# The Intel AES Instructions Set and the SHA-3 Candidates

R. Benadjila[1], O. Billet[1], S. Gueron[2,3], and M.J.B. Robshaw[1]

[1] Orange Labs, Issy les Moulineaux, France
{ryad.benadjila,olivier.billet,matt.robshaw}@orange-ftgroup.com
[2] University of Haifa, Israel
[3] Intel Corporation, Haifa, Israel
shay.gueron@intel.com, shay@math.haifa.ac.il

**Abstract.** The search for SHA-3 is now well-underway and the 51 accepted submissions reflect a wide variety of design approaches. A significant number are built around Rijndael/AES-based operations and, in some cases, the AES round function itself. Many of the design teams have pointed to the forthcoming Intel AES instructions set, to appear on Westmere chips during 2010, when making a variety of performance claims. In this paper we study, for the first time, the likely impact of the new AES instructions set on all the SHA-3 candidates that might benefit. As well as distinguishing between those algorithms that are AES-based and those that might be described as AES-inspired, we have developed optimised code for all the former. Since Westmere processors are not yet available, we have developed a novel software technique based on publicly available information that allows us to accurately emulate the performance of these algorithms on the currently available Nehalem processor. This gives us the most accurate insight to-date of the potential performance of SHA-3 candidates using the Intel AES instructions set.

## 1 Introduction

Intel has announced that a new AES instructions set, denoted AES-NI in this paper,[4] will be introduced in new processors such as Westmere and available early in 2010. These instructions will provide resistance to a range of software side-channel attacks [3, 30] and offer significant performance benefits for encryption and decryption using AES [24]. Simultaneously the NIST SHA-3 effort [25] to establish a new cryptographic hash algorithm is well-underway and several teams of submitters have used AES-like transformations as a cryptographic building block. Several of these teams have explicitly expressed the assumption that their hashing algorithms could take advantage of AES-NI and thereby enjoy significant performance benefits. Since the Westmere processor is still unavailable, there have been no substantive efforts to assess the possible implications of this

[4] For "new instructions".

important issue. In this paper, we provide the first quantitative analysis that estimates the likely impact of the Intel AES instructions set on SHA-3 candidates.

The first step is to identify which SHA-3 candidates should be considered, and this is not as straightforward as it might appear. AES-NI can be used in different combinations to carry out different transformations, and so AES-NI might be used in many more ways than would naïvely be expected. As a result, there are submissions for which the variant that provides (say) 256-bit digests gains from AES-NI, while the same algorithm providing a 512-bit digest cannot.

The second step is to develop a sound methodology for implementing the different algorithms, optimising them, and measuring their performance. Clearly this is a challenge when Westmere processors are unavailable. So we developed new techniques from publicly available information—in effect, uncovering the behavior of AES-NI—and this allowed us to emulate Westmere behavior on the publicly-available Nehalem chips. While this might appear to detract from the value of the performance figures we derive, the level of validation and confirmation that took place during this work makes us confident that our results are close to the Westmere reality.

Our sole goal in this paper has been to compare the performance of SHA-3 candidates when using AES-NI. To this end, we have set aside cryptanalytic discussions [10] and we have implemented and optimised all the algorithms that we believe might benefit from AES-NI. While the authors of this paper are independent (co-)submitters of two SHA-3 proposals, we have strived to be fair and consistent. In addition, all the code is publicly available via [29] and we welcome interested parties to download and improve upon it. When Westmere processors appear, the same samples can be used for real silicon running AES-NI.

## 2 The Intel AES Instructions

To start we provide a brief description of the Intel AES instructions, and complete details can be found in [13, 14]. Intel's AES instructions set consists of six instructions, four of which `aesenc`, `aesenclast`, `aesdec`, and `aesdeclast` are designed to support data encryption and decryption. The names of these instructions are short for AES encryption (inner and last) round and AES decryption (inner and last) round, see Table 3 from Appendix A. These instructions have register/register and register/memory variants.

There are two other instructions for the AES key expansion but they seem to be of little use to the SHA-3 submissions and are omitted from this paper.

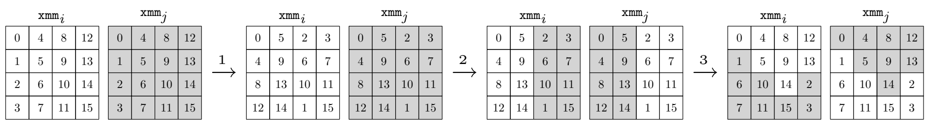
### 2.1 What operations can we use AES-NI for?

Clearly, AES instructions can be used whenever a SHA-3 proposal uses one of the internal or final AES encryption (or decryption) rounds. But they can be used more widely than this. For instance, calling `aesdeclast` and `aesenc` back-to-back, both with a zeroed second operand, is functionally equivalent to performing AES MixColumns on the first operand, see Appendix A.

