New AES Software Speed Records

Daniel J. Bernstein[1] and Peter Schwabe[2,⋆]

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7045, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, Netherlands
peter@cryptojedi.org

Abstract. This paper presents new speed records for AES software, taking advantage of (1) architecture-dependent reduction of instructions used to compute AES and (2) microarchitecture-dependent reduction of cycles used for those instructions. A wide variety of common CPU architectures—amd64, ppc32, sparcv9, and x86—are discussed in detail, along with several specific microarchitectures.

Keywords: AES, software implementation.

1 Introduction

This paper describes a new AES software implementation achieving extremely high speeds on various common CPUs. For example, on an UltraSPARC III or IV, this implementation's main loop takes only 193 cycles/block. The smallest cycle count previously claimed for AES software on *any* SPARC was 270 cycles/block by Helger Lipmaa's proprietary software.

Almost all of the specific techniques we use are well known. The main novelty in this paper lies in the analysis and combination of these techniques, producing surprisingly high speeds for AES. We have published our software to ensure verifiability of our results; we have, furthermore, placed the software into the public domain to maximize reusability of our results.

Section 2 reviews the standard structure of "32-bit" AES software implementations. Section 3 surveys techniques for reducing the number of CPU *instructions* required to compute AES. Section 4 explains how we reduced the number of CPU *cycles* required to compute AES on various platforms.

Thanks to Ruben Niederhagen and the anonymous reviewers for suggesting many improvements in our explanations.

⋆ The first author was supported by the National Science Foundation under grant ITR–0716498. He carried out parts of this work while visiting Technische Universiteit Eindhoven. The second author was supported by the European Commission through the ICT Programme under Contract ICT–2007–216499 CACE. Permanent ID of this document: b90c51d2f7eef86b78068511135a231f. Date: 2008.09.25.

D.R. Chowdhury, V. Rijmen, and A. Das (Eds.): INDOCRYPT 2008, LNCS 5365, pp. 322–336, 2008.

Which AES? This paper focuses on the most common AES key size, namely 16 bytes (128 bits). We plan to adapt our implementation later to support 32-byte (256-bit) keys. One might guess that AES is 40% slower with 32-byte keys, since 16-byte keys use 10 rounds while 32-byte keys use 14 rounds; actual costs are not exactly linear in the number of rounds, but 40% is a reasonable estimate.

There are several modes of operation of AES: cipher-block chaining (CBC), output feedback (OFB), counter mode (CTR), et al. There are also, in the literature, many different ways to benchmark AES software. This variability interferes with comparisons. Often a faster AES performance report is from a *slower* AES implementation measured with a less invasive benchmarking framework.

A big improvement in comparability has been achieved in the last few years by eSTREAM, a multi-year ECRYPT project that has identified several promising new stream ciphers. The eSTREAM benchmarking framework has been made public, allowing anyone to verify performance data; includes long-message and short-message benchmarks; and includes AES-CTR as a basis for comparison. The original AES-CTR implementation in the benchmarking framework is a reference implementation written by Brian Gladman; other authors have contributed implementations optimized for several architectures.

This paper reports cycle counts directly from the eSTREAM benchmarking framework, and uses exactly the same form of AES-CTR. Our software passes the extensive AES-CTR tests included in the benchmarking framework. By similar techniques we have also sped up Biryukov's LEX stream cipher [6].

See [10] for much more information on the eSTREAM project; [9] for the benchmarking framework; [4] for a more portable version of the benchmarking framework (including our software as of version 20080905); and [12] for more information about Gladman's AES software.

Bitslicing. The recent papers [23], [17], and [19] have proposed bitsliced AES implementations for various CPUs. The most impressive report, from Matsui and Nakajima in [19], is 9.2 cycles/byte for bitsliced AES on a Core 2.

Unfortunately, this speed is achieved only for 2048-byte chunks that have been "transposed" into bitsliced form. Transposition of ciphertext costs about 1 cycle/byte. More importantly, bitsliced encryption of a 576-byte Internet packet costs as much as bitsliced encryption of a 2048-byte packet, multiplying the cycle counts by approximately 3.5. Consequently this bitsliced implementation is not competitive in speed with the implementation reported in this paper. The very recent semi-bitsliced implementation in [14] uses much smaller chunks, only 64 bytes, but it is also non-competitive: it takes 19.81 cycles/byte on an Athlon 64.

Bitslicing remains of interest for several reasons: first, some applications encrypt long streams and do not mind padding to 2048-byte boundaries; second, some applications will use bitslicing on both client and server and can thus eliminate the costs of transposition; third, bitsliced implementations are inherently immune to the cache-timing attacks discussed in [3] and [21].

