

# Intel's New AES Instructions for Enhanced Performance and Security

Shay Gueron<sup>1,2</sup>

<sup>1</sup> Intel Corporation, Mobility Group, Israel Development Center, Haifa, Israel  
<sup>2</sup> University of Haifa, Faculty of Science, Department of Mathematics, Haifa, Israel

**Abstract.** The Advanced Encryption Standard (AES) is the Federal Information Processing Standard for symmetric encryption. It 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. To this end, Intel is introducing a new set of instructions into the next generation of its processors, starting from 2009. 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, and are therefore useful for a wide range of cryptographic applications. The 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. They can also be used for all other block sizes of the general RIJNDAEL cipher. The instructions 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. Beyond improving performance, the AES instructions provide important security benefits. Since the instructions run in data independent time and do not use table lookups, they help eliminating the major timing and cache-based attacks that threaten table-lookup based software implementations of AES. In addition, these instructions make AES simple to implement, with reduced code size. This helps reducing the risk of inadvertent introduction of security flaws, such as difficult-to-detect side channel leaks. This paper provides an overview of the new AES instructions and how they can be used for achieving high performance and secure AES processing. Some special usage models of this architecture are also described.

**Keywords:** Advanced Encryption Standard, computer architecture, new instructions set.

## 1 Introduction

The Advanced Encryption Standard (AES), defined in 2001 by NIST [11]. (FIPS197 hereafter), is considered the state of the art in symmetric encryption, and a crucial ingredient for security and privacy applications. Rising requirements for high encryption/decryption bandwidths that have minimal impact on the user experience, increase the value of a high throughput AES solution for commodity processors. One example is disk encryption applications, such as Microsoft’s Vista BitLocker [10], where due to increased volume size and disks speed, software encryption overhead may become a bottleneck for both the client and the server platforms.

The security of AES execution is an additional consideration added to the PC environment due to increased awareness to recent side channel attacks on AES software that uses lookup tables (e.g., [13]). Mitigation techniques significantly degrade the resulting performance, therefore making a hardware based AES solution even more advantageous.

Intel offers a comprehensive hardware solution for AES, introducing six new instructions to its processors, starting from the processor called “Westmere”.

This paper describes the instructions, how they can be used efficiently and flexibly, and explains some of the benefits of this particular AES architecture.

## 2 Intel’s AES Architecture

### 2.1 Preliminaries and Notations

Hereafter, we use the terminology of FIPS197, which details of all transformations, flows for encryption/decryption and key expansion that define AES.

We point out some subtlety related to the notation conventions. 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 transformations. On the other hand, Intel’s Architecture convention is the opposite: hexadecimal strings are written with the low-memory byte on the right and the high-memory byte on the left (analogous to writing integers in a “Little Endian” convention). Of course, store/load processor operations are consistent with the way that the AES instructions operate (i.e., using these instructions does not require any byte reversal). For reference, we provide here an example for all of the eight AES transformations, expressed in the “Little Endian” convention which is used on Intel’s processors.

SubBytes(73744765635354655d5b56727b746f5d) =	8f92a04dfbed204d4c39b1402192a84c
MixColumns(627a6f6644b109c82b18330a81c3b3e5) =	7b5b54657374566563746f725d53475d
ShiftRows(7b5b54657374566563746f725d53475d) =	73744765635354655d5b56727b746f5d
InvMixColumns(8dcab9dc035006bc8f57161e00cafd8d) =	5be3eb11928b5eaeec9cc3bc55f5777
InvShiftRows(7b5b54657374566563746f725d53475d) =	5d7456657b536f65735b47726374545d
InvSubBytes(5d7456657b536f65735b47726374545d) =	8dcab9dc035006bc8f57161e00cafd8d
RotWord(3c4fcf09) = 093c4fcf	SubWord(73744765) = 8f92a04d

**Fig. 1.** The AES transformations expressed in “Little Endian” notation, as used in Intel’s architecture

## 2.2 The Six AES Instructions

Intel’s architecture offers six instructions to support AES (see Fig. 2). AESENC, AESENCLAST, support encryption. AESDEC and AESDECLAST are building blocks suitable for decryption using the Equivalent Inverse Cipher (see FIPS197 for definition). Each instruction has a register-memory and a register-register variant. AESIMC and AESKEYGENASSIST support the Key Expansion. AESIMC facilitates the conversion of the encryption round keys to a form suitable for the Equivalent Inverse Cipher. AESKEYGENASSIST uses an immediate byte as part of the input (used as RCON).

## 3 Basic Usage of the AES Instructions

This section illustrates the basic usage of the AES instructions, using AES-128 (ECB mode) as an example. The general paradigm is that for AESENC, AESENCLAST, AESDEC, AESDECLAST, the inputs xmm1 and xmm2 are interpreted as xmm1 = State and xmm2 = Round Key. For AESIMC, the input xmm2 is interpreted as xmm2 = Round Key. Fig. 3 illustrates encryption/decryption flows. For AESKEYGENASSIST, the input should be interpreted as an intermediate step in the Key Expansion procedure, where the immediate byte is the value of RCON. An example for AES-128 Key Expansion is illustrated in Fig. 4 (Key Expansion for AES-192 and AES-256 is provided in the Appendix).