In fact if we use the `pshufb` instruction which shuffles bytes in a 128-bit word, see Appendix A, then we can isolate all of the AES-constituents using AES-NI [14], namely:

$$\begin{array}{lll} \mathsf{SubBytes}\,, & \mathsf{ShiftRows}\,, & \mathsf{MixColumns}\,, \\ \mathsf{InvSubBytes}\,, & \mathsf{InvShiftRows}\,, & \mathsf{InvMixColumns}\,. \end{array}$$

To illustrate the versatility this gives us, we combine standard `xmm` instructions with AES-NI to perform encryption with Rijndael [8] operating on 256-bit blocks. The plaintext is stored in $\texttt{xmm}_i$ and $\texttt{xmm}_j$, but AES-NI cannot be used directly since half the bytes of $\texttt{xmm}_i$ must be swapped with half the bytes of $\texttt{xmm}_j$. However, this swap can be efficiently implemented using two `pshfub` (1) to pack the bytes to-be-swapped into two 32-bit words, two `pblendw` (2) to swap the 32-bit words, and two `pshufb` (3) to re-order the bytes giving, in total, the following state permutation:



After this, `aesenc` can be applied in parallel to $\texttt{xmm}_i$ and $\texttt{xmm}_j$, thereby giving the appropriate ShiftRows for the large state, and Rijndael encryption on a larger state has been emulated. Techniques like these are important to us since it is possible that several SHA-3 candidates that do not use the complete AES round, or they use a larger state, might still benefit from AES-NI.

## 2.2 The "In-Scope" SHA-3 Candidates

**Table 1.** The SHA-3 submissions with substantial Rijndael-based components (see text). Those that might benefit from Intel's AES-NI, for different hash output lengths, are indicated with a checkmark.

| *Algorithm* | *224*-bit | *256-bit* | *384-bit* | *512-bit* |
|---|---|---|---|---|
| ARIRANG | ✓ | ✓ | no | no |
| CHEETAH | ✓ | ✓ | no | no |
| ECHO | ✓ | ✓ | ✓ | ✓ |
| FUGUE | no | no | no | no |
| GRØSTL | no | no | no | no |
| LANE | ✓ | ✓ | ✓ | ✓ |
| LESAMNTA | ✓ | ✓ | ✓ | ✓ |
| LUX | ✓ | ✓ | no | no |
| SHAVITE-3 | ✓ | ✓ | ✓ | ✓ |
| VORTEX | ✓ | ✓ | ✓ | ✓ |

Obviously SHA-3 candidates that use the AES round as a building block can benefit from using AES-NI. In addition, algorithms that use the AES S-box along with some byte shuffling with or without the AES MDS mixing matrix can benefit. One can also apply these operations to larger states, as we have seen for Rijndael with 256-bit blocks. The main problems in using AES-NI tend to arise when designs move away from the AES MDS matrix. Generally speaking, this dramatically limits any potential gain from AES-NI, particularly since most optimised assembly implementations would incorporate the MDS matrix operation, potentially combined with other operations, into table look-ups.

There are four submissions that directly, and transparently, use AES rounds for all hash output lengths. These are ECHO [2], LANE [18], SHAVITE-3 [4], and VORTEX [23]. For these algorithms it is clear that we can directly use AES-NI. There are others that are clearly inspired by Rijndael-like techniques in their construction. These include CHEETAH [22], FUGUE [15], GRØSTL [12], LESAMNTA [16], LUX [27], and TWISTER [11]. The submission SHAMATA [1] has already been withdrawn, and while some other surveys [5] describe SARMAL [31] as being AES-inspired, a non-AES S-box and MDS mixing layer take it out-of-scope.

While LESAMNTA offers advantages for 256- and 512-bit hash outputs, it is interesting that only the 256-bit versions of CHEETAH and LUX benefit from AES-NI. By contrast, it appears that no variant of FUGUE, GRØSTL, or TWISTER are likely to benefit. These algorithms use a very different MDS mixing matrix to the AES and, as a result, end-up being too distant to use AES-NI in any efficient way. So even though a combination of AES-NI instructions could be used to isolate the S-box operations for FUGUE and GRØSTL, say, the table look-ups typically used for the MDS operations in current optimised implementations mean that there is no easy way for these algorithms to benefit from AES-NI.

Finally, even though the submission ARIRANG [6] is quite different from the Rijndael-based constructions, it might potentially benefit from AES-NI. We have therefore included it in our considerations and Table 1 summarizes the (alphabetically ordered) list of algorithms and hash output lengths that we consider.

## 3 Implementation and Measurements

Obviously the best way to get performance timings is to write the appropriate code, run it on a Westmere processor (the first with AES-NI), and measure the performance. However, since this processor is not yet available, we propose a new methodology that can be used to get an accurate emulation of AES-NI. Our methodology relies on the fact that Westmere (formerly Nehalem-C) and Nehalem processors share the same micro-architecture. This means that if we can find suitable instructions patterns that behave exactly as AES-NI instructions, we will get very good estimates for the future performance of AES-based SHA-3 candidates on a Westmere processor, but using today's Nehalem processor.

Previously, a substitution instruction was proposed [23] for future processors. However this substitution does not exhibit the correct behaviour for Westmere and can give misleading results, see Section 3.1 and Appendix B. Here, we provide a particularly accurate replacement instructions pattern for `aesenc` and we explain how to derive it from publicly available information only.

## 3.1 Replacement instructions pattern

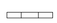
The first step is to understand the exact behavior of the AES-NI instructions at the micro-operation ($\mu$op) level,[5] in particular that of `aesenc` and `aesenclast`.

An Intel code analyzer tool (IACA [21]) is publicly available and gives the following information about `aesenc` (`aesdec` yields the same output):

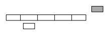
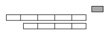
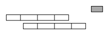
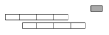
```
Total Latency:   6 Cycles;    Total number of Uops:  3

| Num of |          Ports pressure in cycles             |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    3   |  2 |    |    |    |    |    |    |    |  1 | CP | aesenc xmm1, xmm0
```

(In this trace, '`DV`' stands for the divider pipe of port 0, '`D`' for the data fetch pipe of ports 2 and 3. Additionally, an '`X`' in the trace will be used to denote the possible ports a $\mu$op can be dispatched to.)

