How Far Can We Go on the x64 Processors?

Mitsuru Matsui[1]

Information Technology R&D Center
Mitsubishi Electric Corporation
5-1-1 Ofuna Kamakura Kanagawa, Japan
Matsui.Mitsuru@ab.MitsubishiElectric.co.jp

Abstract. This paper studies the state-of-the-art software optimization methodology for symmetric cryptographic primitives on the new 64-bit x64 processors, AMD Athlon64 (AMD64) and Intel Pentium 4 (EM64T). We fully utilize newly introduced 64-bit registers and instructions for extracting maximal performance of target primitives. Our program of AES with 128-bit key runs in 170 cycles/block on Athlon 64, which is, as far as we know, the fastest implementation of AES on a PC processor.
Also we implemented a "bitsliced" AES and Camellia for the first time, both of which achieved very good performance. A bitslice implementation is important from the viewpoint of a countermeasure against cache timing attacks because it does not require lookup tables with a key-dependent address. We also analyze performance of SHA256/512 and Whirlpool hash functions and show that SHA512 can run faster than SHA256 on Athlon 64. This paper exhibits an undocumented fact that 64-bit *right* shifts and 64-bit rotations are extremely slow on Pentium 4, which often leads to serious and unavoidable performance penalties in programming encryption primitives on this processor.

Keywords: *Fast Software Encryption, x64 Processors, Bitslice*

1 Introduction

This paper explores instruction-level software optimization techniques for the new 64-bit x64 architecture on Athlon 64 and Pentium 4 processors. Of course a 64-bit programming is not a new topic; Alpha, PA-RISC, Sparc, etc. have been already studied in many literatures, but the new x64 architecture is extremely important and promising in the sense that it is a superset of the currently dominant x86 architecture and Microsoft finally launched 64-bit Windows running on the x64 processors in the PC market.

Interestingly and ironically, the x64 architecture was initially designed and published by AMD under the name AMD64, and later followed by Intel under the name EM64T. EM64T is binary compatible with AMD64, but the internal hardware design of Pentium 4 is completely different from that of Athlon 64. Intel pursues higher clock frequency with a deep pipeline and AMD seeks for higher superscalability and lower memory latency with an on-chip I/O controller (HyperTransport). Which of Intel and AMD is fast has been always a controversial issue.

From programmer's side, the x64 structure is what was long awaited; one of its biggest impacts is liberation from the "register starvation", which was x86 programmer's nightmare. This paradigm shift could be compared with the liberation from the "segment wall" of a 16-bit program in 1980's.

Our interest in the x64 architecture is two-fold. First we would like to see to what extent performance gain is expected on this architecture for symmetric cryptographic primitives, and then compare AMD with Intel, targeting at Athlon 64 and Pentium 4 processors. To do this, we started at looking at instruction-level performance. AMD has published a reliable list of an instruction latency and throughput of 64-bit instructions [20], but no similar information is currently available for Pentium 4. Our own experiments exhibit that some 64-bit instructions of Pentium 4 are unexpectedly slow, which can lead to serious and unavoidable performance penalties in programming encryption primitives.

Our first target primitive is AES [6]. We show that Athlon 64 is very suitable for an AES program mainly because it can issue up to two memory instructions in parallel. This boosts encryption speed of AES and, as a result, our program runs at the speed of 170 cycles/block, which is the fastest record of AES on a PC processor. Pentium 4 has a higher clock frequency than Athlon 64 in general, but still Athlon 64 seems to outperform Pentium 4 for AES.

The next target is Camellia [1]. the structure of Camellia should fit to 64-bit processors in nature. However, due to a long dependency chain, it was not easy to obtain high performance. We here propose a two-block parallel encryption, which can be used in a non-feedback mode such as the CTR mode, and show that its encryption speed on Athlon 64 reaches 175 cycles/block, thanks to the doubled number of registers.

We next develop a bitslice implementation of AES and Camellia. The bitslice technique for speeding up an encryption algorithm was introduced by Biham [4], which was remarkably successful for DES key search on 64-bit processors such as Alpha. We revisit this implementation from the viewpoint of a countermeasure against cache attacks [17].

A bitsliced cipher can achieve a good performance if the number of registers is many and a register size is long, which fully meets the x64 architecture. We carefully optimized the S-boxes of AES and Camellia in bit-level, and succeeded in obtaining very good and performance. Note that a bitslice implementation of AES was also discussed by Rudra et al. [18], but our paper for the first time reports a measured performance of bitsliced AES and Camellia implemented on a real processor.

Finally we study software performance of hash functions SHA256/512 and Whirlpool. Our x64 implementation results show that SHA512 can be faster than SHA256 on Athlon 64, and also faster than Whirlpool on both of Athlon 64 and Pentium 4 processors, unlike the results on 32-bit Pentium shown in [13][14].

Table 1 shows our reference machines and environments. Throughout this paper, we refer to Pentium 4 with Prescott core as Pentium 4; it is a current dominant core of Intel's EMT64 architecture.

Processor Name	AMD Athlon 64 3500+	Intel Pentium 4 HT
Core Name	Winchester	Prescott
Other Processor Info		Stepping 4 Revision 14
Clock Frequency	2.2GHz	3.6GHz
Cache (Code/Data)	64KB / 64KB	12Kμops / 16KB
Memory	1GB	1GB
Operation System	Windows XP 64-bit Edition	
Compiler	Microsoft Visual Studio 2005 beta	

Table 1. Our reference machines and environments.