## 4 Some Design Considerations That Led to the Selection of the AES Architecture

Introducing a new instruction to Intel’s processors implies long-term legacy commitment. Additionally, silicon area is a precious “real-estate”. This mandates a great deal of care in the definitions and cost-performance-flexibility tradeoffs.

<b>AESENC xmm1, xmm2/m128</b> Tmp := xmm1 RoundKey := xmm2/m128 Tmp := ShiftRows (Tmp) Tmp := SubBytes (Tmp) Tmp := MixColumns (Tmp) xmm1 := Tmp xor RoundKey	<b>AESENCLAST xmm1, xmm2/m128</b> Tmp := xmm1 RoundKey := xmm2/m128 Tmp := ShiftRows (Tmp) Tmp := SubBytes (Tmp) xmm1 := Tmp xor RoundKey
<b>AESDEC xmm1, xmm2/m128</b> Tmp := xmm1 RoundKey := xmm2/m128 Tmp := InvShiftRows (Tmp) Tmp := InvSubBytes (Tmp) Tmp := InvMixColumns (Tmp) xmm1 := Tmp xor RoundKey	<b>AESDECLAST xmm1, xmm2/m128</b> Tmp := xmm1 RoundKey := xmm2/m128 Tmp := InvShiftRows (Tmp) Tmp := InvSubBytes (Tmp) xmm1 := Tmp xor RoundKey
<b>AESKEYGENASSIST xmm1, xmm2/m128, imm8</b> Tmp := xmm2/m128 RCON[31-8] := 0; RCON[7-0] := imm8; X3[31-0] := Tmp[127-96]; X2[31-0] := Tmp[95-64]; X1[31-0] := Tmp[63-32]; X0[31-0] := Tmp[31-0]; xmm1 := [RotWord (SubWord (X3)) XOR RCON, SubWord (X3), Rotword (SubWord (X1)) XOR RCON, SubWord (X1)]	
<b>AESIMC xmm1, xmm2/m128</b> RoundKey := xmm2/m128; xmm1 := InvMixColumns (RoundKey)	
<b>Examples:</b>	
xmm1 =	7b5b54657374566563746f725d53475d
xmm2 =	48692853686179295b477565726f6e5d
AESENC result:	a8311c2f9fdba3c58b104b58ded7e595
AESENCLAST result:	c7fb881e938c5964177ec42553fdc611
AESDEC result:	138ac342faea2787b58eb95eb730392a
AESDECLAST result:	c5a391ef6b317f95d410637b72a593d0
xmm2 =	7b5b54657374566563746f725d53475d
AESIMC result:	627a6f6644b109c82b18330a81c3b3e5
xmm2 =	3c4f cf098815f7aba6d2ae2816157e2b; imm8 = 1
AESENC result:	01eb848beb848a013424b5e524b5e434

**Fig. 2.** Functional descriptions (architectural behavior) and examples of the AES instructions (note that ShiftRows and SubBytes, InvShiftRows and InvSubBytes commute)

Obviously, the AES architecture must offer an adequate solution for the short term requirements, but as importantly, it should have the ability to accommodate long range requirements that may emerge in the future. Therefore, the AES architecture needs to address the following considerations: a) Flexibility, b) Performance, c) Performance scalability, d) Security. We explain how these properties are achieved by the AES architecture.

#### 4.1 Design for Software Flexibility

Software flexibility implies that the architecture should be able to support all of the current usage models for AES. Indeed, it is easy to realize that this the case with the new AES instructions: They are the building blocks that can support all the AES variants defined by FIPS197, uses of AES in 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. As an example, Fig. 5 shows encryption in CBC mode.

AES-128 encryption	Decryption Round Keys	AES-128 decryption
pxor xmm1, xmm2		pxor xmm1, xmm12;
AESENC xmm1, xmm3	AESIMC xmm3, xmm3	AESDEC xmm1, xmm11
AESENC xmm1, xmm4	AESIMC xmm4, xmm4	AESDEC xmm1, xmm10
AESENC xmm1, xmm5	AESIMC xmm5, xmm5	AESDEC xmm1, xmm9
AESENC xmm1, xmm6	AESIMC xmm6, xmm6	AESDEC xmm1, xmm8
AESENC xmm1, xmm7	AESIMC xmm7, xmm7	AESDEC xmm1, xmm7
AESENC xmm1, xmm8	AESIMC xmm8, xmm8	AESDEC xmm1, xmm6
AESENC xmm1, xmm9	AESIMC xmm9, xmm9	AESDEC xmm1, xmm5
AESENC xmm1, xmm10	AESIMC xmm10, xmm10	AESDEC xmm1, xmm4
AESENC xmm1, xmm11	AESIMC xmm11, xmm11	AESDEC xmm1, xmm3
AESENCLAST xmm1, xmm12		AESENCLAST xmm1, xmm2

**Fig. 3.** Left panel: AES-128 encryption. Register xmm1 holds the data to encrypt, xmm2 is the whitening key, and xmm3–xmm12 hold Round Keys 1–10. The AES flow starts with a whitening step (XOR with xmm2). Rounds 1–9 are implemented using AESENC, and round 10 is implemented using AESENCLAST. Middle panel: AESIMC is used for transforming the round keys for decryption using the Equivalent Inverse Cipher. Right panel: AES-128 decryption. Register xmm1 holds the data to decrypt. Registers xmm12-xmm2 hold the decryption round keys and the whitening key.