This shows that `aesenc` consists of three $\mu$ops, two of which are dispatched to a unit on port 0 and one which is dispatched to a unit on port 5, and that the instruction's latency is 6 cycles. However, this information is too coarse to provide hints for the right instructions pattern replacement: we need to derive the exact scheduling of these $\mu$ops. In what follows, we represent $\mu$ops by bars for which the length varies according to their latency. The gray bars denote the $\mu$ops on port 5 while the white ones denote the $\mu$ops on port 0. Hence is a 2 cycle $\mu$op on port 5 and  is a 3 cycle $\mu$op on port 0.

From Intel's white paper [13] we know that AES-NI are highly parallelizable. This discards the sequential $\mu$op patterns on port 0. Moreover, the white paper explains (see Fig. 9 and 15) that `aesdec` is structured using the equivalent inverse cipher (described in Appendix B), which is confirmed by a IACA trace identical to that of `aesenc` displayed above (see Appendix B). This leads us to assume that the $\mu$op on port 5 is the exclusive-or with the key, which is corroborated by the purpose of unit 5, see [19]. Therefore, the $\mu$op on port 5 runs in cycle 6 and requires that $\mu$ops from port 0 are finished.

Intel's optimization reference manual [19] gives additional information on the possible $\mu$op latencies and throughput for each port on the Nehalem micro-architecture. In particular, we see that $\mu$ops dispatched on port 0 can only have latencies 1, 4, or 5 cycles, and that $\mu$ops on port 5 all have a 1 cycle latency. Since `aesenc` has a total latency of 6 cycles, this only leaves the following possible patterns: , , and . (Two $\mu$ops cannot start at the same cycle in the same unit but a $\mu$op is started as soon as possible to maximize the overall throughput). It is impossible that a $\mu$op on port 0 performs the SubBytes and/or ShiftRows step while it runs in parallel with the other $\mu$op performing the MixColumns step which would then need the output of the first $\mu$op. So both $\mu$ops on port 0 perform at least one of the four MixColumn multiplications of the MixColumns step. The most natural way of doing this is to symmetrically split the computation on two independent halves of the state. In this case, the two $\mu$ops on port 0 have the same latency, which only leaves the  pattern. This is again supported by the IACA trace of `aesimc` instruction, as well as the choice of inverse equivalent cipher for `aesdec`.

[5] Instructions are split into micro-operations and dispatched to specialized CPU units.

Now we turn to the replacement instructions set which would give exactly the same $\mu$op-behavior as the instruction `aesenc reg, reg`. A previously proposed replacement [23, 17] is not appropriate for Westmere (see Appendix B). Instead, a sequence that closely simulates the $\mu$op behavior of `aesenc` xmm$_i$, xmm$_j$ is:

movdqu xmm$_k$, xmm$_i$
mulps  xmm$_i$, xmm$_j$
mulps  xmm$_k$, xmm$_j$
xorps  xmm$_i$, xmm$_k$

For now, let us ignore the `movdqu` instruction. The IACA trace displayed below shows that the last three instructions of the replacement behave exactly as the `aesenc xmm0, xmm1` instruction with a latency of 6 cycles. It yields two identical and independent $\mu$ops (they both come from `mulps`) on port 0, a 1 cycle $\mu$op on port 5 which is forced to start after the two $\mu$ops on port 0 since `xorps` has a 1 cycle $\mu$op on port 0 together with a dependency on register `xmm2`:

```
Total Latency:   6 Cycles;    Total number of Uops:  4

| Num of |          Ports pressure in cycles          |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    1   |  X |    |  1 |    |    |    |    |    |  X | CP | movdqu xmm2, xmm0
|    1   |  1 |    |    |    |    |    |    |    |    |    | mulps  xmm0, xmm1
|    1   |  1 |    |    |    |    |    |    |    |    | CP | mulps  xmm2, xmm1
|    1   |    |    |    |    |    |    |    |    |  1 | CP | xorps  xmm0, xmm2
```

The reader might wonder why we added the `movdqu` instruction to the beginning of the replacement: by introducing a dependency on `xmm0`, we try to prevent the processor from re-ordering the instructions at the prefetch and re-order step. Hence, `movdqu` acts as a fence and ensures that the replacement fragment exhibits a similar atomic behavior as `aesenc`. Since `movdqu` only has a latency of one cycle and can be dispatched on port 0, 1, or 5, it will in most cases execute on port 1 in parallel of the other $\mu$ops—and does not interfere with the replacement, and rarely on port 5 or 0 which would add one cycle to the replacement latency.

Note however, that though the replacement allows for a very good simulation of `aesenc` in terms of latency, throughput, and port behavior, it does introduce a significant issue: the use of a third register xmm$_k$ ($k = 2$ in IACA's trace) might interfere with code surrounding the replacement by introducing false dependencies. We took extra care in our implementations to avoid these when using the replacement. This was not an easy task, especially for those SHA-3 candidates that make heavy use of AES-NI parallelism such as ECHO and LANE.

Another potential issue is that the `aesenc` instruction is 5 to 10 bytes long depending on the variant whereas our replacement is 13 to 22 bytes. This can lead to an efficiency penalty as the prefetch buffer of the Nehalem micro-architecture has a size of 16 bytes. However an experiment (see Appendix B) shows that the size of replacement is unlikely to be a significant factor.

Finally, we refer the reader to Appendix B for a justification of our choice of the following replacement for memory-based variants like `aesenc` xmm$_i$, `[mem]`:

movdqu xmm$_k$, xmm$_i$
mulps  xmm$_i$, [mem]
mulps  xmm$_k$, xmm$_j$
xorps  xmm$_i$, xmm$_k$

as well as for a discussion regarding replacements for other AES-NI instructions.