2 The x64 Architecture

2.1 x86 vs x64

The x64 (or x86-64) is the first 64-bit processor architecture that is a superset of the x86 architecture. It was initially proposed and implemented by AMD and, in an ironic twist of processor history, later adopted by Intel under the name EM64T. Most of the extended features of the x64 architecture are what PC programmers long awaited:

1. The size of general registers is extended to 64 bits. The 32-bit `eax` register, for instance, is now lower half of the 64-bit `rax` register.
2. Additional eight general registers `r8-r15` and eight xmm registers `xmm8-xmm15` are introduced.
3. Almost all x86 instructions now accept 64-bit operands, including rotate shift instructions.

In particular the register increase has liberated PC programmers from the nightmare of register starvation. It is highly expected that these benefits open up new possibilities of fast and efficient cryptographic applications in near future. On the other side, using these extended features may cause the following new penalties, which can be serious in some cases:

1. An instruction requires an additional prefix byte in using a 64-bit operand or a new register. An increase in instruction length reduces decoding rate.
2. A 64-bit instruction is not always as efficient as its corresponding 32-bit instruction. Performance of an instruction might vary in 32-bit mode and 64-bit mode.

How fast a specific instruction runs is an issue of processor hardware design, not instruction set design. We will see that the second penalty above can be serious for Intel Pentium 4 processor and show that a great care must be taken when we implement a cryptographic algorithm on this processor.

2.2 Athlon 64 vs Pentium 4

Athlon 64 is a 3-way superscalar processor with 12 pipeline stages. It can decode and execute up to three instructions per cycle. Its three ALU's and three AGU's work independently and simultaneously, and moreover up to two 64-bit read/write instructions can access data cache each cycle in any combination of reads and writes. Hence, for example, an n-time repetition of the following (highly practical) code, which consists of five micro operations, works in n cycles, that is, 5μops/cycle.

```
xor  rax,TABLE1[rsi*8]        ; 64-bit load and 64-bit ALU
xor  rbx,TABLE2[rsi*8]        ; 64-bit load and 64-bit ALU
add  rsi,1                    ;                 64-bit ALU
```

Almost all of the 64-bit instructions of Athlon 64 runs in the same performance as its corresponding 32-bit x86 instructions, which is why Athlon 64 is often called a genuine 64-bit processor. The internal architecture of this processor is well-documented by AMD [20] and relatively easy to understand. A document written by Hans de Vries [21] is also helpful for understanding the architecture of Athlon 64. Working on this processor is generally less frustrating than exploring on Pentium 4.

One drawback of Athlon 64 is that it can fetch only 16 bytes of instructions from instruction cache per cycle. This means that the decoding stage can be still a bottleneck of performance, unlike Pentium 4. It is hence critically important for programmers to reduce an average instruction length for obtaining maximal performance on this processor.

A prominent feature of the Pentium 4 processor family is that instructions are cached after decoded, and hence the decoding capability is not a performance limiter any more, as long as a critical loop is covered by the trace cache (instruction cache) entirely.

There exist two different core architectures in the Pentium 4 family, of which we treat a newer one, the Prescott core, in this paper. Prescott has a deep 31-stage pipeline and achieves high clock frequency. At the time of writing, the highest frequency of the Athlon 64 family is 2.8GHz, while that of the Pentium 4 family is 3.8GHz, 36% faster than fastest Athlon 64.

Intel has not published pipeline architecture details of the Prescott core, nor documented information about how EM64T instructions are handled in its pipeline stages. To optimize a program on Pentium 4, we have to refer to not only Intel's document of the 32-bit architecture IA-32 [10], which is often erroneous, but also resources outside Intel such as Agner [8] Kartunov [12].

As far as we know, Prescott can run continuously three micro operations per cycle in an average (some resource says four, but we are not sure), which is less than Athlon 64. Moreover, many instructions of Prescott have a longer latency and/or a lower throughput than those of Athlon 64. This is a clear consequence of the high clock frequency of the Prescott core. Table 2 shows a brief comparative summary of these processors.

	Athlon 64	Pentium 4
Current highest clock frequency	2.8GHz	3.8GHz (good)
Decoding bottleneck	possible	mostly no (good)
Average instruction latency	low (good)	sometimes high
Maximal continuous execution rate	5μops/cycle (good)	3μops/cycle

Table 2. A simple comparison between Athlon 64 and Pentium 4.

Which of AMD and Intel is faster is always a controversial issue. We will show in this paper that in many cryptographic algorithms Athlon 64 outperforms Pentium 4 on the 64-bit platform, except a code using xmm instructions (SSE2), even if we take into consideration Pentium 4's faster clock frequency.

2.3 Instruction Latency and Throughput

Table 3 shows a list of an instruction latency (left) and throughput (right) of some of x64 instructions. We derived all the data in the list experimentally. Specifically, we measured the number of execution cycles of a code that consists of 100-1000 repetitions of a target instruction. Some fractional values on Pentium 4 are approximate. A latency of an instruction is n, when its result can be used in n cycles after the instruction has been issued. A throughput is the maximal number of the same instructions that can run continuously in parallel per cycle.