<pre> movdqu xmm1, XMMWORD PTR Key movdqu XMMWORD PTR Key_Sched, 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 </pre>	<pre> 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 </pre>
---	--

**Fig. 4.** AES-128 Key Expansion example (the cipher key is stored in the array “Key” and the generated key expansion is stored in the array “Key\_Sched”. (see comments in the Appendix)

Software has the flexibility to pre-expand the keys and re-use them (which is the typical usage model in bulk encryption) or to expand them on-the-fly. In addition, when compared with existing software implementations, one can realize that the AES instructions can help reduce the associated code size. We also point out here that the AES round instructions remain as useful as they are now, even if future analysis would change the standard to perform more rounds during

```

void AES_128_CBC_Encrypt (...) {
    int i, j, k;
    __m128i tmp, feedback;
    __m128i RKEY [11];
    for (k=0; k<11; k++) {
        RKEY [k] = _mm_load_si128 ( (__m128i*)&Key_Schedule [4*k]);
    }
    feedback = _mm_load_si128 ( (__m128i*)&IV [0]);
    for(i=0; i < NBLOCKS; i++) {
        tmp = _mm_load_si128 ( (__m128i*)&PLAINTEXT[i*4]);
        tmp = _mm_xor_si128 (tmp,feedback);
        tmp = _mm_xor_si128(tmp, RKEY[0]);
        for(j=1; j < 10; j++) {
            tmp = _mm_aesenc_si128 (tmp, RKEY [j]);
        }
        tmp = _mm_aesencast_si128 (tmp, RKEY [10]);
        feedback = tmp;
        _mm_store_si128 ((__m128i*)&CIPHERTEXT[4*i], tmp);
    }
}

```

**Fig. 5.** Encryption in CBC mode. A C code snippet, using compiler intrinsics, illustrates a function that encrypts NBLOCKS data blocks.

encryption/decryption. Furthermore, as long as the Key Expansion procedure is not fundamentally changed, AESKEYGENASSIST (taking any Round Constant as an input byte) could be used for generating additional round keys.

## 4.2 Design for Performance

Performance is a main motivation for introducing the AES instructions. To this end, the architecture takes advantage of the 128-bit data-path available in the Intel’s modern processors (compare with the 32-bit instructions proposed in [14], in a different setup, that does not have such a wide data-path).

The AES architecture is optimized for the common usage model for the PC platform where the round keys are generated once, stored in registers or in the cache memory, and then used for multiple data blocks. To this end, the hardware support for the key expansion is decoupled from the more performance-critical encryption/decryption acceleration. The four AES rounds instructions encapsulate the maximal sequence of transformations which is possible without having micro-architectural branches. To illustrate, consider a possible alternative instruction such as AESROUND xmm1, xmm2, imm8, where the immediate byte is a control that selects encryption/decryption and round/last round. Such architecture would require the implementation to have micro-branching which could incur some performance loss. To avoid this, four separate instructions are dedicated to each of the four “flavors” of the AES rounds.

## 4.3 Design for Performance Scalability

Performance scalability is also achieved by encapsulating the “maximal” possible flow in the performance-critical instructions, thus leaving room for micro architectural cost-performance tradeoffs. To illustrate this flexibility, consider the

AESENC instruction that performs the sequence of transformation: ShiftRows; SubBytes; MixColumns; AddRoundKey (=XOR). These could be implemented by one piece of dedicated hardware, or by means of hardware elements that process the data in small granularity combined with some micro-instruction flows. Thus, it is possible to choose the cost-performance balance across processors and processors generations, according to the performance requirements.

To show the benefit of bundling the maximal flow in one instruction, consider the following alternative of having two separate instructions, `SUBBYTES xmm1, xmm2`, and `MIXCOL xmm1, xmm2`. With these, the AES encryption round could be performed by the sequence `PSHUFB` (for ShiftRows), `SUBBYTES`, `MIXCOL`, `PXOR`. However, such an architectural approach limits the highest possible performance of the instruction.

#### 4.4 Design for Security

We briefly explain here how side channel attacks can compromise the security of AES software implementations, and how the new architecture mitigates this problem.

Processor cache is a special type of memory that allows faster access compared to accessing main memory. The processor stores recently read memory areas in cache, with the speculative anticipation that these areas would be re-accessed “soon”. In each memory access, the processor first checks if the data is in the cache (enjoying fast access) and if not, it reads from main memory (or lower level caches), and stores it in the cache for future usage. To place new data in the cache, the processor needs to evict less recent data.

Currently, many common efficient software implementations of AES use lookup tables (e.g., Gladman’s code [4], OpenSSL [12], and Lipmaa [617]). The entries in the table(s), which are read during encryption, depend implicitly on the secret round key and on the processed data. A “spy process”, which runs at the same privilege level, can exploit this fact: it runs in parallel to some AES code, fills the cache lines with its own data, and reads them again after the table was accessed by the AES code. Depending on the reading latency that the spy experiences (for its own data, as measured by using the RDTSC instruction), it can discover if the corresponding cache line was evicted or not, and therefore deduce which part of the table was accessed by the AES code. Repeatedly collected, and combined with the appropriate analysis, this information could eventually leak out the secret key (see e.g., [133]).

This side channel threat can be avoided by writing the AES software in a way that the memory access patterns hide the key dependence (e.g., [133]). However, these mitigation techniques may involve a significant performance penalty. There are also software implementations of AES that do not use table lookup at all (e.g., Matsui [89], Bernstein and Schwabe [2]).