### 3.2 Timing methodology

For each in-scope candidate and for each hash output length, we implemented two versions of the submission. These were identical in every way, except one had AES-NI instructions and was used to ensure the correctness of our AES-NI optimized implementation against the NIST-submitted test vectors with Intel's Emulator [20]; the other had AES-NI instructions substituted with their replacements allowing it to run on a Nehalem to derive performance estimates.

To get consistent results over the candidates, we measured the number of cycles (using `rdtsc` instructions and averaging over more than $10^8$ samples to get stable results) taken by the compression function of each algorithm on the same Nehalem machine running Linux. However NIST's API was fully implemented to check correctness and, in many cases, these were taken from the reference code sent to NIST by the submitters. To eliminate as much noise as possible from the OS, high priority scheduling was allocated to the measured code. All algorithms were implemented by the same programmers, providing a somewhat uniform level of optimization.

## 4 Candidate Descriptions and AES-NI Implementations

In this section we consider the design, and discuss the implementation, of the in-scope candidates. Full details of the algorithms can be found in the respective algorithm descriptions, so we only give a brief overview of their functionality along with insights into their design with regards to AES-NI. Our implementation proposals will be available from our website [29].

**ARIRANG** is a single-pipe compression function-based proposal. The bulk of the computation in the compression function consists of the 40-step expansion of a 512-bit message block, which is highly efficient in general purpose registers and can be pre-computed, and a StepFunction that is repeated 40 times. StepFunction requires eight exclusive-ors, four fixed rotations, and two calls to a function $G^{256}$ that uses elements of the AES. For longer hash outputs, the equivalent function $G^{512}$ uses a larger MDS matrix that cannot be emulated using AES-NI, and so any potential gain is restricted to 256-bit outputs.

However, the extent of this gain is very limited since ARIRANG uses $\frac{1}{4}$ of an AES round as a building block, but the latency cost of `aesenc` while only performing $\frac{1}{4}$ of an AES round means that the performance of AES-NI, when compared to the use of lookup tables, is not competitive. Attempts to parallelize two of the $\frac{1}{4}$ AES rounds introduced too many overheads. We conclude that AES-NI is unlikely to offer any substantial benefits to ARIRANG.

**CHEETAH** is a single-pipe compression function-based proposal. The compression function consists of two strands of computation: a message-dependent EXPANDED BLOCK is generated which provides a key-like input to encrypt the INTERNAL STATE. While the computations on EXPANDED BLOCK and INTERNAL

STATE are both Rijndael-inspired, the former uses a different non-AES MDS matrix that is hard to emulate. Thus this key derivation is unlikely to benefit from AES-NI and the use of look-up tables seems better suited.

For operations on the INTERNAL STATE, the 224- and 256-bit versions of CHEETAH use an operation InternalRound that can be emulated using AES-NI. However, the inherent sequential nature of the rounds and the fact that AES-NI cannot be used in the most straightforward way means that while there are gains, they are not as significant as they might be for some other submissions.

For the 384- and 512-bit versions, the operation InternalRound is modified to use a larger MDS matrix that, once again, cannot exploit AES-NI. So for these larger outputs, there is unlikely to be any gain with AES-NI.

**ECHO** is a double-pipe compression-based hash function. The 224- and 256-bit (resp. 384- and 512-bit) versions encrypt a sixteen 128-bit words state in eight (resp. ten) rounds of a compression function calculation. The encryption round applies two AES rounds to each word of the state with a counter or salt as a key, followed by a BIG.MixColumns MDS and row shift operation that provides mixing across the entire state. For all hash output lengths, ECHO can benefit from AES-NI and, while ECHO is primarily a double-pipe compression-based hash function, a simple single-pipe variant was announced at the first NIST workshop. We therefore include it in our considerations.

The AES encryption rounds are directly performed with `aesenc` with pre-computed keys in memory. This allows the algorithm to take full advantage of the AES-NI parallelism. The BIG.MixColumns operation however cannot further benefit from AES-NI, though it is based on MixColumns. As ECHO encryption round does not vary with the output length, the same optimizations apply.

**LANE** is a single-pipe compression function-based hash function. COMPRESS consists of a message expansion, a set of six P-PERMUTATIONS, and then a set of two Q-PERMUTATIONS. As both sets of permutations are based on the AES round, LANE benefits from AES-NI at all hash function output lengths.

Both PERMUTATIONS are made of $L = 2$ (resp. $L = 4$) lines of AES rounds for hash outputs of 256 (resp. 512) bits and after each round of AES in each line, an operation SwapColumns mixes the $L$ computation strands. LANE therefore offers two levels of parallelism: the P- and Q-PERMUTATIONS and the lines inside the permutations. The latter does not allow to take full advantage of AES-NI parallelism as SwapColumns breaks the instructions flow so we use the two levels of parallelism simultaneously: we compute an AES round for each of the $6L$ lines of the P-PERMUTATIONS in parallel before applying SwapColumns in each P-PERMUTATION, and do the same for the Q-PERMUTATIONS. (The code is completely unrolled and all keys are precomputed.)

For 256-bit outputs, the state nicely fits the available `xmm` registers. But for 512-bit outputs, the state does not fit anymore and only three P-PERMUTATIONS are computed in parallel instead of all six before. This, in itself, does not change the AES-NI throughput as the number of lines is doubled in each PERMUTATION and thus the same number of AES rounds as before is performed in parallel. However, the 512-bit version of SwapColumns imposes an additional overhead.