Other literature. Worley et al. in [26] report AES implementations for PA-RISC and IA-64. Schneier and Whiting in [24] report AES implementations for the Pentium, Pentium Pro, HP PA-8200, and IA-64. Weiss and Binkert in

[25] report AES implementations for the Alpha 21264. Aoki and Lipmaa in [1] report AES implementations for the Pentium II. Most of these CPUs are now quite difficult to find, but these papers—particularly [1]—are well worth reading for their discussions of AES optimizations.

More recent AES speed reports: Osvik in [20] covers Pentium III, Pentium 4, and Athlon. Lipmaa in [16] and [15] covers many CPUs. Matsui and Fukuda in [18] cover the Pentium III and Pentium 4. Matsui in [17] covers the Athlon 64.

[5], [2], and [8] report implementations for smaller CPUs using the ARM architecture. [13] reports implementations for graphics processors (GPUs). There is also an extensive literature on implementations of AES in hardware, in FPGAs, and in hardware-software codesigns.

2 A Short Review of AES

AES expands its 16-byte key into 11 "round keys" $r_0, \ldots, r_{10}$, each 16 bytes. Each 16-byte block of plaintext is xor'ed with round key r_0, transformed, xor'ed with round key r_1 (ending "round 1"), transformed, xor'ed with round key r_2 (ending "round 2"), transformed, etc., and finally xor'ed with round key r_{10} (ending "round 10") to produce a 16-byte block of ciphertext.

Appendix A is sample code in the C programming language for a typical round of AES. The round reads a 16-byte state stored in four 4-byte variables `y0`, `y1`, `y2`, `y3`; transforms the variables; xor's a 16-byte round key stored in four 4-byte variables; and puts the result into `z0`, `z1`, `z2`, `z3`.

The main work in the transformation is 16 table lookups indexed by the 16 bytes of `y0`, `y1`, `y2`, `y3`; beware that the last round of AES is slightly different. Each table lookup produces 4 bytes. In this sample code there are four tables interleaved in memory, with the jth entry of table i at address `table` $+ 4i + 16j$; `table` $+ 4i$ is precomputed as a byte pointer `table`i. For example,

```
p03 = (uint32) y0 << 4;
p03 &= 0xff0;
p03 = *(uint32 *) (table3 + p03);
```

in the sample code extracts the bottom byte of `y0`, multiplies it by 16, adds it to the byte pointer `table3`, and reads the 4 bytes at that address.

One can eliminate the table interleaving, and store the jth entry of table i at address `table` $+ 1024i + 4j$; then the shift distances `20`, `12`, `4`, `4` need to be changed to `22`, `14`, `6`, `2`. An intermediate possibility is to interleave the first two tables and interleave the second two tables, using shift distances `21`, `13`, `5`, `3`.

We do not describe the work required to invert this process, computing a 16-byte plaintext from a 16-byte ciphertext and a 16-byte key. In AES-CTR, the "plaintext" is actually a 16-byte counter; the encrypted counter serves as keystream that is xor'ed to the user's actual plaintext, producing counter-mode ciphertext. AES-CTR decryption is the same as AES-CTR encryption so it does not require inverting AES.

Relationship to `SubBytes` etc. An AES round is often described differently, as a series of four operations on 4×4 matrices: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`. The last round skips `MixColumns`.

As part of the original AES proposal, Daemen and Rijmen described how to merge `SubBytes`, `ShiftRows`, and `MixColumns` into 16 lookups from 4 tables T_0, T_1, T_2, T_3, each containing 256 32-bit entries; `AddRoundKey` is nothing but xor'ing the round key r_i. See [7, Section 5.2]. As far as we know, the first implementation using this structure was written by Rijmen, Bosselaers, and Barreto. This is the structure used in the sample code in Appendix A, and the starting structure for the speedups described in the following sections.

3 Saving Instructions for AES

This section surveys several methods to reduce the number of CPU integer instructions, load instructions, etc. used for AES. Many of these methods take advantage of additional instructions and features provided by some CPUs.

Beware that not all of the methods can be combined. Furthermore, minimizing *cycles* is a much more subtle task than minimizing *instructions*. In Section 4 we discuss the cycle counts that we have achieved on various platforms, taking account of limited register sets, instruction latencies, etc.

3.1 Baseline (720 Instructions)

One round of AES can be decomposed into 16 shift instructions, 16 mask instructions, 16 load instructions for the table lookups, 4 load instructions for the round keys, and 16 xor instructions. See Appendix A. Overall there are 68 instructions, specifically 20 loads and 48 integer instructions.

Subsequent code examples in this section express CPU instructions using the `qhasm` language, `http://cr.yp.to/qhasm.html`. Each line that we display represents one CPU instruction.