We believe that this table is of independent interest. It is quite surprising that 64-bit right shifts `shr` and 64-bit rotations `ror,rol` are extremely slow on Pentium 4. We do not know what was behind in this decision in Intel. Clearly this is a bad news for programmers of cryptographic algorithms.

Throughout this paper, we assume that a memory read/write is one μop each, and hence that `xor reg,[mem]` and `xor [mem],reg` consist of two μops and three μops, respectively. An exact μop break-down rule has not been published.

Processor	Pentium 4 (EM64T)		Athlon 64 (AMD64)	
Operand Size	32	64	32	64
`mov reg,[mem]`	4, 1	4, 1	3, 2	3, 2
`mov reg,reg`	1, 3	1, 3	1, 3	1, 3
`movzx reg,reg8L`	1, 3	1, 3	1, 3	1, 3
`movzx reg,reg8H`	2, 4/3	-	1, 3	-
`add reg,reg`	1, 2.88	1, 2.88	1, 3	1, 3
`sub reg,reg`	1, 2.88	1, 2.88	1, 3	1, 3
`adc reg,reg`	10, 2/5	10, 2/5	1, 5/2	1, 5/2
`sbb reg,reg`	10, 2/5	10, 2/5	1, 5/2	1, 5/2
`xor/and/or reg,reg`	1, 7/4	1, 7/4	1, 3	1, 3
`not reg`	1, 7/4	1, 7/4	1, 3	1, 3
`shr reg,imm`	1, 7/4	7, 1	1, 3	1, 3
`shl reg,imm`	1, 7/4	1, 7/4	1, 3	1, 3
`ror/rol reg,imm`	1, 1	7, 1/7	1, 3	1, 3

Processor	Pentium 4 (EM64T)	Athlon 64 (AMD64)
Operand Size	128	128
`movdqa xmm,[mem]`	-, 1	-, 1
`movdqa xmm,xmm`	7, 1	2, 1
`movd xmm,reg + movd reg,xmm`	13, -	14, -
`paddb/paddw/paddd xmm,xmm`	2, 1/2	2, 1
`paddq xmm,xmm`	5, 2/5	2, 1
`pxor/pand/por xmm,xmm`	2, 1/2	2, 1
`psllw/pslld/psllq xmm,xmm`	2, 1/2	2, 1
`pslldq xmm,xmm`	4, 1/2	2, 1
`psrlw/psrld/psrlq xmm,xmm`	2, 1/2	2, 1
`psrldq xmm,xmm`	4, 1/2	2, 1

Table 3. A list of an instruction latency and throughput of Pentium 4 and Athlon 64.

After our creating this table, we found an independent (but not formally published) result obtained by Granlund [9]. Table 3 contains several results that are not covered in [9]. Our results mostly agree with Granlund's for Athlon 64, but look slightly different for some instructions of Pentium 4. It is known that Pentium 4 Prescott has many variations (stepping, revision), which can lead to subtly different instruction timings. Since Intel has not published detailed information on the hardware design of Prescott, it is difficult to derive the precise timing information.

3 AES

First we discuss a fast implementation of AES on the x64 architecture. See [6] for the detailed specification of the AES algorithm. In [13], an x86 code of the "basic component" of AES, which corresponds to `Subbytes+Shiftrows+Mixcolumns`, was proposed for Pentium 4 processors. Code 1 shows the proposed code with a modification for the x64 platform. One round of AES, except the final round, can be implemented with four additional xor instructions, which corresponds to `AddRoundKey`, and four-time repetition of the basic component:

```
movzx    esi,al                  ; first address
mov/xor  reg32_1,table1[rsi*4]   ; first table lookup
movzx    esi,ah                  ; second address
mov/xor  reg32_2,table2[rsi*4]   ; second table lookup
shr      eax,16
movzx    esi,al                  ; third address
mov/xor  reg32_3,table3[rsi*4]   ; third table lookup
movzx    esi,ah                  ; fourth address
mov/xor  reg32_4,table4[rsi*4]   ; fourth table lookup
```

Code 1. An x64 implementation of the basic component of AES.

Alternatively, the last two lines can be also written in the following form, which was recommended for Pentium 4 with Prescott core, where using `ah` is a bit expensive:

```
shr       eax,8                    ; fourth address
mov/xor   reg32_4, table4[rax*4]   ; fourth table lookup
```

Code 2. An alternative implementation of the basic component (part).

Since each of the four tables occupies 1KB and we need additional four tables for the final round, a total of 8KB data memory is needed for the entire AES tables. This implementation is highly optimized and well scheduled, and hence also works on 64-bit environments excellently.

Readers might be tempted to write the following code instead, which looks more “genuine” 64-bit style:

```
movzx    rsi,al
mov/xor  reg32_1,table1[rsi*4]
movzx    rsi,ah                     ; no such instruction
mov/xor  reg32_2,table2[rsi*4]
shr      rax,16                     ; slow on Pentium 4
movzx    rsi,al
mov/xor  reg32_3,table3[rsi*4]
movzx    rsi,ah                     ; no such instruction
mov/xor  reg32_4,table4[rsi*4]
```

Code 3. This code does not work.