The AES instructions are designed to mitigate all of the known timing and cache side channel leakage of sensitive data. 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,

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 easily writing high performance AES software which is, at the same time, protected against the currently known side channel threats.

## 5 Performance Optimizations for Parallel Modes of Operation

Significant performance optimization for encryption/decryption using the AES instructions can be achieved by re-ordering the code. This helps taking better advantage of parallelism in parallel modes of operation such as ECB, CTR, and CBC-Decrypt (with the CBC-Encrypt serial mode being the 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, 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

<pre> ; load Round key mov rdx, OFFSET keyex_addr movdqu xmm0, XMMWORD PTR [rdx]  pxor xmm1, xmm0 pxor xmm2, xmm0 pxor xmm3, xmm0 pxor xmm4, xmm0 pxor xmm5, xmm0 pxor xmm6, xmm0 pxor xmm7, xmm0 pxor xmm8, xmm0  mov ecx, 9  main_loop: ; load Round key add rdx, 0x10 movdqu xmm0, XMMWORD PTR [rdx]  aesenc xmm1, xmm0 aesenc xmm2, xmm0 aesenc xmm3, xmm0 aesenc xmm4, xmm0 aesenc xmm5, xmm0 aesenc xmm6, xmm0 aesenc xmm7, xmm0 aesenc xmm8, xmm0  loop main_loop </pre>	<pre> add rdx, 0x10 movdqu xmm0, XMMWORD PTR [rdx]  aesenclast xmm1, xmm0 aesenclast xmm2, xmm0 aesenclast xmm3, xmm0 aesenclast xmm4, xmm0  aesenclast xmm5, xmm0 aesenclast xmm6, xmm0 aesenclast xmm7, xmm0 aesenclast xmm8, xmm0  ; storing the encrypted blocks  movdqu XMMWORD PTR [dest], xmm1 movdqu XMMWORD PTR [dest+0x10], xmm2 movdqu XMMWORD PTR [dest+0x20], xmm3 movdqu XMMWORD PTR [dest+0x30], xmm4 movdqu XMMWORD PTR [dest+0x40], xmm5 movdqu XMMWORD PTR [dest+0x50], xmm6 movdqu XMMWORD PTR [dest+0x60], xmm7 movdqu XMMWORD PTR [dest+0x70], xmm8 </pre>
---	---

Fig. 6. Encrypting multiple data blocks in parallel (ECB mode)

for multiple blocks (using another round key). For such optimization, one needs to choose the number of blocks that will be processed in parallel. The 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. In general, it is useful to process 4–8 blocks in parallel, in order to achieve high throughput. We provide here two examples: Figure 6 outlines assembler code for encrypting 8 blocks in parallel, in ECB mode, and Figure 7 gives a C code snippet for decrypting 4 blocks in parallel in CBC mode.

```

void AES_128_CBC_Decrypt_C_4_blocks (...) {
    __m128i RKEY_DECRYPT [11];
    __m128i tmp1, tmp2, tmp3, tmp4, feedback;
    __m128i z1, z2, z3, z4;
    int j, k;
    for (k=0; k<11; k++) {
        RKEY_DECRYPT [10-k] =
            _mm_load_si128 ( (__m128i*)&Key_Schedule_Decrypt [4*k]);
    }
    feedback = _mm_load_si128 ( (__m128i*)&IV [0]);

    z1 = _mm_load_si128 ( (__m128i*)&CIPHERTEXT[0]);
    z2 = _mm_load_si128 ( (__m128i*)&CIPHERTEXT[4]);
    z3 = _mm_load_si128 ( (__m128i*)&CIPHERTEXT[8]);
    z4 = _mm_load_si128 ( (__m128i*)&CIPHERTEXT[12]);

    tmp1 = _mm_xor_si128(z1,RKEY_DECRYPT[0]);
    tmp2 = _mm_xor_si128(z2,RKEY_DECRYPT[0]);
    tmp3 = _mm_xor_si128(z3,RKEY_DECRYPT[0]);
    tmp4 = _mm_xor_si128(z4,RKEY_DECRYPT[0]);

    for(j=1; j <10; j++) {
        tmp1 = _mm_aesdec_si128 (tmp1, RKEY_DECRYPT [j]);
        tmp2 = _mm_aesdec_si128 (tmp2, RKEY_DECRYPT [j]);
        tmp3 = _mm_aesdec_si128 (tmp3, RKEY_DECRYPT [j]);
        tmp4 = _mm_aesdec_si128 (tmp4, RKEY_DECRYPT [j]);
    }
    tmp1 = _mm_aesdeclast_si128 (tmp1, RKEY_DECRYPT [10]);
    tmp2 = _mm_aesdeclast_si128 (tmp2, RKEY_DECRYPT [10]);
    tmp3 = _mm_aesdeclast_si128 (tmp3, RKEY_DECRYPT [10]);
    tmp4 = _mm_aesdeclast_si128 (tmp4, RKEY_DECRYPT [10]);

    tmp4 = _mm_xor_si128(tmp4,z3);
    tmp3 = _mm_xor_si128(tmp3,z2);
    tmp2 = _mm_xor_si128(tmp2,z1);
    tmp1 = _mm_xor_si128(tmp1,feedback);

    _mm_store_si128 ((__m128i*)&DECRYPTED_TEXT[0], tmp1);
    _mm_store_si128 ((__m128i*)&DECRYPTED_TEXT[4], tmp2);
    _mm_store_si128 ((__m128i*)&DECRYPTED_TEXT[8], tmp3);
    _mm_store_si128 ((__m128i*)&DECRYPTED_TEXT[12], tmp4);
}