All of the target platforms have shift instructions, mask instructions, load instructions, and xor instructions. Some platforms—for example, the x86 and amd64—do not support three-operand shift instructions (i.e., shift instructions where the output register is not the input register) but do have byte-extraction instructions that are adequate to achieve the same instruction counts. In this section we ignore register-allocation issues.

The 10-round main loop uses more than 680 instructions, for four reasons:

- Before the first round there are 4 extra round-key loads and 4 extra xors.
- The last round has 16 extra masks, one after each of the table lookups.
- For AES-CTR there are 4 loads of 4-byte plaintext words, 4 xors of keystream with plaintext, and 4 stores of ciphertext words.
- There are 4 extra instructions for miscellaneous tasks such as incrementing the AES-CTR input.

Overall there are 720 instructions, specifically 208 loads, 4 stores, and 508 integer instructions. The 508 integer instructions consist of 160 shift instructions, 176 mask instructions, 168 xor instructions, and 4 extra instructions.

In this count we ignore the costs of conditional branches; these costs are easily reduced by unrolling. We also ignore extra instructions needed to handle, e.g., big-endian loads on a little-endian architecture; almost all endianness issues can be eliminated by appropriate swapping of the AES code and tables.

We also ignore the initial costs of computing the 176 bytes of round keys from a 16-byte key. This computation involves hundreds of extra instructions—certainly a noticeable cost—but the round keys can be reused for all the blocks of a message. Round keys can also be reused for other messages if they are saved.

3.2 Table Structure and Index Extraction

Combined shift-and-mask instructions (−160 instructions). Some architectures allow a shift instruction `p02=y0>>4` and a mask instruction `p02&=0xff0` to be combined into a single instruction `p02=(y0>>4)&0xff0`. Replacing 160 shifts and 160 masks by 160 shift-and-mask instructions saves 160 instructions.

On the ppc32 architecture, for example, the `rlwinm` instruction can do any rotate-and-mask where the mask consists of consecutive bits.

Scaled-index loads (−80 instructions). On other architectures a shift instruction `p03<<=4` and a load instruction `p03=*(uint32*)(table3+p03)` can be combined into a single instruction. The instructions

```
p03 = (uint32) y0 << 4
p03 &= 0xff0
p03 = *(uint32 *) (table3 + p03)
```

for handling the bottom byte of `y0` can then be replaced by

```
p03 = y0 & 0xff
p03 = *(uint32 *) (table3 + (p03 << 4))
```

Similarly, the instructions

```
p00 = (uint32) y0 >> 20
p00 &= 0xff0
p00 = *(uint32 *) (table0 + p00)
```

for handling the top byte of `y0` can be replaced by

```
p00 = (uint32) y0 >> 24
p00 = *(uint32 *) (table0 + (p00 << 4))
```

In 10 rounds there are 40 top bytes and 40 bottom bytes.

The x86 architecture, for example, allows scaled indices in load instructions. The x86 scaling allows only a 3-bit shift, not a 4-bit shift, but this is easily accommodated by non-interleaved (or partially interleaved) tables.

Second-byte instructions (−40 instructions). All architectures support a mask instruction `p03=y0&0xff` that extracts the bottom byte of `y0`.

Some architectures—for example, the x86—also support a single instruction `p02=(y0>>8)&0xff` to extract the *second* byte of `y0`. In conjunction with scaled-index loads this instruction allows

```
p02 = (uint32) y0 >> 6
p02 &= 0x3fc
p02 = *(uint32 *) (table2 + p02)
```

to be replaced by

```
p02 = (y0 >> 8) & 0xff
p02 = *(uint32 *) (table2 + (p02 << 2))
```

saving another 40 instructions overall.

Padded registers (−80 instructions). Some architectures—e.g., sparcv9—do not have any of the combined instructions described above, but *do* have 64-bit registers. On these architectures one can expand a 4-byte value such as `0xc66363a5` into an 8-byte value such as `0x0c600630063a50` (or various other possibilities such as `0x0000c0630630a50`). If this expansion is applied consistently in the registers, the lookup tables (before the last round), and the round keys, then it does not cost any extra instructions.

The advantage of the padded 8-byte value `0x0c600630063a50` is that a single mask instruction produces the shifted bottom byte `a50`, and a single shift instruction produces the shifted top byte `c60`. Consequently the original eight shift-and-mask instructions, for extracting four shifted bytes from `y0`, can be replaced by six instructions:

```
p00 = (uint64) y0 >> 48
p01 = (uint64) y0 >> 32
p02 = (uint64) y0 >> 16
p01 &= 0xff0
p02 &= 0xff0
p03 = y0 & 0xff0
```

Expanded lookup tables, like scaled-index loads, thus save 80 instructions overall.

