]
                                        [0x974104846d0ad3ad7734ecb3ecee4eef]
                                        [0xef7afd2270e2e60adce0ba2face6444e]
                                        [0x9a4b41ba738d6c72fb16691603c18e0e]
The CIPHERTEXT equals to the EXPECTED CIHERTEXT.

The DECRYPTED TEXT equals to the original PLAINTEXT.
```

For CBC use:

```
icc cbc_main.c cbc.o key_expansion.o aes.c -D[AES128/192/256] -o cbc_exe
```

Figure 52. CBC Output Example

```
CPU support AES instruction set.

Performing AES128 CBC.

The Cipher Key:
                                        [0x2b7e151628aed2a6abf7158809cf4f3c]
The Key Schedule:
                                        [0x2b7e151628aed2a6abf7158809cf4f3c]
                                        [0xa0fafe1788542cb123a339392a6c7605]
                                        [0xf2c295f27a96b9435935807a7359f67f]
                                        [0x3d80477d4716fe3e1e237e446d7a883b]
                                        [0xef44a541a8525b7fb671253bdb0bad00]
                                        [0xd4d1c6f87c839d87caf2b8bc11f915bc]
                                        [0x6d88a37a110b3efddbf98641ca0093fd]
                                        [0x4e54f70e5f5fc9f384a64fb24ea6dc4f]
                                        [0xead27321b58dbad2312bf5607f8d292f]
                                        [0xac7766f319fadc2128d12941575c006e]
The PLAINTEXT:
                                        [0x6bc1bee22e409f96e93d7e117393172a]
                                        [0xae2d8a571e03ac9c9eb76fac45af8e51]
                                        [0x30c81c46a35ce411e5fbc1191a0a52ef]
                                        [0xf69f2445df4f9b17ad2b417be66c3710]


The CIPHERTEXT:
                                        [0x7649abac8119b246cee98e9b12e9197d]
                                        [0x5086cb9b507219ee95db113a917678b2]
                                        [0x73bed6b8e3c1743b7116e69e22229516]
                                        [0x3ff1caa1681fac09120eca307586e1a7]
The CIPHERTEXT equals to the EXPECTED CIHERTEXT for bytes where expected text was entered.

The DECRYPTED TEXT equals to the original PLAINTEXT.
```

For CTR:

```
icc ctr_main.c ctr.o key_expansion.o aes.c -D[AES128/192/256] -o ctr_exe
```

Figure 53. CTR Output Example

```
CPU support AES instruction set.

Performing AES256 CTR.

The Cipher Key:
                                        [0xf6d66d6bd52d59bb0796365879eff886]
                                        [0xc66dd51a5b6a99744b50590c87a23884]
The Key Schedule:
                                        [0xf6d66d6bd52d59bb0796365879eff886]
                                        [0xc66dd51a5b6a99744b50590c87a23884]
                                        [0xcdd1327c18fc6bc71f6a5d9f6685a519]
                                        [0xf5fad3ceae904abae5c013b662622b32]
                                        [0x652011d67ddc7a1162b6278e04338297]
                                        [0x0739c046a9a98afc4c69994a2e0bb278]
                                        [0x4a17ade737cbd7f6557df078514e72ef]
                                        [0xd61680997fbf0a6533d6932f1ddd2157]
                                        [0x83eaf643b42121b5e15cd1cdb012a322]
                                        [0x31df8a0a4e60806f7db61340606b3217]
                                        [0xecc9069358e82726b9b4f6eb09a655c9]
                                        [0x30fb76d77e9bf6b8032de5f86346d7ef]
                                        [0x96c7d968ce2ffe4e779b08a57e3d5d6c]
                                        [0xc3dc3a87bd47cc3fbe6a29c7dd2cfe28]
The PLAINTEXT:
                                        [0x000102030405060708090a0b0c0d0e0f]
                                        [0x101112131415161718191a1b1c1d1e1f]
                                        [0x000102030405060708090a0b0c0d0e0f]
                                        [0x101112131415161718191a1b1c1d1e1f]


The CIPHERTEXT:
                                        [0xf05e231b3894612c49ee000b804eb2a9]
                                        [0xb8306b508f839d6a5530831d9344af1c]
                                        [0xd59e1f4edc334422d0192f2679722a1c]
                                        [0x2a771e7d6ae0d56113dcc8762b8bc18d]
The CIPHERTEXT equals to the EXPECTED CIHERTEXT for bytes where expected text was entered.