```

**Fig. 7.** Decrypting 4 blocks in parallel, in CBC mode (C code using compiler intrinsics)

## 5.1 Parallelizing CBC Encryption for Performance

CBC encryption is a serial mode of operation, because encrypting a block requires the encryption result of the previous block. Therefore, CBC encryption does not allow for hiding the latency of the AES instructions by operating on

```

void AES_128_CBC_Encrypt_Parallel_4_Blocks (...) {
    int i, j, k;
    __m128i tmp, feedback, feedback1, feedback2, feedback3, feedback4;
    __m128i tmp1, tmp2, tmp3, tmp4;
    __m128i RKEY [11];

    for (k=0; k<11; k++) {
        RKEY [k] = _mm_load_si128 ( (__m128i*)&Key_Schedule [4*k]);
    }

    feedback1 = _mm_load_si128 ( (__m128i*)&IV1 [0]);
    feedback2 = _mm_load_si128 ( (__m128i*)&IV2 [0]);
    feedback3 = _mm_load_si128 ( (__m128i*)&IV3 [0]);
    feedback4 = _mm_load_si128 ( (__m128i*)&IV4 [0]);

    for(i=0; i < NBLOCKS; i++) {
        tmp1 = _mm_load_si128 ( (__m128i*)&PLAINTEXT1[i*4]);
        tmp2 = _mm_load_si128 ( (__m128i*)&PLAINTEXT2[i*4]);
        tmp3 = _mm_load_si128 ( (__m128i*)&PLAINTEXT3[i*4]);
        tmp4 = _mm_load_si128 ( (__m128i*)&PLAINTEXT4[i*4]);

        tmp1 = _mm_xor_si128 (tmp1, feedback1);
        tmp2 = _mm_xor_si128 (tmp2, feedback2);
        tmp3 = _mm_xor_si128 (tmp3, feedback3);
        tmp4 = _mm_xor_si128 (tmp4, feedback4);

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

        for(j=1; j <10; j++) {
            tmp1 = _mm_aesenc_si128 (tmp1, RKEY [j]);
            tmp2 = _mm_aesenc_si128 (tmp2, RKEY [j]);
            tmp3 = _mm_aesenc_si128 (tmp3, RKEY [j]);
            tmp4 = _mm_aesenc_si128 (tmp4, RKEY [j]);
        }
        tmp1 = _mm_aesencast_si128 (tmp1, RKEY [10]);
        tmp2 = _mm_aesencast_si128 (tmp2, RKEY [10]);
        tmp3 = _mm_aesencast_si128 (tmp3, RKEY [10]);
        tmp4 = _mm_aesencast_si128 (tmp4, RKEY [10]);

        feedback1 = tmp1;
        feedback2 = tmp2;
        feedback3 = tmp3;
        feedback4 = tmp4;

        _mm_store_si128 ((__m128i*)&CIPHERTEXT1[4*i], tmp1);
        _mm_store_si128 ((__m128i*)&CIPHERTEXT2[4*i], tmp2);
        _mm_store_si128 ((__m128i*)&CIPHERTEXT3[4*i], tmp3);
        _mm_store_si128 ((__m128i*)&CIPHERTEXT4[4*i], tmp4);
    }
}

```

**Fig. 8.** CBC encryption for 4 blocks in parallel (C code using compiler intrinsics)

independent blocks as shown above. However, in some cases it is possible to parallelize CBC encryption if the application needs to operate on multiple independent data streams. One possible example can be disk encryption applications where disk sectors are encrypted independently (not necessarily with the same key). If the software can encrypt multiple sectors in parallel, the application

can enjoy the speedup of a parallel mode. Figure 8 gives a C code snippet for encrypting 4 blocks in parallel, in CBC mode (in this example, using the same key and different IV's).

## 6 More on Software Flexibility and Surprising Usage Models

### 6.1 Supporting RIJNDAEL with Block Size Larger Than 128 Bits

Although the main usage model for the AES instructions is AES, which operates on 128-bit blocks, they can also be used for processing the general RIJNDAEL cipher that supports any block size which is a multiple of 32 bits, from 128 to 256 bits. Figure 9 gives an example for computing a RIJNDAEL-256 round, using the new AES instructions.