32-bit shifts of padded registers (−40 instructions). Some architectures—for example, sparcv9—have not only a 64-bit right-shift instruction but also a 32-bit right-shift instruction that automatically masks its 64-bit input with `0xffffffff`. This instruction, in conjunction with padded registers, allows

```
p02 = (uint64) y0 >> 16
p02 &= 0xff0
```

to be replaced by `p02 = (uint32) y0 >> 16`, saving 40 additional instructions.

3.3 Speedups for the Last Round

Byte loads (−4 instructions). As mentioned earlier, the last round of AES has 16 extra masks for its 16 table lookups. Four of the masks are `0xff`. All of

the target architectures allow these masks to be absorbed into single-byte load instructions. For example,

```
p00 = *(uint32 *) (table0 + p00)
p00 &= 0xff
```

can be replaced with `p00 = *(uint8*) (table0 + p00)` on little-endian CPUs or `p00 = *(uint8*) (table0 + p00 + 3)` on big-endian CPUs.

Two-byte loads (−4 instructions). Four of the masks are `0xff00`. These masks can be absorbed into two-byte load instructions if the table structure has `00` next to the desired byte. Often this structure occurs naturally as part of other table optimizations, and in any case it can be achieved by a separate table.

Masked tables (−8 instructions). The other eight masks are `0xff0000` and `0xff000000`. These masked values cannot be produced by byte loads and two-byte loads but can be produced by four-byte loads from separate tables whose entries are already masked.

Separate masked tables are also the easiest way to handle the distinction between padded 64-bit registers (see Section 3.2) and packed 32-bit AES output words.

Combined mask and insert (−16 instructions). A 4-byte result of the last round, such as `z0`, is produced by 4 xors with 4 masked table entries, where the masks are `0xff`, `0xff00`, `0xff0000`, `0xff000000`.

Some architectures have an instruction that replaces specified bits of one register with the corresponding bits of another register. For example, the ppc32 architecture has a `rlwimi` instruction that does this, optionally rotating the second register. The instruction sequence

```
p00 &= 0xff000000
p11 &=   0xff0000
p22 &=     0xff00
p33 &=       0xff
z0 ^= p00
z0 ^= p11
z0 ^= p22
z0 ^= p33
```

can then be replaced by

```
p00 bits 0xff0000 = p11 <<< 0
p00 bits   0xff00 = p22 <<< 0
p00 bits     0xff = p33 <<< 0
z0 ^= p00
```

(Note for C programmers: in C notation, `p00 bits 0xff0000 = p11` would be `p00 = (p00&0xff00ffff)|(p11&0xff0000)`.) This is another way—without using byte loads, and without constraining the table structure—to eliminate all the extra masks.

3.4 Further Speedups

Combined load-xor (−168 instructions). Often the result of a load is used solely for xor'ing into another register. Some architectures—for example, x86 and amd64—allow load and xor to be combined into a single instruction.

Byte extraction via loads (−160 . . . −320 integer instructions; +200 load/store instructions). Extracting four indices from `y0` takes at most 8 integer instructions, and on some architectures as few as 4 integer instructions, as discussed in Section 3.2.

A completely different way to extract four bytes from `y0`—and therefore to extract indices, on architectures allowing scaled-index loads—is to store `y0` and then do four byte loads from the stored bytes of `y0`. This eliminates between 4 and 8 integer instructions—potentially helpful on CPUs where integer instructions are the main bottleneck—at the expense of 5 load/store instructions.

One can apply this conversion to all 160 byte extractions. One can also apply it to some of the byte extractions, changing the balance between load instructions and integer instructions. The optimum combination is CPU-dependent.

Round-key recomputation (−30 load instructions; +30 integer instructions). In the opposite direction: Instead of loading 44 round-key words, say words $0, 1, 2, \ldots, 43$, one can load 14 round-key words, specifically words $0, 1, 2, 3, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40$, and compute the other round-key words by 30 xors, taking advantage of the AES round-key structure. This reduces the number of load instructions—potentially helpful on CPUs where loads are the main bottleneck—although it increases the number of integer instructions.

One can also use intermediate combinations, such as 22 loads and 22 xors. As before, the optimum combination is CPU-dependent.

Round-key caching (≈ −44 instructions). Each 16-byte block of input involves 44 round-key loads (or xors). The same round keys are used in the next block. On an architecture with many registers, some or all of the round keys can be kept in registers, moving these loads out of the main loop. The same type of savings appears if several blocks are handled in parallel.