**LESAMNTA** is a single-pipe compression function-based hash function. The underlying block cipher has the general topology of an unbalanced Feistel cipher; at each round two strands of the eight that comprise the cipher state are updated using a message dependent “subkey” and the round function $f_{256}$ (resp. $f_{512}$) for the 256-bit (resp. 512-bit) hash output. The subkey generation and the $f_{256}$ and $f_{512}$ functions in the encryption path all involve AES-like operations and LESAMNTA can potentially benefit from AES-NI.

For the 256-bit version, the key schedule poses few problems. However, one difficulty for encryption path is that the AES-like transformations operate on 64-bit values and the MDS matrix is distinct from that of AES. The MDS matrix $\left(\begin{smallmatrix} 2 & 1 \\ 1 & 2 \end{smallmatrix}\right)$ that is used is however a submatrix of MixColumn and so inserting zero bytes at the entry of the appropriate MixColumns entries will allow to perform the AES-like transformation using AES-NI. This can be achieved with the sequence: `pshufb`, `pxor` with a particular constant, `aesenc`, and `pshufb`. Note that in this case, `aesenc` is used at $\frac{1}{2}$ of its normal efficiency.

In the case of 512-bit hash outputs, the AES-like transformation in the key schedule involves an MDS that is too different from MixColumns, and so AES-NI is not really of any use there: the keys are therefore precomputed in a classical way. However, on the encryption side the round functions now use the full AES round, which gives nice advantages.

For both sets of outputs, it is possible to use the unbalanced nature of the Feistel construction to perform four $f$ functions in parallel for both output sizes. In the 256-bit version, this carries a greater benefit: the four instances of the sequence preparing the data mentioned above can also be grouped to increase the overall throughput.

**LUX** is a stream-cipher based hash function that uses two banks of cipher state; the BUFFER and the CORE. At each iteration a block of message is input to both the BUFFER and CORE, both of which are then updated with information being passed between them. Sixteen blank rounds of computation seal the hashing process after the last block of message has been processed. While the BUFFER transformation is very simple, the CORE transformation is built on Rijndael-like operations. And it is the Rijndael-like operations in the CORE that are the most time-consuming parts of LUX, with mixing of the CORE and BUFFER requiring only a few, simple `xmm` instructions.

For all hash output lengths, the CORE transformation operates on a larger state than we find in the AES. However for 256-bit hash outputs it is equivalent to Rijndael operating on 256-bit blocks and techniques described in Section 2.1 can be used. Thus LUX with 256-bit outputs will benefit from AES-NI.

When used to generate longer hash outputs, however, LUX changes the form of the MDS transformation in such a way that it cannot easily be emulated using AES-NI. It appears for these longer outputs that AES-NI will not offer any advantage. In fairness, the optimised implementations of LUX for 512-bit outputs are already extremely competitive.

As an aside on the timing methodology, it is worth observing that we implemented sixteen iterations of the classical compression function found in LUX as a single COMPRESS operation. This avoided buffer rotations and helped treat LUX in a way that was more consistent with the other algorithms.

**SHAVITE-3** is a single-pipe compression function-based design, with the compression function being built closely on a Feistel cipher. The round function for this Feistel cipher is built directly from an AES round, and the accompanying message expansion also uses the AES round function. As a result, all hash output sizes can expect to benefit from AES-NI.

For the 256-bit hash output, the round function for the 12-round Feistel cipher consists of three rounds of the AES and we can therefore use AES-NI directly. To avoid any interaction with the memory, it is much more efficient to perform key derivation inside the `xmm` registers. Key derivation produces 36 subkeys of 128 bits using a combination of a non-linear layer based on four `aesenc` operations and a linear layer. It is possible to interleave key derivation with encryption since there are sufficient registers. The linear part of the key derivation only requires a few `xmm` manipulations (if handled properly) while the four AES rounds in the key schedule can be performed in parallel. The Feistel round function involves three AES rounds, but this time they are chained. SHAVITE-3 derives a significant benefit from avoiding memory access.

For the 512-bit hash output, the underlying 14 rounds block cipher is a generalised Feistel network. At each round there are two parallel invocations of four AES rounds. Now, however, key derivation produces new 128-bit words in sets of eight, rather than four, and so this needs to be performed in place while keeping the rest of the state in registers. The linear part of key derivation can still be implemented efficiently and the eight AES rounds can be parallelized. Within the encryption operation, there are now two Feistel round functions, each with four dependent AES rounds but these can be interleaved, increasing the throughput slightly. SHAVITE-3 is very closely built around the AES round operation and substantially gains from AES-NI.

**VORTEX** is a single-pipe compression function-based design that uses the enveloped Merkle-Damgård construction and builds upon MDC-2 [7]. The building blocks of VORTEX are Rijndael rounds on 128-bit blocks for VORTEX-256 and Rijndael rounds on 256-bit blocks for VORTEX-512. Cross-mixing between the 128-bit strands (resp. 256-bit strands for VORTEX 512) is multiplication-based. The parameter $M_T$ determines whether integer multiplication ($M_T = 1$) or carry-less multiplication ($M_T = 0$) is used. A motivation behind VORTEX was to directly exploit AES-NI and the carry-less multiplication instructions on future Intel processors. In this paper we consider the case of $M_T = 1$. For VORTEX with 256-bit outputs we can directly exploit the `aesenc` operation. The key schedule calls upon the AES S-box but this can be easily emulated. For the 512-bit outputs, the underlying cipher operates on 256-bit states and, using similar techniques to those described in Section 2.1, it is straightforward to operate on this larger state. In contrast to some other algorithms, e.g. ECHO and LANE, VORTEX fits into the registers. On the other hand, it turns out that there is a bit less room to exploit AES-NI parallelism.