Code 3 does not work since a higher 8-bit partial register such as `ah` can be used only in x86 code (the third and eighth lines), which is one of the small number of exceptional instructions that do not have an extended 64-bit form. If we change `movzx rsi,ah` into the original form `movzx esi,ah`, then code 3 works as expected, but is still slow on Pentium 4 because `shr rax,16` is a 64-bit right shift instruction.

Moreover, since the number of higher 8-bit registers are still limited to four (`ah,bh,ch,dh`) in the x64 environment, we have to assign `eax,ebx,ecx,edx` to `reg32_1,...,reg32_4`, which are used as address registers in the next round, to minimize the number of instructions, but this is impossible without saving/restoring at least one input register in each round. In the x86 environment, we had to access temporary memory for this due to register starvation, but in the x64, we can use a new register instead, which slightly improves performance.

In summary, we should keep an x86 style, using new registers for temporary memory in implementing AES on x64 environments. Table 4 summarizes our implementation results of the AES algorithm with 128-bit key on Athlon 64 and Pentium 4 processors, where the right most column shows the best known result on 32-bit Pentium 4:

Our program runs very fast on the Athlon 64 processor. As far as we know, this is the fastest AES implementation ever made on a PC processor; faster than Pentium 4 even if we take into consideration higher clock frequency of the Pentium 4 processor. This is mainly because Athlon 64 can execute two memory load instructions with 3 latency cycles in parallel. The number of memory reads for one block encryption of AES is 4 (for plaintext loads) + 11×4 (for subkey loads) + 16×10 (for table lookups) = 208, which means that Pentium 4 takes at least 208 cycles/block for one block encryption.

Considering an instruction latency of Athlon 64, the theoretical limit of AES performance on this processor seems around 16 cycles/round = 160 cycles/block. Our result is hence reaching closely this limit.

Processor	Athlon 64 64-bit	Pentium 4 64-bit	Pentium 4 32-bit [13]
cycles/block	170	256	284
cycles/byte	10.6	16.0	17.8
instructions/cycle	2.74	1.81	-
μops/cycle	3.53	2.34	-

Table 4. Our implementation results of AES with 128-bit key.

4 Camellia

The next example of our implementation is another 128-bit block cipher Camellia [1]. Recently Camellia has been adopted in the NESSIE project [16], Japan's CRYPTREC project [5] and also the ISO/IEC 18033-3 standard [11].

Camellia supports three key sizes; 128 bits, 192 bits and 256 bits as AES, where we treat the 128-bit key version. The basic structure of Camellia is Feistel type, consisting eighteen rounds with additional four small FL functions. Figures 1 and 2 show the F-function and the FL-function, respectively.

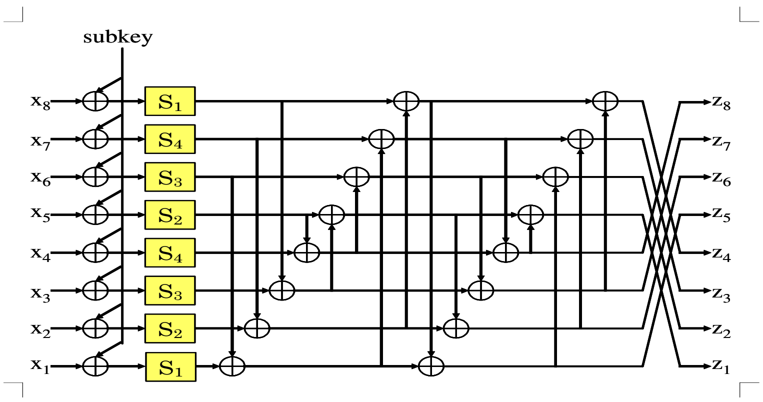

Fig. 1. The F-function of Camellia.

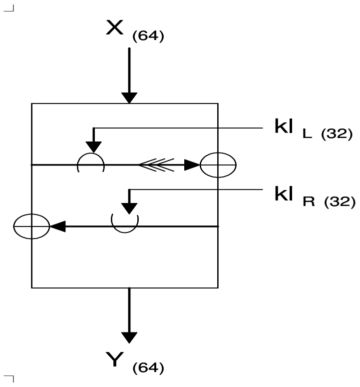

Fig. 2. The FL-function of Camellia.

The F-function has a typical SP-structure of eight-byte width, which is quite suitable for 64-bit processors. Code 4 shows an implementation of (half of) the F-function of Camellia with the minimum number of x64 instructions, which should get the benefits of 64-bit registers:

```
movzx    esi,al                   ; first address
xor      rbx,table1[rsi*4]        ; first table lookup
movzx    esi,ah                   ; second address
xor      rbx,table2[rsi*4]        ; second table lookup
shr      rax,16
movzx    esi,al                   ; third address
xor      rbx,table3[rsi*4]        ; third table lookup
movzx    esi,ah                   ; fourth address
xor      rbx,table4[rsi*4]        ; fourth table lookup
shr      rax,16
...                               ; fifth to eighth
```

Code 4. An x64 implementation of the F-function of Camellia.

In practice, however, this code does not run very fast because of a long dependency chain; especially the xor chain hinders parallel execution of multiple instructions. Although we can obtain some performance improvement by introducing some intermediate variables to cut the dependency chain, the resultant performance gain is limited.