The DECRYPTED TEXT equals to the original PLAINTEXT.
```

Performance Results

This chapter provides the “Westmere” performance results (single thread only) obtained from running the code given in the “AES Library” chapter. The performance measurements of the given functions were carried out by using the Time Stamp Counter (RDTSC instruction) and averaging over a large number of repetitions, after some “warmup” iterations. The following “measurement macro” was used.

Figure 54. The Measurement Macro

```
#ifndef REPEAT
    #define REPEAT 1000000
#endif
#ifndef WARMUP
    #define WARMUP REPEAT/4
#endif

    UINT64 start_clk,end_clk;
    double total_clk;

__inline UINT64 get_Clks(void) {
    UINT64 tmp;
    __asm__ volatile(
            "rdtsc\n\t\
            mov %%eax,(%0)\n\t\
            mov %%edx,4(%0)"::"rm"(&tmp):"eax","edx");
    return tmp;
}

#define MEASURE(x)  for (i=0; i< WARMUP; i++)           \
                                {x;}                    \
                    start_clk=get_Clks();               \
                    for (i = 0; i < REPEAT; i++)        \
                    {                                   \
                                {x;}                    \
                    }                                   \
                    end_clk=get_Clks();                 \
                    total_clk=(double)(end_clk-start_clk)/REPEAT;
```

The experiments were carried out on a Westmere-based processor 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).

Performance Results

AES Key Expansion	
Key Size	Cycles
AES-128	108
AES-192	104
AES-256	136

Table 1. The Performance of the AES Key Expansion (Westmere)

	AES 128	AES 192	AES 256
	Performance in CPU Cycles Per Byte for a 1KB buffer		
ECB Encryption	1.28	1.53	1.76
ECB Decryption	1.26	1.51	1.76
CBC Encryption	4.15	4.91	5.65
CBC Decryption	1.30	1.53	1.78
CTR Encryption /Decryption	1.38	1.61	1.88

Table 2. The Performance of AES Encryption and Decryption of a 1K Bytes Buffer, in Various Modes of Operation (Westmere)

Code:	CBC encryption of 4 buffers in parallel	ECB encrypt of 1 block	CTR encrypt of 1 block
	Performance in CPU Cycles Per Byte		
AES-128	1.33	2.01	2.09

Table 3. Additional Performance Numbers (Westmere)

Code:	Key Expansion	CBC encrypt	ECB encrypt	CTR encrypt
		Performance in CPU Cycles Per Byte		
AES-128	164.00	17.66	15.38	19.60

Table 4. AES Performance Numbers in OpenSSL (Westmere, without AES-NI)

Figure 55. The Performance of AES-128 Encryption in ECB Mode, as a Function of the Buffer Size (Westmere)

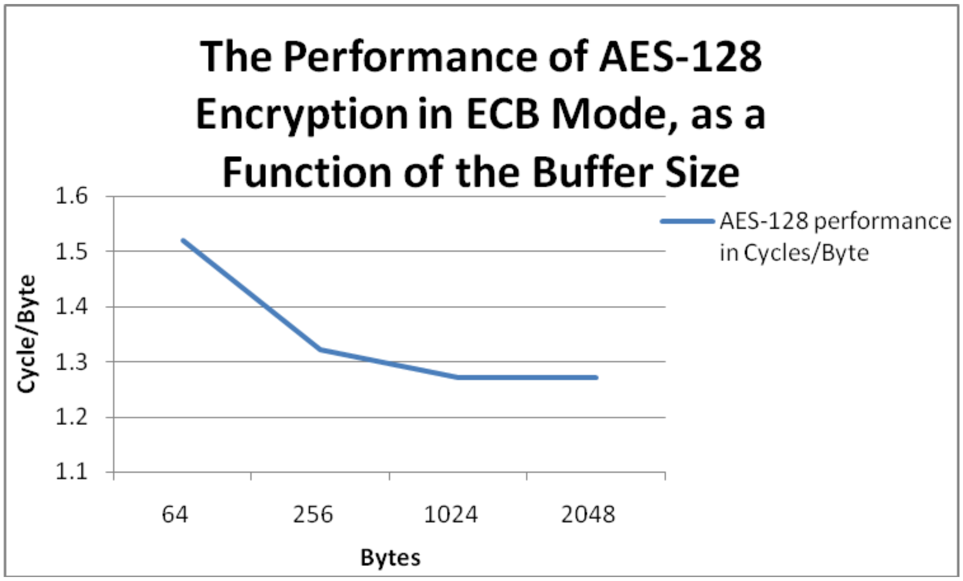


Figure 56. The Performance of AES-128 Decryption in CBC Mode, as a Function of the Buffer Size (Westmere)

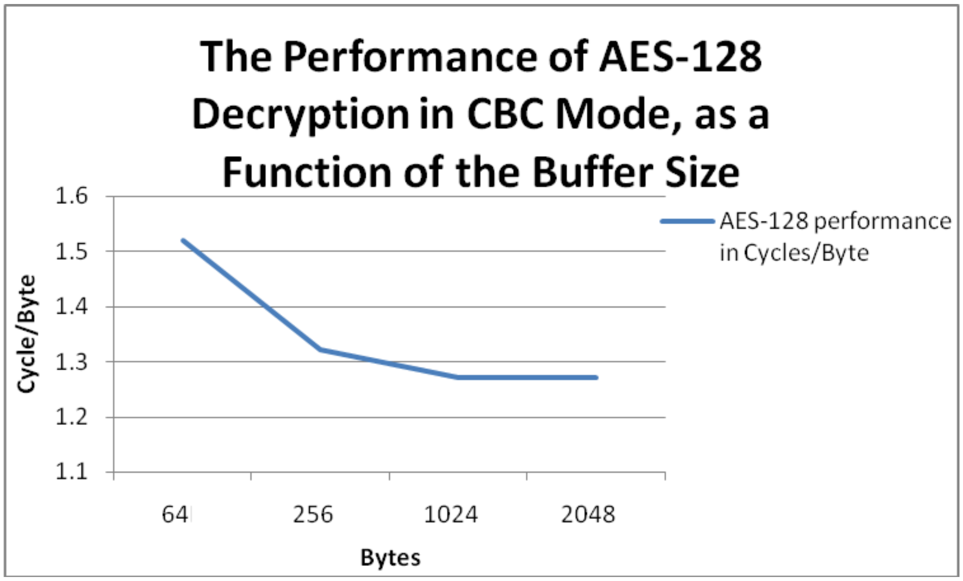
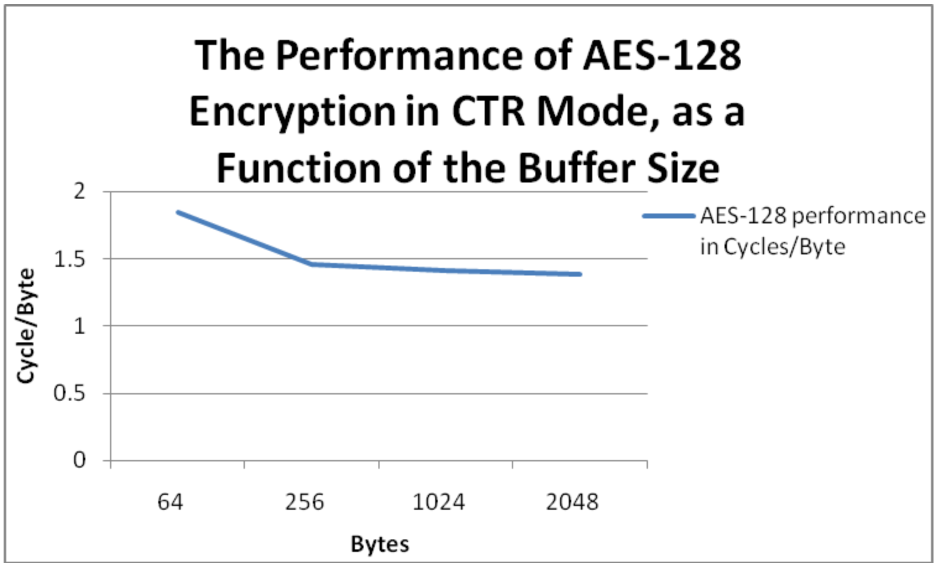


Figure 57. The Performance of AES-128 Encryption in CTR Mode, as a Function of the Buffer Size (Westmere)