Counter-mode caching (≈ −100 instructions). Recall that our AES software uses counter mode in exactly the form specified by eSTREAM. AES is applied to a 16-byte counter that is increased by 1 for every block of input.

Observe that 15 bytes of the counter remain constant for 256 blocks; just one byte of the counter changes for every block of input. All operations in the first round not depending on this byte are shared between 256 blocks. The resulting values `y0`, `y1`, `y2` and `y3` can be saved and reused in 256 consecutive blocks.

Similar observations hold for round 2: only one of the four 4-byte input words of round 2 changes every block. All computations not depending on this word can be saved and reused in 256 consecutive blocks.

This caching is perhaps the least well known of all AES software speedups. We learned it from an eSTREAM AES implementation by Hongjun Wu; we have not found it elsewhere in the literature.

4 Saving Cycles for AES

Minimizing instructions is not the same as minimizing cycles. A CPU that advertises "four instructions per cycle" actually performs *at most* four instructions per cycle; software that does not take account of microarchitecture-specific bottlenecks often runs much more slowly, sometimes below one instruction per cycle.

For example, a typical microarchitecture does not allow the results of an integer instruction to be used until the next cycle; multiple instructions in the same cycle must therefore be independent parallel operations. The results of a load instruction cannot be used until two or three cycles later, requiring even more independent instructions to be carried out in parallel. On "in-order" CPUs, parallel instructions need to write to different output registers; instruction scheduling is often heavily constrained by limits on the number of architectural registers. On "out-of-order" CPUs, controlling the precise scheduling of instructions can be extremely difficult.

This section reports the cycle counts that we have achieved on several specific CPUs. For each CPU we describe the important bottlenecks and the measures that we took to address those bottlenecks. We describe the CPUs in decreasing order of our cycle counts.

We report the speeds of our implementation and previous AES implementations measured by the eSTREAM benchmarking framework. The framework focuses on long-stream performance, but it also measures short-packet performance for the fastest software; our tables include the results for 576-byte packets.

4.1 Motorola PowerPC G4 7410, ppc32 Architecture

We measured our software on a computer named `gggg` in the Center for Research and Instruction in Technologies for Electronic Security (RITES) at the University of Illinois at Chicago. This computer has two 533MHz Motorola PowerPC G4 7410 processors; measurements used one processor. Resulting speeds for encrypting a long stream:

Software	Measurement	Cycles/byte
This paper	eSTREAM	14.57 (or 15.34 for 576 bytes)
Wu	eSTREAM	16.26
Bernstein	eSTREAM	17.84
Gladman	eSTREAM	26.74
OpenSSL 0.9.8c	`openssl speed aes`	29

Unpublished code by Denis Ahrens is claimed in [16] to use 25.06 cycles/byte on a PowerPC G4 7400, a very similar CPU to a PowerPC G4 7410, and 24.06 cycles/byte on a PowerPC G4 7457, a somewhat more powerful CPU.

Reducing instructions. For this CPU, our implementation uses 461 instructions in the main loop, specifically 180 load/store instructions, 279 integer instructions, and 2 branch instructions. We use the following techniques from

Section 3: combined shift-and-mask instructions; combined mask-and-insert; and counter-mode caching.

Reducing cycles. The PowerPC G4 7410 can dispatch at most 3 instructions per cycle. At most 2 of the instructions can be load/store or integer instructions, so our 459 non-branch instructions take at least $459/2 = 229.5$ cycles, i.e., 14.34 cycles/byte. At most 1 of the instructions can be a load/store instruction, but we have only 180 load/store instructions, so this is a less important bottleneck.

The G4 is, for most purposes, an in-order CPU, so load instructions have to be interleaved with arithmetic instructions. Results of load instructions are available after 3 cycles. Saving all possible callee-save registers makes 29 4-byte integer registers available for AES encryption. Detailed analysis shows that these are enough registers for almost perfect instruction scheduling, making the processor execute 2 instructions almost every cycle in the AES main loop. Our 233 cycles/loop are very close to the 229.5 cycles/loop lower bound for 459 instructions. We do not mean to suggest that 29 registers are ample; further registers would be useful for round-key caching.

4.2 Intel Pentium 4 f12, x86 Architecture

Warning: There are considerable performance differences between, e.g., a Pentium 4 f12, a Pentium 4 f29, a Pentium 4 f41, etc. "Pentium 4" is not an adequate CPU specification for performance measurements.