On the other side, it should be noted that code 4 is using only three registers. Therefore it is possible to compute two blocks in parallel without register starvation, which is expected to boost the performance. This parallel computation method can be applied to a non-feedback mode of operation, such as a counter mode. Our optimized code for encrypting two blocks of Camellia in parallel, where they are interleaved in every half round, runs in 175 cycles/block on Athlon 64; this performance is almost the same as that of AES.

On the other hand, the performance on Pentium 4 is still poor because of a long latency of the 64-bit right shift instructions. Probably a code without using 64-bit instructions could be faster on Pentium 4. Table 5 summarizes our implementation results of (two-block parallel) Camellia. Further optimization efforts on Pentium 4 are now ongoing.

Processor	Athlon 64	Pentium 4
cycles/block	175	457
cycles/byte	10.9	28.6
instructions/cycle	2.46	0.94
μops/cycle	3.28	1.26

Table 5. Performance of our two-block parallel program of Camellia with 128-bit key.

Code 5 is our implementation of the FL-function on 64-bit registers without dividing a 64-bit input into two 32-bit halves. The required number of instructions of this tricky code is a bit smaller than that of a straightforward method.

```
mov   rcx,[key]          ; load 64 bits (klL+klR)
and   rcx,rax            ; rax = input
shr   rcx,32
rol   ecx,1
xor   rax,rcx
mov   ecx,[key]          ; load 32 bits again (klR)
or    ecx,eax
shl   rcx,32
xor   rax,rcx            ; rax = output
```

Code 5. An x64 implementation of the FL-function of Camellia.

5 Bitslice Implementation

This section discusses a bitslice implementation of AES and Camellia. The bitslice implementation was initially proposed by Biham [4], which makes an n-block parallel computation possible, where n is a block size and one (software) instruction corresponds to n simultaneous one-bit (hardware) operations, by regarding the i-th bit of register j as the j-th bit of the i-th block.

In general, this implementation can be faster than an ordinary implementation when the following conditions are met:

- The bit-level complexity of the target algorithm is small.
- The number of registers of the target processor is many.
- The size of registers of the target processor is long.

The bitslice implementation was successful for DES [4] and MISTY [15] on the Alpha processor, since these algorithms are small in hardware and Alpha has thiry-two 64-bit general registers.

It is obvious that there is no hope of gaining performance using the bitslice technique on x86 processors, which have only eight 32-bit registers. However we have now sixteen 64-bit registers and it is now an interesting topic to see to what extent the x64 architecture contributes to fast encryption. Note that 128-bit xmm registers are of no use, due to a poor latency and throughput of SSE2 instructions.

Moreover, a program written in the bitslice method does not use any table lookups with a key-dependent address. This means that a bitsliced code is safe against implementation attacks such as cache timing attacks [17]. As far as we know, this is the first paper that describes bitslice implementations of AES and Camellia on an actual processor.

Clearly the most critical part of the bitslice program of these cipher algorithms is a design of the 8x8 S-boxes. To minimize the number of instructions of each of the S-boxes, which is composed of an inversion function over $GF(2^8)$ and a linear transformation for either of AES and Camellia, we should look at its hardware implementation, not software, due to the nature of the bitslice implementation.

Satoh et. al [19] proposed to design an inversion circuit of $GF(2^8)$ using circuits of $GF(2)$ in hardware by recursively applying circuits of a subfield of index two. We further considered an optimality of linear transformations that are required before and after the inversion function to design the entire S-box structure, and reached the following basis of $GF(2^8)$ over $GF(2)$ for a bitslice S-box with a small number of instructions:

$$(1, \beta^5, \beta, \beta^6, \alpha, \beta^5\alpha, \beta\alpha, \beta^6\alpha),$$

where $\alpha^8 + \alpha^6 + \alpha^5 + \alpha^3 + 1 = 0$, and $\beta = \alpha^6 + \alpha^5 + \alpha^3 + \alpha^2 \in GF(2^4)$.

Table 6 shows the number of x64 instructions required for implementing the S-box of AES and Camellia, respectively. The "Before inversion"/"After inversion" column shows the number of register-register logical instructions required for the linear transformation before/after the Galois field inversion, respectively. The two numbers of the "Inversion on $GF(2^8)$" column shows the number of register-register logical instructions and register-memory load/store instructions. In the inversion part, all of the fifteen 64-bit general registers are used except the stack register, and additional five 64-bit temporary memory areas are needed. Appendix B shows a source code of our implementation of this AES S-box.

	Before inversion	Inversion on $GF(2^8)$	After inversion	Total
AES S-box	12	156(reg) + 21(mem)	16	205
Camellia S-box	12	156(reg) + 21(mem)	14	203

Table 6. The number of x64 instructions of the S-box of AES and Camellia

Using this S-box implementation, we made the entire bitslice programs of AES and Camellia. Table 7 shows the resultant performance of our codes.

The speed shown in this table is slower than that of the ordinary implementation method shown in the previous sections, but is still in a very practical level. Note that in the bitslice implementation Camellia is slightly faster than the bitsliced AES. This is mainly because Camellia has a fewer number of S-boxes (144) than AES (160), though Camellia has an additional FL-function. Also the performance of Athlon 64 is excellent again, since three logical instructions can run on Athlon in parallel, but only two on Pentium 4.