## 5 Implementation Results

**Table 2.** The predicted Westmere performance in cycles/Byte for those algorithms that can benefit from the Intel AES instructions set. For illustration, we provide the optimised performance figures given by submitters at the first NIST SHA-3 workshop. Other performance data can be found at [9]. Since in all cases 224- and 384-bit outputs are obtained by truncating 256- and 512-bit outputs, we only give figures for the latter.

| | *256-bit* | | *512-bit* | |
|---|---|---|---|---|
| *Algorithm* | AES-NI | previous | AES-NI | previous |
| ARIRANG | 14.9 | 14.9 | – | 11.3 |
| CHEETAH | 7.6 | 9.3 | – | 13.6 |
| ECHO (DOUBLE-PIPE) | 6.6 | 28.5 | 12.3 | 53.5 |
| ECHO (SINGLE-PIPE) | 5.7 | 24.4 | 8.1 | 35.7 |
| LANE | 5.5 | 25.7 | 13.9 | 145.0 |
| LESAMNTA | 30.8 | 52.7 | 19.9 | 51.2 |
| LUX | 6.6 | 10.2 | – | 9.5 |
| SHAVITE-3 | 5.6 | 26.7 | 5.5 | 38.2 |
| VORTEX ($M_T = 1$) | 4.4 | 46.3 | 5.2 | 56.1 |

Performance estimates for all SHA-3 candidates considered in this paper are given in Table 5. The Nehalem measurements were made on a Core i7 920 processor[6] clocked at 2.67 GHz with GNU/Linux Debian running a 2.6.26-1-amd64 kernel. The compiler was `icc for amd64, Version 11.0, Build 20081105`. As explained in Section 3, we believe that these results will be very close to the real performance of the algorithms when run on the Westmere processor. For reference, some performance figures using assembly code from OpenSSL [28] for SHA-256 and SHA-512 timed under the same methodology on the same processor are 18.6 and 12.0 cycles/Byte respectively. While our results are preliminary, we feel they are sound enough to make some general observations.

While it is tempting to group all AES/Rijndael-based SHA-3 submissions together [5], one significant point of difference between them is that some will not be able to take advantage of AES-NI. Further, there are some algorithms, e.g. CHEETAH and LUX, for which the shorter hash outputs are likely to gain from

[6] Note that to ensure stable and clean results, we disabled two features of the processor: Hyperthreading and Turbo Boost.

AES-NI while the longer hash outputs, *i.e.* 384 and 512-bit, won't. Interestingly, CHEETAH is one of the fastest AES-inspired SHA-3 submissions on the NIST reference platform. But its performance when used with AES-NI is somewhat constrained by other non-AES components and CHEETAH may be slightly less competitive than the other algorithms when using AES-NI. That said, currently optimised code for this algorithm is reasonably efficient anyway. Our results for LESAMNTA differ from those at [17] which unfortunately use a different, inappropriate replacement instruction (see Section 3.1 and Appendix B).

As would be expected, algorithms that are specifically designed around the AES round operation—ECHO, LANE, SHAVITE-3, and VORTEX—have the most to gain by appealing to AES-NI. If we consider the figures for 256-bit hash outputs then, for single-pipe variants, the throughput performance of these four algorithms is similar. However there is a much greater contrast in performance when we turn to 512-bit hash outputs, and this is due to differences in design. For instance, SHAVITE-3 for 512-bit outputs gains substantially from AES-NI since the modified round function for 512-bit outputs offers many opportunities for parallelism. This is something that is especially suited to AES-NI. On the other hand, when we move from 256- to 512-bit outputs with LANE, while the number of AES operations per byte increases in roughly the same proportion as was the case for SHAVITE-3, there is a performance impact that comes from doubling the size of the lanes in the P- and Q-PERMUTATIONS. Of course, when compared to existing optimised implementations LANE will still gain considerably when using AES-NI. But it does demonstrate how different design decisions can lead to very different performance profiles.

## 6 Conclusions

In this paper we have provided the first in-depth analysis of the likely impact of Intel's AES instructions set on SHA-3 candidates. To do this we designed a new methodology to replicate and anticipate the likely behavior of AES-NI in Westmere and we feel that this, in itself, will be of considerable interest. We have also provided the first performance estimates for those submissions that are likely to gain from AES-NI. Throughout we have tried to make a consistent and comprehensive comparison, and we have used the best currently-available information. We believe that our predictions are accurate and, in fact, may even be conservative. All the code we have developed will be public [29] and this will allow others to develop their own optimized versions and to obtain improved performance projections.

Finally this paper sheds light on what has, until now, been a somewhat hidden issue. It is clear that the new Intel AES instructions set will have a profound effect on the performance of some of the SHA-3 submissions. At the same time, this low-level support for AES will become very widespread within a few years. Certainly this is only one factor among many for the SHA-3 candidates; but it may well be one of the important ones.