### 6.2 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 protection of the AES instructions if they are designed to use the AES transformations. In particular, the AES transformations can be a useful building block for hash functions. For example, the MixColumns transformation provides rapid diffusion and the AES S-box is a good nonlinear mixer. Manipulations on large block sizes could be useful for constructing hash functions, with a long digest size. This concept is already being used in quite a few of the new Secure Hash Function algorithms that have been recently submitted to the NIST cryptographic hash Algorithm Competition (some of the examples from the First Round Candidates list include LANE, SHAMATA, SHAvite-3, ECHO, GrØstl, Lesamnta (512-bit), and Vortex). Some algorithms use the whole AES round as a building block, some only one AES transformations, and some use variants of these transformations.

```

VPBLENDVB xmm3, xmm2, xmm1, xmm5
VPBLENDVB xmm4, xmm1, xmm2, xmm5
PSHUFB xmm3, xmm8
PSHUFB xmm4, xmm8
AESENC xmm3, xmm6
AESENC xmm4, xmm7

```

**Fig. 9.** Using the AES instructions for computing a RIJNDAEL round with a 256-bits block size. Register xmm1 holds the “left” half of RIJNDAEL input state (columns 0–3), xmm2 hold the right half of state (columns 4–7), xmm6 and xmm7 hold the left half and right half of RIJNDAEL round key, respectively. The output state is written into registers xmm1 (left half) and xmm2 (right half). Register xmm8 holds a mask (0x03020d0c0f0e0908b0a050407060100) used for the shuffling step which is necessary to account for the difference in ShiftRows offsets between the 256 (1,3,4) and 128-bit (1,2,3) versions of RIJNDAEL. Register xmm5 holds a mask for VPBLENDVB, selecting bytes 1–3, 6–7, 10–11, and 15 of the RIJNDAEL state from the first source operand, and all other bytes from the second source operand.

Therefore, it is important to note that although the AES instructions perform bundled sequences of AES transformations, each one of these transformations can be isolated by a proper combination of these instructions, and the use of the byte shuffling (PSHUFB instruction). This is shown in Figure 10.

### 6.3 Using the AES Instructions for RAID-6

We show here a surprising usage for the AES instructions for a non cryptographic application.

A Redundant Array of Independent Disks (RAID) combines a multiple physical hard disk drives into a logical drive for purposes of reliability, capacity, or performance. A level 6 RAID (RAID-6) system provides a high level of redundancy allowing recovery from two disk failures. Two syndromes ( $P$  and  $Q$ ) are generated for the data and stored on hard disk drives in the RAID system. The  $P$  syndrome is generated by computing parity information for the data in a strip. The generation of the  $Q$  syndrome requires Finite Field multiplications in  $GF(2^8)$  defined by the reduction polynomial  $x^8 + x^4 + x^3 + x + 1$  (same as the one used for AES). Recovering data and/or  $P$  and/or  $Q$  syndromes requires both  $GF(2^8)$  multiplications and inversions. In a RAID array with  $n$  data disks  $D_0, D_1, D_2, \dots, D_{n-1}$  (for  $n \leq 255$ )  $P$  and  $Q$  are defined by:  $P = D_0 + D_1 + D_2 + \dots + D_{n-1}$ , and  $Q = g^0 \cdot D_0 + g^1 \cdot D_1 + g^2 \cdot D_2 + \dots + g^{n-1} \cdot D_{n-1}$ , where  $g = \{02\}$  is a generator of  $GF(2^8)$ , and  $+$  and  $\cdot$  denote the operations in this field. The computational bottleneck associated with the RAID-6 system is the cost of computing  $Q$ . The performance of the generation of the  $Q$  syndrome may be improved by expressing  $Q$  in its Horner representation  $Q = ((\dots D_{n-1} \dots) \cdot g + D_2) \cdot g + D_1) \cdot g + D_0$ . The difficulty in the related software implementation stems from the fact that traditional processors have poor performance with Finite Fields computations. See [1] for a detailed overview.

We now note that the MixColumns transformation is a matrix multiplication in  $GF(2^8)$ , therefore useful for computing the  $Q$  syndrome. In order to use

<b>Isolating ShiftRows</b>	PSHUFB xmm0, 0x0b06010c07020d08030e09040f0a0500
<b>Isolating InvShiftRows</b>	PSHUFB xmm0, 0x0306090c0f0205080b0e0104070a0d00
<b>Isolating MixColumns</b>	AESDECLAST xmm0, 0x00000000000000000000000000000000 AESENC xmm0, 0x00000000000000000000000000000000
<b>Isolating InvMixColumns</b>	AESENCLAST xmm0, 0x00000000000000000000000000000000 AESDEC xmm0, 0x00000000000000000000000000000000
<b>Isolating SubBytes</b>	PSHUFB xmm0, 0x0306090c0f0205080b0e0104070a0d00 AESENCLAST xmm0, 0x00000000000000000000000000000000
<b>Isolating InvSubBytes</b>	PSHUFB xmm0, 0x0b06010c07020d08030e09040f0a0500 AESDECLAST xmm0, 0x00000000000000000000000000000000