Algorithm	AES		Camellia	
Processor	Athlon 64	Pentium 4	Athlon 64	Pentium 4
cycles/block	250	418	243	415
cycles/byte	15.6	26.1	15.2	25.9
instructions/cycle	2.75	1.66	2.74	1.61
μops/cycle	3.20	1.93	2.99	1.75

Table 7. Our implementation results of bitsliced AES and Camellia with 128-bit key

6 Hash Functions: SHA256/512 and Whirlpool

This section briefly shows our implementations of three recent hash functions SHA256, SHA512 [7] and Whirlpool [2][3], all of which are now under consideration for an inclusion in the new version of the ISO/IEC 10118 standard. Note that the message block size of SHA256 and Whirlpool is 64 bytes, while SHA512 has a 128-byte message block.

Table 8 summarizes our performance results of SHA256, where the first column presents an ordinary implementation using general registers, and the second column shows a four-block parallel implementation (in the sense of [13][14]). It is seen that the x64 code of Athlon 64 establishes an excellent superscalability, 2.88 instructions/cycle, which is very close to its structural limit, 3 instructions/cycle. On the other hand, Pentium 4 is faster than Athlon 64 on the xmm code, considering Pentium 4's faster clock frequency.

Processor	Athlon 64		Pentium 4	
Instructions	x64 (1b)	xmm (4b)	x64 (1b)	xmm (4b)
cycles/block	1173	1154	1600	1235
cycles/byte	18.3	18.0	25.0	19.3
instruction/cycle	2.88	1.15	2.11	1.08
μops/cycle	3.16	1.22	2.31	1.14

Table 8. Our implementation results of SHA256.

Table 9 illustrates our implementation results of SHA512, where the second column shows a two-block parallel code using xmm instructions. Also the right two columns are previous results shown on [13]. A remarkable fact is that SHA512 runs faster than SHA256 on Athlon 64 because an 64-bit instruction runs in the same latency/throughput as its corresponding 32-bit instruction on this processor. On the other side, Pentium 4 is very slow due to a long latency of 64-bit rotate operations, which are unavoidable in programming SHA512.

Our final example is Whirlpool. Our experimental results presented in Table 10 shows that Whirlpool is not faster than SHA512 on either of Athlon 64 and Pentium 4 in 64-bit environments.

Processor	Athlon 64		Pentium 4		Pentium 4 [13]	
Instructions	x64 (1b)	xmm (2b)	x64 (1b)	xmm (2b)	mmx (2b)	xmm (4b)
cycles/block	1480	2941	3900	3059	5294	3111
cycles/byte	11.6	23.0	30.5	23.9	41.4	24.3
instruction/cycle	2.85	1.15	1.08	1.10	-	-
μops/cycle	3.17	1.21	1.20	1.14	-	-

Table 9. Our implementation results of SHA512.

Processor	Athlon 64	Pentium 4	Pentium 4 [13]
Instructions	x64	x64	mmx
cycles/block	1537	2800	2319
cycles/byte	24.0	43.8	36.2
instruction/cycle	2.27	1.24	-
μops/cycle	3.08	1.69	-

Table 10. Our implementation results of Whirlpool.

7 Concluding Remarks

This paper explored the state-of-the-art implementation techniques for speeding up symmetric primitives on the x64 architecture. In many cases Athlon 64 attains better performance than Pentium 4 EM64T, even if Pentium 4's higher clock frequency is taken into consideration. Probably the slow 64-bit right shifts and 64-bit rotations of Pentium 4 will be (should be) redesigned in the next core architecture for EM64T.

We also showed the first bitslice implementation of AES and Camellia on these processors and demonstrated that our program achieved very good performance. We believe that a bitslice implementation has a significant and practical impact from the viewpoint of resistance from cache timing attacks. For interested readers, we summarize the coding style we adopted and how we measured clock cycles of our programs in appendix A, and list an assembly language source code of a bitsliced S-box of AES in appendix B.

References

[1] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, T. Tokita: "The 128-Bit Block Cipher Camellia", IEICE Trans. Fundamentals, Vol.E85-A, No.1, pp.11-24, 2002.

[2] P. Barreto, V. Rijmen: "The Whirlpool Hashing Function", Proceedings of First Open NESSIE Workshop, Heverlee, Belgium, 2000.

[3] P. Barreto: "The Whirlpool Hash Function", `http://planeta.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`

[4] E. Biham: "A Fast New DES Implementation in Software", Proceedings of Fast Software Workshop FSE'97, Lecture Notes in Computer Science, Vol.1267, pp.260-272, Springer-Verlag, 1997.

[5] Cryptography Research and Evaluation Committees: The CRYPTREC Homepage http://www.cryptrec.org/

[6] Federal Information Processing Standards Publication 197, "Advanced Encryption Standard (AES)", NIST, 2001.

[7] Federal Information Processing Standards Publication 180-2, "Secure Hash Standard", NIST, 2002.

[8] A. Fog: "How To Optimize for Pentium Family Processors", Available at http://www.agner.org/assem/

[9] T. Granlund: "Instruction latencies and throughput for AMD and Intel x86 Processors", Available at http://swox.com/doc/x86-timing.pdf

[10] IA-32 Intel Architecture Optimization Reference Manual, Order Number 248966-011, http://developer.intel.ru/download/design/Pentium4/manuals/24896611.pdf

[11] ISO/IEC 18033-3, "Information technology - Security techniques - Encryption algorithms - Part3: Block ciphers", 2005.