## References

1. A. Atalay, O. Kara, F. Karakoc, and C. Manap. Shamata Hash Function Algorithm Specifications. Available from [26].
2. R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. SHA-3 Proposal: ECHO. Available from [26].
3. D. Bernstein. Cache-timing attacks on AES, preprint 2005. Available via `http://cr.yp.to/papers.html#cachetiming`.
4. E. Biham and O. Dunkelman. The SHAvite-3 Hash Function. Available from [26].
5. T. Bjørstad. A Short Note on AES-inspired Hashes. Posting to NIST SHA-3 mailing list, 25 May, 2009.
6. D. Chang, S. Hong, C. Kang, J. Kang, J. Kim, C. Lee, J. Lee, J. Lee, S. Lee, Y. Lee, J. Lim, and J. Sung. Arirang. Available from [26].
7. D. Coppersmith, S. Pilpel, C.H. Meyer, S.M. Matyas, M.M. Hyden, J. Oseas, B. Brachtl, and M. Schilling. Data authentication using modification detection codes based on a public one way encryption function. U.S. Patent No. 4,908,861, March 13, 1990.
8. J. Daemen and V. Rijmen. The Design of Rijndael, ISBN 3-540-42580-2, Springer.
9. ECRYPT. eBASH: ECRYPT Benchmarking of All Submitted Hashes. Available from `http://bench.cr.yp.to/ebash.html`.
10. ECRYPT. The SHA-3 Zoo: `http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo`.
11. E. Fleischmann, C. Forler, and M. Gorski. The Twister Hash Function Family. Available from [26].
12. P. Gauravaram, L. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. Thomsen. Grøstl—a SHA-3 Candidate. Available from [26].
13. S. Gueron. Intel's Advanced Encryption Standard (AES) Instructions Set. Intel Corporation White Paper, March 2009. Available at `http://software.intel.com`.
14. S. Gueron. Intel's New AES Instructions for Enhanced Performance and Security. In O. Dunkelman, editor, *Proceedings of FSE 2009*, to appear.
15. S. Halevi, W. Hall, and C. Jutla. The Hash Function Fugue. Available from [26].
16. S. Hirose, H. Kuwakado, and H. Yoshida. SHA-3 Proposal: Lesamnta. Available from [26].
17. S. Hirose, H. Kuwakado, and H. Yoshida. The Hash Function Famly Lesamnta. Available via `http://www.sdl.hitachi.co.jp/crypto/lesamnta`.
18. S. Indesteege. The LANE Hash Function. Available from [26].
19. Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, Table 2-6 of `http://www.intel.com/Assets/PDF/manual/248966.pdf`.
20. Intel Corporation. Intel Software Development Emulator (SDE). Available from `http://software.intel.com/en-us/avx/`.
21. Intel Corporation. Intel IACA tool: A Static Code Analyser. Available from `http://software.intel.com/en-us/avx/`.
22. D. Khovratovich, A. Biryukov, and I. Nikolić. The Hash Function Cheetah. Available from [26].
23. M. Kounavis and S. Gueron. Vortex: A New Family of One Way Hash Functions based on Rijndael Rounds and Carry-less Multiplication. Available from [26].
24. National Institute of Standards and Technology. FIPS 197: Advanced Encryption Standard. Available via `http://csrc.nist.gov/publications/fips/`.
25. National Institute of Standards and Technology. The SHA-3 Hash Function Competition. Available from [26].

26. National Institute of Standards and Technology. First Round Candidates of the SHA-3 Hash Function Competition.
`http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_md1.html`
27. I. Nikolić, A. Biryukov, and D. Khovratovich. Hash Family LUX. Available from [26].
28. OpenSSL 1.0.0. Available at: `http://www.openssl.org/source/`.
29. Orange Labs. Optimised implementations of SHA-3 submissions using AES-NI. Available via `http://crypto.rd.francetelecom.com/sha3/AES/`.
30. D. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: The Case of AES. In D. Pointcheval, editor, Topics in Cryptology—CT-RSA 2006, volume 3860 of *LNCS*, pages 1–20, Springer, 2006.
31. K. Varıcı, O. Özen, Ç. Kocair. Sarmal: SHA-3 Proposal. Available from [26].

## Appendix A: Instructions

**Table 3.** The instructions that provide AES encryption.

| aesenc xmm1, xmm2/m128 | aesenclast xmm1, xmm2/m128 |
|---|---|
| Tmp := xmm1; | Tmp := xmm1; |
| Round Key := xmm2/m128; | Round Key := xmm2/m128; |
| Tmp := ShiftRows (Tmp); | Tmp := ShiftRows(Tmp); |
| Tmp := SubBytes (Tmp); | Tmp := SubBytes (Tmp); |
| Tmp := MixColumns (Tmp); | xmm1 := Tmp xor Round Key |
| xmm1 := Tmp xor Round Key; | |

**Table 4.** How to derive the MixColumns operation from AES-NI

| aesdeclast xmm1, 0x0 ··· 0 |
|---|
| aesenc xmm1, 0x0 ··· 0 |
| Tmp := xmm1 |
| Tmp := InvShiftRows (Tmp); |
| Tmp := InvSubBytes (Tmp); |
| xmm1 := Tmp xor 0x0; |
| Tmp := xmm1 |
| Tmp := ShiftRows (Tmp); |
| Tmp := SubBytes (Tmp); |
| Tmp := MixColumns (Tmp); |
| xmm1 := Tmp xor 0x0; |

### Description of some additional operations used in this work

`pshufb xmm1, xmm2/m128` This instruction is used to generate a byte-wise permutation of the contents of the first 128-bit operand, where the permutation is defined by the second operand (`xmm` register or a memory location). The second source operand (`xmm2/m128`) is used as a mask, as follows. For each byte of `xmm2/m128`, the least significant four bits specify from where to select the corresponding byte of the source operand (`xmm1`). In addition, if the most significant bit of a byte of `xmm2/m128` equals one, then, regardless of the values of the other bits in that byte, zero is written in the result byte.

`pblendw  xmm1, xmm2/m128, imm8` This operation "blends" the contents of two 128-bit operands (two registers or a register and a memory location) at the granularity of 16-bit words. Words from the second operand are conditionally written to the destination operand, depending on the setting of bits in the byte operand `imm8`. If bit $k$ of this byte is set, then word $k$ of the source is copied to the destination. If bit $k$ is zero, word $k$ of the destination is unchanged.