Fig. 10. Isolating the AES transformations using combinations of AES instructions

```

__declspec (align(16)) unsigned int zero_ [4] =
    {0x0, 0x0, 0x0, 0x0};
__declspec (align(16)) unsigned int mask1 [4] =
    {0xff02ff00,0xff06ff04,0xff0aff08, 0xff0eff0c};
__declspec (align(16)) unsigned int mask2 [4] =
    {0x03ff01ff,0x07ff05ff,0x0bff09ff, 0x0fff0dff};
__declspec (align(16)) unsigned int mask3 [4] =
    {0x01000302,0x05040706,0x09080b0a, 0x0d0c0f0e};

void RAID6_1_block_in_parallel (...) {
    int ind1;
    __m128i MASK1, MASK2, MASK3, ZERO;
    __m128i XMM0, XMM1, XMM2;

    MASK3 = _mm_load_si128 ((__m128i*)&mask3[0]);
    MASK2 = _mm_load_si128 ((__m128i*)&mask2[0]);
    MASK1 = _mm_load_si128 ((__m128i*)&mask1[0]);
    ZERO = _mm_load_si128 ((__m128i*)&zero_[0]);

    for (ind1=0; ind1 < NBLOCKS; ind1++) {
        XMM0 = _mm_load_si128 ((__m128i*)&DATA[4*ind1]);
        XMM1 = _mm_shuffle_epi8(XMM0, MASK1);
        XMM1 = _mm_aesdeclast_si128 (XMM1, ZERO);
        XMM2 = _mm_shuffle_epi8(XMM0, MASK2);
        XMM0 = _mm_shuffle_epi8(XMM0, MASK3);
        XMM1 = _mm_aesenc_si128(XMM1, ZERO);
        XMM2 = _mm_aesdeclast_si128 (XMM2, ZERO);
        XMM1 = _mm_shuffle_epi8(XMM1, MASK1);
        XMM2 = _mm_aesenc_si128(XMM2, ZERO);
        XMM2 = _mm_shuffle_epi8(XMM2, MASK2);
        XMM2 = _mm_xor_si128(XMM2, XMM1);
        XMM0 = _mm_xor_si128(XMM0, XMM2);

        _mm_store_si128 ((__m128i*)&RES[4*ind1], XMM0);
    }
}

```

**Fig. 11.** Using the AES instructions for RAID-6: multiplying 16 bytes by  $\{02\}$

the AES instructions, the MixColumns transformation needs to be isolated, as explained above. This transformation operates separately on the 4 columns of the state. If a column (32 bits) is denoted by the four bytes  $[d, c, b, a]$ , then the output  $[d', c', b', a']$  of MixColumns is  $a' = (\{02\} \cdot a) + (\{03\} \cdot b) + c + d$ ;  $b' = a + (\{02\} \cdot b) + (\{03\} \cdot c) + d$ ;  $c' = a + b + (\{02\} \cdot c) + (\{03\} \cdot d)$ ;  $d' = (\{03\} \cdot a) + b + c + (\{02\} \cdot d)$  denoted in shorthand by  $[3a + b + c + 2d, a + b + 2c + 3d, a + 2b + 3c + d, 2a + 3b + c + d]$ . If the bytes  $b, d$  (odd positions) are set to 0, then the result of MixColumns becomes  $[3a + c, a + 2c, a + 3c, 2a + c]$ , and with the PSHUFB instruction odd position bytes can be zeroed to yield  $[0, a + 2c, 0, 2a + c]$ . If this result is XOR-ed with  $[0, a, 0, c]$  (a shuffled version of the input), the final result is  $[0, 2c, 0, 2a]$ , that is, two of the 4 bytes of the column were multiplied by  $\{02\}$ . Similar operations can be applied to the even-positioned bytes of the state. Figure 11 shows a code snippet that uses the AES instructions for RAID-6 (here, for clarity and brevity the code operates on a single block at a time. Operating on multiple blocks in parallel, improves the performance as explained above).

## 7 Conclusion

This paper provided some details and insights on Intel’s new AES instructions which are expected to be widely used for security and privacy, by a wide range of applications and operating systems.

The AES instructions provide a substantial performance speedup to bulk data encryption and decryption. Exact performance measurements will be made available as soon as processors with these instructions are released. However, we can indicate that when using parallelizable modes of operation (e.g., CBC decryption, CTR, and CTR-derived modes GCM, XTS), the performance speedup could exceed an order of magnitude over the current performance of software-only AES implementations. In scenarios where pipelined operation is impossible, for example in CBC encryption, operating on a single buffer, the performance speedup would still be significant, around 2–3 times over software implementation. Note that AES implementations using the new instructions are inherently protected against the software side channel attacks associated with AES implementations based on table-lookup.

The paper showed some of the advantages of the AES instructions and how they can be used flexibly and efficiently.

An important observation that we pointed out was that due to the out-of-order execution capabilities of modern processors, hardware pipelining, and software techniques, parallel modes of operation can achieve a much higher throughput than serial modes. This is one point to consider when selecting modes of operation in future cryptosystems. For example, AES-GCM may become a favorable mode for achieving secrecy and authentication. In this context, we also mention that, together with the AES instructions, another instruction for computing carry-less (polynomial) multiplications (called PCLMULDQ) is released. This could give further speedup to AES-GCM (see [5]).

**Acknowledgements.** Many people contributed to the concepts, the studies, the definition of the architecture, and to the micro-architectural implementation. 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.

## References

1. Anvin, H.P.: The mathematics of RAID-6, <http://www.kernel.org/pub/linux/kernel/people/hpa/raid6.pdf>
2. Bernstein, D.J., Schwabe, P.: New AES Software Speed Records. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 322–336. Springer, Heidelberg (2008)