We measured our AES software on a computer named `fireball` in the Center for Research and Instruction in Technologies for Electronic Security (RITES) at the University of Illinois at Chicago. This computer has a single-core 1900MHz Intel Pentium 4 f12 processor. Resulting speeds for encrypting a long stream:

Software	Measurement	Cycles/byte
This paper	eSTREAM	14.13 (or 14.54 for 576 bytes)
Bernstein	eSTREAM	16.97
Wu	eSTREAM	18.23
OpenSSL 0.9.8g	`openssl speed aes`	21
Gladman	eSTREAM	26.48

Unpublished code by Matsui and Fukuda is claimed in [18] to use 15.69 cycles/byte on a "Pentium 4 Northwood," i.e., a Pentium 4 f2, and 17.75 cycles/byte on a "Pentium 4 Prescott," i.e., a Pentium 4 f3/f4. Unpublished code by Osvik is claimed in [20] to use 16.25 cycles/byte on an unspecified type of Pentium 4. Unpublished code by Lipmaa is claimed in [16] to use 15.88 cycles/byte on an unspecified type of Pentium 4. We have seen several other reports of Pentium 4 AES speeds above 20 cycles/byte.

Reducing instructions. For this CPU, our implementation uses 414 instructions in the main loop. We use the following techniques from Section 3: scaled-index loads; second-byte instructions; byte loads; two-byte loads; masked tables; combined load-xor; and counter-mode caching. We use some extra stores and

loads to handle the extremely limited number of general-purpose x86 integer registers. We compressed our total table size (including masked tables for the last round) to 4096 bytes; this improvement does not affect the eSTREAM benchmark results but reduces cache-miss costs in many applications.

Reducing cycles. There are several tricky performance bottlenecks on the Pentium 4. We recommend the manuals by Agner Fog [11] for much more comprehensive discussions of several x86 (and amd64) microarchitectures.

The most obvious bottleneck is that the Pentium 4 can do only one load per cycle. Our main loop has 177 loads, accounting for most—although certainly not all—of the 226 cycles that we actually use.

4.3 Sun UltraSPARC III, Sparcv9 Architecture

We measured our AES software on a computer named `icarus` at the University of Illinois at Chicago. This computer has eight 900MHz Sun UltraSPARC III CPUs; measurements used one CPU. Resulting speeds:

Software	Measurement	Cycles/byte
This paper	eSTREAM	12.06 (or 12.36 for 576 bytes)
Bernstein	eSTREAM	20.75
Gladman	eSTREAM	24.08
Wu	eSTREAM	28.88
OpenSSL 0.9.7e	`openssl speed aes`	35

We also measured our AES software on a computer named `nmisolaris10` in the NMI Build and Test Lab at the University of Wisconsin at Madison. This computer has two 1200MHz Sun UltraSPARC III Cu processors; measurements used one processor.

Software	Measurement	Cycles/byte
This paper	eSTREAM	12.03 (or 12.33 for 576 bytes)
Wu	eSTREAM	17.27
Bernstein	eSTREAM	25.08
Gladman	eSTREAM	25.08

Unpublished code by Lipmaa is claimed in [16] to use 16.875 cycles/byte on a "480 MHz SPARC," presumably an UltraSPARC II. Lipmaa discusses counter mode in [15] but does not report any speedups for the SPARC in this mode.

Reducing instructions. For this CPU, our implementation uses 505 instructions in the main loop, specifically 178 load/store instructions, 325 integer instructions, and 2 branch instructions. We use the following techniques from Section 3: padded registers; 32-bit shifts of padded registers; masked tables; and counter-mode caching.

Reducing cycles. An UltraSPARC CPU dispatches at most four instructions per cycle. Only one of these instructions can be a load/store instruction, so our

178 load/store instructions use at least 178 cycles. Furthermore, only two of these instructions can be integer instructions, so our 325 integer instructions use at least 162.5 cycles.

The simplest way to mask a byte is with an arithmetic instruction: for example, `&0xff00`. The SPARC architecture supports only 12-bit immediate masks, so three of the masks have to be kept in registers.

The UltraSPARC is an in-order CPU, except for store instructions. Proper instruction scheduling thus requires each load instruction to be grouped with two integer instructions. Only 24 8-byte integer registers are available, posing some challenges for instruction scheduling. We have built a simplified UltraSPARC simulator that accounts for 186 cycles with our current instruction scheduling; we are continuing to analyze the gaps between 178 cycles, 186 cycles, and the 193 cycles actually used by our main loop.

We have considered round-key recomputation (see Section 3) to trade some loads for integer instructions, but this makes scheduling even more difficult. With more registers we would expect to be able to reach approximately 170 cycles.