## Appendix B: Rationale behind the Replacements

### Additional IACA traces

AES-NI provides the `aesimc` instruction to perform InvMixColumns:

```
Total Latency:   6 Cycles;    Total number of Uops:  3

| Num of |          Ports pressure in cycles           |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    3   |  2 |    |    |    |    |    |    |    |  1 | CP | aesimc xmm0, xmm1
```

The IACA tool supports the `aesdec` instruction the trace of which is shown below but does not support the `aesdeclast` instructions. From what has been derived for `aesenc`, `aesdec`, and `aesimc`, it is reasonable to assume its trace would have been identical to that of `aesdec`.

```
Total Latency:   6 Cycles;    Total number of Uops:  3

| Num of |          Ports pressure in cycles           |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    3   |  2 |    |    |    |    |    |    |    |  1 | CP | aesdec xmm0, xmm1
```

### Instructions replacement size

In order to evaluate the possible impact on the prefetching step (the prefetch buffer has a size of 16 bytes) or on the instruction cache, we conducted the following experiment: we went through the same kind of analysis as we conducted on `aesenc` and we replaced `pmulld xmm15, [mem]` which has two sequential $\mu$ops of 3 cycles on port 1 by

```
phminposuw xmm15,[mem]
phminposuw xmm15, xmm15
```

which have a single $\mu$op on port 1 each, but are interdependent. While the size of `pmulld` is 7 bytes and the size of the proposed replacement is 17 bytes, they both ran on the Nehalem with identical timings. Not only does this lend support to our approach, but it also suggests that the increased size of our AES-NI instructions set replacement is unlikely to have a significant effect.

### Instructions replacement for the memory variant

The `aesenc reg, [mem]` replacement we propose is actually quite similar to the `aesenc reg, reg` one. The only difference lies in the simulation of the memory access: it shouldn't impact the $\mu$op flows and, to accurately simulate `aesenc reg, [mem]`, the corresponding $\mu$op should start at the same cycle as the first $\mu$op on port 0. This is why we chose to launch the memory access at the first `mulps` instruction:

movdqu $\texttt{xmm}_k$, $\texttt{xmm}_i$
mulps $\texttt{xmm}_i$ , [mem]
mulps $\texttt{xmm}_k$, $\texttt{xmm}_j$
xorps $\texttt{xmm}_i$ , $\texttt{xmm}_k$

The validity of this replacement is assessed by the two following IACA traces:

```
Total Latency:  12 Cycles;    Total number of Uops:  4

| Num of |          Ports pressure in cycles          |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    4   |  2 |    |    |  1 |  1 |  X |  X |    |  1 | CP | aesenc xmm0, [0x6008f0]
```

```
Total Latency:  11 Cycles;    Total number of Uops:  5

| Num of |          Ports pressure in cycles          |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    1   |  X |    |  1 |    |    |    |    |    |  X |    | movdqu xmm2, xmm0
|    2   |  1 |    |    |  1 |  1 |  X |  X |    |    | CP | mulps  xmm0, [0x6008f0]
|    1   |  1 |    |    |    |    |    |    |    |    |    | mulps  xmm2, xmm1
|    1   |    |    |    |    |    |    |    |    |  1 | CP | xorps  xmm0, xmm2
```

An unfortunate side-effect of this replacement is that it affects an additional `xmm` register, putting additional constraints when avoiding false dependencies. This mainly concerns the ECHO and LANE algorithms.

### Equivalent inverse cipher

The equivalent inverse cipher [8] allows for a decryption structure that is very similar to that of encryption. This is achieved by noticing that the straightforward decryption algorithm

$$\mathsf{InvShiftRows}\,, \quad \mathsf{InvSubBytes}\,, \quad \mathsf{AddRoundKey}\,, \quad \mathsf{InvMixColumns}\,,$$

can be replaced by the equivalent one

$$\mathsf{InvSubBytes}\,, \quad \mathsf{InvShiftRows}\,, \quad \mathsf{InvMixColumns}\,, \quad \mathsf{AddRoundKey}\,,$$

as the two first rounds commute and the last two commute when the key expansion is tweaked accordingly; decryption is now similarly structured to encryption:

$$\mathsf{SubBytes}\,, \quad \mathsf{ShiftRows}\,, \quad \mathsf{MixColumns}\,, \quad \mathsf{AddRoundKey}\,.$$

### An inappropriate replacement

In this paragraph, we give the IACA trace for the `pmuludq` instruction. This shows that the replacement proposed in [23] is not appropriate as a generic `aesenc` replacement on the Nehalem architecture. In the trace below, `pmuludq` has a latency of 3 cycles whereas the `aesenc` instruction has a latency of 6 cycles, so the two instructions behave differently. It is even worse at the $\mu$op level, as `aesenc` has 3 $\mu$ops dispatched through ports 0 and 5 whereas `pmuludq` has a single $\mu$op dispatched on port 1: this will lead to very distinct behaviors, and almost certainly a different throughput.

```
Total Latency:   3 Cycles;    Total number of Uops:  1

| Num of |          Ports pressure in cycles          |    |
|  Uops  |  0 - DV |  1 |  2 -  D |  3 -  D |  4 |  5 |    |
------------------------------------------------------------
|    1   |    |    |  1 |    |    |    |    |    |    | CP | pmuludq xmm0, xmm1
```

This explains the differences in the performance of LESAMNTA derived in this paper and quoted at [17].