3. Brickell, E., Graunke, G., Neve, M., Seifert, J.P.: Software mitigations to hedge AES against cache based software side channel vulnerabilities, IACR ePrint Archive, Report 2006/052 (2006), <http://eprint.iacr.org/2006/052>
4. Gladman, B.: Implementations of AES (Rijndael) in C/C++ and assembler, [http://www.gladman.me.uk/cryptography\\_technology/rijndael](http://www.gladman.me.uk/cryptography_technology/rijndael)
5. Gueron, S., Kounavis, M.E.: Carry-Less Multiplication and Its Usage for Computing the GCM Mode, [http://softwarecommunity.intel.com/isn/downloads/intelavx/Carry-Less-Multiplication-and-The-1.GCM-Mode\\_WP%20.pdf](http://softwarecommunity.intel.com/isn/downloads/intelavx/Carry-Less-Multiplication-and-The-1.GCM-Mode_WP%20.pdf)
6. Lipmaa, H.: Fast Software Implementations of SC 2000. In: Chan, A.H., Gligor, V.D. (eds.) ISC 2002. LNCS, vol. 2433, pp. 63–74. Springer, Heidelberg (2002)
7. Lipmaa, H.: AES / Rijndael: speed, <http://research.cyber.ee/~lipmaa/research/aes/rijndael.html>
8. Matsui, M.: How far can we go on the x64 processors? In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 341–358. Springer, Heidelberg (2006)
9. Matsui, M., Fukuda, S.: How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 398–412. Springer, Heidelberg (2005)
10. Microsoft, BitLocker, <http://www.bitlocker.com>
11. National Institute of Standards and Technology (NIST), FIPS-197: Advanced Encryption Standard (November 2001), <http://www.itl.nist.gov/fipspubs/>
12. OpenSSL: the open-source toolkit for SSL/TLS, <http://www.openssl.org>
13. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006)
14. Tillich, S., Großschädl, J.: Instruction Set Extensions for Efficient AES Implementation on 32-bit Processors. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 270–284. Springer, Heidelberg (2006)

## A Code Sequences for AES-192 and AES-256 Key Expansion

<pre> movdqu xmm1, XMMWORD PTR Key movq xmm3, QWORD PTR Key_ movdqu XMMWORD PTR Key_Sched, xmm1 movq QWORD PTR[Key_Sched+0x10], xmm3 mov ecx, OFFSET Key_Sched+24  AESKEYGENASSIST xmm2, xmm3, 0x1 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x2 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x4 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x8 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x10 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x20 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x40 call key_expansion_192 AESKEYGENASSIST xmm2, xmm3, 0x80 call key_expansion_192 jmp END; </pre>	<pre> key_expansion_192: pshufd xmm2, xmm2, 0x55 vpslldq xmm4, xmm1, 0x4 pxor xmm1, xmm4 pslldq xmm4, 0x4  pxor xmm1, xmm4 pslldq xmm4, 0x4  pxor xmm1, xmm4 pxor xmm1, xmm2 pshufd xmm2, xmm1, 0xff vpslldq xmm4, xmm3, 0x4  pxor xmm3, xmm4 pxor xmm3, xmm2 movdqu XMMWORD PTR [rcx], xmm1 add rcx, 0x10 movdqu XMMWORD PTR [rcx], xmm3 add rcx, 0x8 ret  END: </pre>
<pre> movdqu xmm1, XMMWORD PTR Key movdqu xmm3, XMMWORD PTR Key_ movdqu XMMWORD PTR Key_Sched, xmm1 movdqu XMMWORD PTR[Key_Sched+0x10], xmm3 mov rcx, OFFSET Key_Sched+0x20  AESKEYGENASSIST xmm2, xmm3, 0x1 call key_expansion_256 AESKEYGENASSIST xmm2, xmm3, 0x2 call key_expansion_256 AESKEYGENASSIST xmm2, xmm3, 0x4 call key_expansion_256 AESKEYGENASSIST xmm2, xmm3, 0x8 call key_expansion_256 AESKEYGENASSIST xmm2, xmm3, 0x10 call key_expansion_256 AESKEYGENASSIST xmm2, xmm3, 0x20 call key_expansion_256 AESKEYGENASSIST xmm2, xmm3, 0x40 call key_expansion_256 jmp END; </pre>	<pre> key_expansion_256: pshufd xmm2, xmm2, 0xff vpslldq xmm4, xmm1, 0x4 pxor xmm1, xmm4 pslldq xmm4, 0x4 pxor xmm1, xmm4 pslldq xmm4, 0x4 pxor xmm1, xmm4 pxor xmm1, xmm2 movdqu XMMWORD PTR [rcx], xmm1 add rcx, 0x10 cmp rcx, OFFSET Key_Schedule+0xf0 jz ReachedLastKey AESKEYGENASSIST xmm4, xmm1, 0 pshufd xmm2, xmm4, 0xaa vpslldq xmm4, xmm3, 0x4 pxor xmm3, xmm4 pslldq xmm4, 0x4 pxor xmm3, xmm4 pslldq xmm4, 0x4 pxor xmm3, xmm4 pslldq xmm4, 0x4 pxor xmm3, xmm4 movdqu XMMWORD PTR [rcx], xmm3 add rcx, 0x10 ReachedLastKey: ret  END: </pre>

**Fig. 12.** AES-192 and AES-256 key expansion

**Remark:** There are several ways for expanding the key, using AESKEYGENASSIST. These given examples use new Intel AVX instructions (<http://software.intel.com/sites/avx/>) with a nondestructive source. For example, instead of (A) `movdqu xmm3, xmm1; pslldq xmm3, 0x4` we use (B) `vpslldq xmm3, xmm1, 0x4`. AVX extensions will be introduced only in the 2010 processors, and therefore option (B) would not be valid in the 2009 processors that require the form (A). The changes from form (B) to form (A) are straightforward.