4.4 Intel Core 2 Quad Q6600 6fb, amd64 Architecture

We measured our AES software on a computer named `latour` in the Coding and Cryptography Computer Cluster (C4) at Technische Universiteit Eindhoven. This computer has a 2400MHz Intel Core 2 CPU with four cores; measurements used one core. Resulting speeds for encrypting a long stream:

Software	Measurement	Cycles/byte
This paper	eSTREAM	10.57 (or 10.79 for 576 bytes)
Wu	eSTREAM	12.27
Bernstein	eSTREAM	13.75
Gladman	eSTREAM	16.17
OpenSSL 0.9.8g	`openssl speed aes`	18

Unpublished code by Matsui and Nakajima is claimed in [19, Table 6] to use 14.5 cycles/byte (without bitslicing) on a Core 2. See also the discussion of bitslicing in Section 1.

Reducing instructions. For this CPU, our implementation uses 434 instructions in the main loop. We use the following techniques from Section 3: scaled-index loads; second-byte instructions; byte loads; two-byte loads; masked tables; combined load-xor; round-key recomputation; round-key caching; and counter-mode caching.

Reducing cycles. The Core 2 can dispatch three integer instructions per cycle but, like the Pentium 4, can dispatch only one load per cycle. We have often spent extra integer instructions to avoid loads and to improve the scheduling of loads. For example, we have kept round-key words in "XMM" registers, even though copying an XMM register to a normal integer register costs an extra integer instruction. Our main loop currently has 143 loads, accounting for most of our 169 cycles.

4.5 AMD Athlon 64 X2 3800+ 15/75/2, amd64 Architecture

We measured our AES software on a computer named `mace` in the Center for Research and Instruction in Technologies for Electronic Security (RITES) at the University of Illinois at Chicago. This computer has one 2000MHz AMD Athlon 64 X2 3800+ 15/75/2 CPU with two cores; measurements used one core. Resulting speeds for encrypting a long stream:

Software	Measurement	Cycles/byte
This paper	eSTREAM	10.43 (or 10.71 for 576 bytes)
Wu	eSTREAM	13.32
Bernstein	eSTREAM	13.40
Gladman	eSTREAM	18.06
OpenSSL 0.9.8g	`openssl speed aes`	21

Unpublished code by Matsui is claimed in [17] to use 10.62 cycles/byte on an Athlon 64. Unpublished code by Lipmaa is claimed in [16] to use 12.44 cycles/byte on an Athlon 64.

Reducing instructions. For this CPU, our implementation uses 409 instructions in the main loop. We use the following techniques from Section 3: scaled-index loads; second-byte instructions; byte loads; two-byte loads; masked tables; combined load-xor; and counter-mode caching.

Reducing cycles. The Athlon 64 can dispatch three instructions per cycle, including *two* load instructions. Loads and stores must be carried out in program order, placing a high priority on careful instruction scheduling.

Our Athlon-64-tuned software runs at 11.54 cycles/byte on the Core 2, and our Core-2-tuned software runs at 14.77 cycles/byte on the Athlon 64, illustrating the importance of microarchitectural differences.

References

1. Aoki, K., Lipmaa, H.: Fast implementations of AES candidates. In: AES Candidate Conference, pp. 106–120 (2000)
2. Atasu, K., Breveglieri, L., Macchetti, M.: Efficient AES implementations for ARM based platforms. In: SAC 2004: Proceedings of the 2004 ACM symposium on Applied computing, pp. 841–845. ACM, New York (2004), `http://doi.acm.org/10.1145/967900.968073`
3. Bernstein, D.J.: Cache-timing attacks on AES (2005), `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`
4. Bernstein, D.J.: Estreambench software package (2008), `http://cr.yp.to/streamciphers/timings.html#toolkit-estreambench`
5. Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M., Marchesin, S.: Efficient software implementation of AES on 32-bit platforms. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 159–171. Springer, Heidelberg (2003)