Conclusion

This paper described Intel®'s new AES instructions set, which, starting January 2010, are now part of the Intel IA-32 architecture.

Six new AES instructions are offered, to provide important performance and security benefits. AES is the leading standard for symmetric encryption, used in a variety of applications. For example, OS level disk encryption is a notable usage model, which can be greatly accelerated with the proposed instructions. Consequently, a high performance and secure solution for AES computations in commodity processors is a useful and important technology.

The new instructions support all standard AES key lengths and modes of operation, as well as some non-standard variants, and usage models. They can increase performance by more than an order of magnitude for parallel modes of operation (e.g., CTR and CBC-decrypt), and provide roughly 2-3 fold gains for non-parallelizable modes (such as CBC-encrypt). The AES code, using the new instructions, which is given in this paper, can run at ~1.3 C/B in parallel modes of operation.

Beyond improving performance, the new instructions help address software side channel vulnerabilities, because they run with data-independent latency and do not use lookup tables. This eliminates the major timing and cache attacks (that can be launched by Ring 3 spy codes) that threaten table-based software implementations of AES.

Acknowledgements

Many people have contributed to the concepts, the studies, and to the implementation of the AES architecture and micro-architecture. The list of contributors includes:

> Roee Bar, Frank Berry, Mayank Bomb, Brent Boswell, Ernie Brickell, Yuval Bustan, Mark Buxton, Srinivas Chennupaty, Tiran Cohen, Martin Dixon, Jack Doweck, Vivek Echambadi, Wajdi Feghali, Shay Fux, Vinodh Gopal, Eugene Gorkov, Amit Gradstein, Mostafa Hagog, Israel Hayun, Michael Kounavis, Ram Krishnamurthy, Sanu Mathew, Henry Ou, Efi Rosenfeld, Zeev Sperber, Kirk Yap.

I also thank Roee Bar, Joseph Bonneau, Mark Buxton, Mark Charney, Kevin Gotze, Michael Kounavis, Paul Kocher, Vlad Krasnov, Shihjong Kuo, Hongjiu Lu, Mark Marson, Trevor Perrin, Aaron Tersteeg, for helpful suggestions and corrections to previous versions of this paper.

About the Author

Shay Gueron is an Intel Principal Engineer. He works at the CPU Architecture Department in the Mobility Group, at the Israel Development Center. His interests include applied security, cryptography, and algorithms. Shay 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. His email is shay.gueron@intel.com.

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

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

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

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

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

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