[12] Victor Kartunov: "Prescott: The Last of the Mohicans? (Pentium 4: from Willamette to Prescott)" http://www.xbitlabs.com/articles/cpu/display/netburst-1.html

[13] M. Matsui, S. Fukuda: "How to Maximize Software Performance of Symmetric Primitives on Pentium III and 4 Processors", Proceedings of Fast Software Workshop FSE2005, Lecture Notes in Computer Science, Vol.3357, pp.398-412, Springer-Verlag, 2005.

[14] J. Nakajima, M. Matsui: "Performance Analysis and Parallel Implementation of Dedicated Hash Functions on Pentium III", IEICE Trans. Fundamentals, Vol.E86-A, No.1, pp.54-63, 2003.

[15] J. Nakajima, M. Matsui: "Fast Software Implementations of MISTY1 on Alpha Processors", IEICE Trans. Fundamentals, Vol.E82-A, No.1, pp.107-116, 1999.

[16] New European Schemes for Signatures, Integrity, and Encryption (NESSIE), https://www.cosic.esat.kuleuven.ac.be/nessie/

[17] D. A. Osvik, A. Shamir, E. Tromer: "Full AES key extraction in 65 milliseconds using cache attacks" Crypto 2005 rump session.

[18] A. Rudra, P. Dubey, C. Jutla, V. Kummar, J. Rao, P. Rohatgi: "Efficient Rijndael Encryption Implementation with Composite Field Arithmetic", Proceedings of CHES 2001, Lecture Notes in Computer Science, Vol.2162, pp.171-184, Springer-Verlag, 2001.

[19] A. Satoh, S. Morioka, K. Takano, S. Munetoh: "A Compact Rijndael Hardware Architecture with S-Box Optimization", Proceedings of Asiacrypt 2001, Lecture Notes in Computer Science, Vol.2248, pp.239-254, Springer-Verlag, 2001.

[20] Software Optimization Guide for AMD64 Processors, Publication 25112, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF

[21] Hans de Vries: "Understanding the detailed Architecture of AMD's 64 bit Core", http://chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html

Appendix A: Coding Style and How to Measure Cycles

The coding style of our programs is basically the same as that in [13]. Our programs are thread-safe; that is, we did not use any static memory area except read-only constant tables. Also we did not write any key dependent code such as a self-modifying trick.

Our assembly codes have the following interface that is callable from C language, and we assume that the subkey has been given in the third argument. In the x64 environments, the arguments are usually passed through registers, not through stack, but this calling convention does not affect performance of encryption functions seriously if `block` is sufficiently large.

We also assume that all addresses are appropriately aligned, at least on a 16-byte boundary to reduce possible mis-alignment penalties.

```
Function( uchar *plaintext, uchar *ciphertext, uint *subkey, int block )
```

The method for measuring a speed of `Function` that we adopted is that using the `cpuid+rdtsc` instruction sequence, which is common in x86 processors, as shown below:

```
xor    eax,eax
cpuid                            ; pipeline flush
rdtsc                            ; read time stamp
mov    CLK1,eax                  ; current time
xor    eax,eax
cpuid

Function(..., int block)

xor    eax,eax
cpuid                            ; pipeline flush
rdtsc                            ; read time stamp
mov    CLK2,eax                  ; current time
xor    eax,eax
cpuid
```

Code 6. A code sequence for measuring a speed of `Function`.

We first ran the code above and recorded `CLK2-CLK1`. Then we removed `Function` from the code, ran again the code and recorded `CLK2-CLK1`. Since the second record is an overhead of the measurement itself, we subtracted the second record from the first record, then divded it by `block` and adopted the resultant value as "`cycles/block`". In practice, we made the measurement 100 times, of which we removed exceptional cases due to, for instance, an interruption caused by an operation system, and took an average on the remaning cases.

Strictly speaking the `rdtsc` instruction returns 64-bit clock tics to `edx` and `eax`, but we used only lower 32 bits, because if an overflow of `eax` took place during the measurement, it could be removed as an exceptional case.