6. Biryukov, A.: A new 128 bit key stream cipher: Lex (2005), http://www.ecrypt.eu.org/stream/papers.html
7. Daemen, J., Rijmen, V.: AES proposal: Rijndael (1999), http://www.iaik.tugraz.at/Research/krypto/AES/old/~rijmen/rijndael/rijndaeldocV2.zip
8. Darnall, M., Kuhlman, D.: AES software implementations on ARM7TDMI. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 424–435. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11941378_30
9. De Cannière, C.: The eSTREAM project: software performance (2008), http://www.ecrypt.eu.org/stream/perf
10. ECRYPT. The eSTREAM project (2008), http://www.ecrypt.eu.org/stream
11. Fog, A.: How to optimize for the Pentium family of microprocessors (2008), http://www.agner.org/assem/
12. Gladman, B.: AES and combined encryption/authentication modes (2006), http://fp.gladman.plus.com/AES/
13. Harrison, O., Waldron, J.: AES encryption implementation and analysis on commodity graphics processing units. In: Paillier and Verbauwhede [22], pp. 209–226
14. Könighofer, R.: A fast and cache-timing resistant implementation of the AES. In: Malkin, T.G. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 187–202. Springer, Heidelberg (2008)
15. Lipmaa, H.: AES ciphers: speed in no-feedback mode (2006), http://www.adastral.ucl.ac.uk/~helger/research/aes/nfb.html
16. Lipmaa, H.: AES/Rijndael: speed (2006), http://www.adastral.ucl.ac.uk/~helger/research/aes/rijndael.html
17. 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), http://www.iacr.org/archive/fse2006/40470344/40470344.pdf
18. 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)
19. Matsui, M., Nakajima, J.: On the power of bitslice implementation on Intel Core2 processor. In: Paillier and Verbauwhede [22], pp. 121–134, http://dx.doi.org/10.1007/978-3-540-74735-2_9
20. Osvik, D.A.: Fast assembler implementations of the AES (2003), http://www.ii.uib.no/~osvik/pres/crypto2003.html
21. 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), http://dx.doi.org/10.1007/11605805_1
22. Paillier, P., Verbauwhede, I. (eds.): CHES 2007. LNCS, vol. 4727. Springer, Heidelberg (2007)
23. Rebeiro, C., Selvakumar, A.D., Devi, A.S.L.: Bitslice implementation of AES. In: Pointcheval, D., Mu, Y., Chen, K. (eds.) CANS 2006. LNCS, vol. 4301, pp. 203–212. Springer, Heidelberg (2006), http://dx.doi.org/10.1007/11935070_14
24. Schneier, B., Whiting, D.: A performance comparison of the five AES finalists. In: AES Candidate Conference, pp. 123–135 (2000)
25. Weiss, R., Binkert, N.L.: A comparison of AES candidates on the Alpha 21264. In: AES Candidate Conference, pp. 75–81 (2000)
26. Worley, J., Worley, B., Christian, T., Worley, C.: AES finalists on PA-RISC and IA-64: implementations & performance. In: AES Candidate Conference, pp. 57–74 (2000)

A Review: One AES Round, in C

```
z0 = roundkeys[i * 4 + 0];
z1 = roundkeys[i * 4 + 1];
z2 = roundkeys[i * 4 + 2];
z3 = roundkeys[i * 4 + 3];

p00 = (uint32) y0 >> 20;
p01 = (uint32) y0 >> 12;
p02 = (uint32) y0 >> 4;
p03 = (uint32) y0 << 4;
p00 &= 0xff0;
p01 &= 0xff0;
p02 &= 0xff0;
p03 &= 0xff0;
p00 = *(uint32 *) (table0 + p00);
p01 = *(uint32 *) (table1 + p01);
p02 = *(uint32 *) (table2 + p02);
p03 = *(uint32 *) (table3 + p03);
z0 ^= p00;
z3 ^= p01;
z2 ^= p02;
z1 ^= p03;

p10 = (uint32) y1 >> 20;
p11 = (uint32) y1 >> 12;
p12 = (uint32) y1 >> 4;
p13 = (uint32) y1 << 4;
p10 &= 0xff0;
p11 &= 0xff0;
p12 &= 0xff0;
p13 &= 0xff0;
p10 = *(uint32 *) (table0 + p10);
p11 = *(uint32 *) (table1 + p11);
p12 = *(uint32 *) (table2 + p12);
p13 = *(uint32 *) (table3 + p13);
z1 ^= p10;
z0 ^= p11;
z3 ^= p12;
z2 ^= p13;

p20 = (uint32) y2 >> 20;
p21 = (uint32) y2 >> 12;
p22 = (uint32) y2 >> 4;
p23 = (uint32) y2 << 4;
p20 &= 0xff0;
p21 &= 0xff0;
p22 &= 0xff0;
p23 &= 0xff0;
p20 = *(uint32 *) (table0 + p20);
p21 = *(uint32 *) (table1 + p21);
p22 = *(uint32 *) (table2 + p22);
p23 = *(uint32 *) (table3 + p23);
z2 ^= p20;
z1 ^= p21;
z0 ^= p22;
z3 ^= p23;

p30 = (uint32) y2 >> 20;
p31 = (uint32) y2 >> 12;
p32 = (uint32) y2 >> 4;
p33 = (uint32) y2 << 4;
p30 &= 0xff0;
p31 &= 0xff0;
p32 &= 0xff0;
p33 &= 0xff0;
p30 = *(uint32 *) (table0 + p30);
p31 = *(uint32 *) (table1 + p31);
p32 = *(uint32 *) (table2 + p32);
p33 = *(uint32 *) (table3 + p33);
z3 ^= p30;
z2 ^= p31;
z1 ^= p32;
z0 ^= p33;
```