Appendix B: Source Program of Bilsliced AES Sbox

This appendix shows a source code of our bitslice implementation of the Sbox of AES, which is written in x64 assembly language with Microsoft MASM syntax. The complete program is described as a macro with eight register inputs and eight register outputs. We wrote several instructions in a single line for saving space.

```
;************************************************
;*  Bitslice Implementation of Sbox of AES      *
;*  Using x64 Instructions (AMD64 / EM64T)      *
;*                                              *
;*  Input  (rax,rbx,rcx,rdx,rbp,r8,r9,r10)      *
;*  Output (rbx,rdx,rax,r15,rbp,rcx,r10,r9)     *
;*                                              *
;*  205 Instructions (184 logical 21 memory)    *
;*   40 Temporary Memory Bytes                  *
;*                                              *
;*  (C) Mitsuru Matsui  2005,2006               *
;************************************************

SBOX  MACRO
    InBasisChange   rax,rbx,rcx,rdx,rbp,r8,r9,r10
    Inv_GF256       r10,rbp,rbx,rcx,rdx,r8,r9,rax,r11,r12,r13,r14,r15,rsi,rdi
    OutBasisChange  r10,rbp,rbx,rcx,rdx,r8,r9,rax,r15
ENDM

;**************************
;*  InBasisChange:  (12)  *
;**************************

InBasisChange  MACRO  g0,g1,g2,g3,g4,g5,g6,g7
    xor  g6,g5    xor  g6,g1    xor  g5,g4    xor  g7,g5    xor  g4,g3
    xor  g4,g0    xor  g0,g2    xor  g7,g0    xor  g3,g2    xor  g2,g6
    xor  g3,g1    xor  g6,g4
ENDM

;***************************
;*  OutBasisChange:  (16)  *
;***************************

OutBasisChange  MACRO  g0,g1,g2,g3,g4,g5,g6,g7,g8
    xor  g1,g3    xor  g1,g5    xor  g1,g0    mov  g8,g1    xor  g8,g2
    xor  g1,g4    xor  g2,g6    xor  g6,g1    xor  g1,g7    xor  g7,g2
    xor  g2,g3    xor  g3,g5    xor  g3,g0    xor  g0,g4    xor  g0,g7
    xor  g4,g5
    ;We can skip the follwing four NOTs by modifying subkey in advance.
    ;not  g6      ;not  g0      ;not  g7      ;not  g4
ENDM
```

```
;***********************************************************
;*  Mul_GF4:  Input x0-x1,y0-y1  Output x0-x1  Temp t0  (8)  *
;***********************************************************

Mul_GF4  MACRO  x0,x1,y0,y1,t0
    mov  t0,x1    xor  x1,x0    and  x0,y0    and  t0,y1
    xor  y0,y1    and  x1,y0    xor  x1,x0    xor  x0,t0
ENDM

;********************************************
;*  Inv_GF4:  Input x0,x1  Output x0,x1  (2)  *
;********************************************

Inv_GF4  MACRO  x0,x1
    and  x0,x1    not  x0
ENDM

;*****************************************************************
;*  Mul_GF16:  Input x0-x3,y0-y3  Output x0-x3  Temp t0-t3  (35)  *
;*****************************************************************

Mul_GF16  MACRO  x0,x1,x2,x3,y0,y1,y2,y3,t0,t1,t2,t3
    mov  t0,x2    mov  t1,x3    mov  t2,y3
    mov  t3,t0    and  t3,t2    and  t0,y2    and  t2,t1    and  t1,y2
    xor  t1,t3    xor  t0,t1    xor  t1,t2
    xor  x2,x0    xor  x3,x1    xor  y2,y0    xor  y3,y1

    Mul_GF4  x2,x3,y2,y3,t3
    Mul_GF4  x0,x1,y0,y1,t3

    xor  x2,x0    xor  x3,x1    xor  x0,t1    xor  x1,t0
ENDM

;***********************************************************
;*  Inv_GF16:  Input x0-x3  Output x0-x3  Temp t0-t3  (34)  *
;***********************************************************

Inv_GF16  MACRO  x0,x1,x2,x3,t0,t1,t2,t3
    mov  t0,x0    mov  t1,x1    xor  t0,x2    xor  t1,x3    mov  t2,t0

    Mul_GF4  x0,x1,t0,t1,t3

    xor  x0,x3    xor  x1,x2

    Inv_GF4  x0,x1

    mov  t0,x0

    Mul_GF4  x2,x3,t0,x1,t3
    Mul_GF4  x0,x1,t2,t1,t3
```

```
ENDM

;*******************************************************************
;*  Inv_GF256:  Input x0-x7  Output x0-x7  Temp t0-t3,s0-s2  (177)  *
;*******************************************************************

Inv_GF256  MACRO  x0,x1,x2,x3,x4,x5,x6,x7,t0,t1,t2,t3,s0,s1,s2
    mov  t0,x0    mov  t1,x1    mov  t2,x2    mov  t3,x3    xor  t0,x4
    xor  t1,x5    xor  t2,x6    xor  t3,x7

    mov  [rsp+0],t0     mov  [rsp+8],t1
    mov  [rsp+16],t2    mov  [rsp+24],t3
    mov  [rsp+32],x7

    Mul_GF16  x0,x1,x2,x3,t0,t1,t2,t3,s0,s1,s2,x7

    mov  x7,[rsp+32]

    xor  x0,x4    xor  x1,x4    xor  x2,x4    xor  x3,x4    xor  x1,x5
    xor  x3,x5    xor  x2,x6    xor  x2,x7    xor  x3,x7

    Inv_GF16  x0,x1,x2,x3,t0,t1,t2,t3

    mov  t0,[rsp+0]     mov  t1,[rsp+8]
    mov  t2,[rsp+16]    mov  t3,[rsp+24]
    mov  [rsp+0],x0     mov  [rsp+8],x1
    mov  [rsp+16],x2    mov  [rsp+24],x3

    Mul_GF16  x0,x1,x2,x3,t0,t1,t2,t3,s0,s1,s2,x7

    mov  t0,[rsp+0]     mov  t1,[rsp+8]
    mov  t2,[rsp+16]    mov  t3,[rsp+24]
    mov  [rsp+24],x3    mov  x7,[rsp+32]

    Mul_GF16  x4,x5,x6,x7,t0,t1,t2,t3,s0,s1,s2,x3

    mov  x3,[rsp+24]
ENDM
```