On the Power of Bitslice Implementation on Intel Core2 Processor

Mitsuru Matsui and Junko Nakajima

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

Abstract. This paper discusses the state-of-the-art fast software implementation of block ciphers on Intel's new microprocessor Core2, particularly concentrating on "bitslice implementation". The bitslice parallel encryption technique, initially proposed by Biham for speeding-up DES, has been successful on RISC processors with many long registers, but on the other side bitsliced ciphers are not widely used in real applications on PC platforms, because in many cases they were actually not very fast on previous PC processors. Moreover the bitslice mode requires a non-standard data format and hence an additional format conversion is needed for compatibility with an existing parallel mode of operation, which was considered to be expensive.
This paper demonstrates that some bitsliced ciphers have a remarkable performance gain on Intel's Core2 processor due to its enhanced SIMD architecture. We show that KASUMI, a UMTS/GSM mobile standard block cipher, can be four times faster when implemented using a bitslice technique on this processor. Also our bitsliced AES code runs at the speed of 9.2 cycles/byte, which is the performance record of AES ever made on a PC processor. Next we for the first time focus on how to optimize a conversion algorithm between a bitslice format and a standard format on a specific processor. As a result, the bitsliced AES code can be faster than a highly optimized "standard AES" code on Core2, even taking an overhead of the conversion into consideration. This means that in the CTR mode, bitsliced AES is not only fast but also fully compatible with an existing implementation and moreover secure against cache timing attacks, since a bitsliced cipher does not use any lookup tables with key/data-dependent address.

Keywords: *Fast Software Encryption, Bitslice, AES, KASUMI, Core2*

1 Introduction

The purpose of this paper is to study software performance optimization techniques for symmetric primitives on PC processors, particularly focusing on "bitslice implementation" on Intel's new Core2 microprocessor, and show that, by fully utilizing its enhanced SIMD instructions, many important ciphers such as

KASUMI, AES and Camellia can be much faster than previously expected with keeping full compatibility with an existing parallel mode of operation.

The bitslicing technique was introduced by Biham [5] in 1997 for speeding-up DES, which was actually implemented on several processors and used for brute force key search of DES in the distributed.net project [7]. In the bitslice implementation one software logical instruction corresponds to simultaneous execution of n hardware logical gates, where n is a register size, as shown in figure 1 Hence bitslicing can be efficient when the entire hardware complexity of a target cipher is small and an underlying processor has many long registers.

Therefore the bitslice implementation is usually successful on RISC processors such as Alpha, PA-RISC, Sparc, etc, but unfortunately was not considered to be very attractive on Intel x86 processors in many cases due to the small number of registers. While several papers already discussed bitslice techniques of block ciphers [4][14][15][18][20], as far as we know, only one paper reported actually measured performance of a real bitslice code of AES on a PC processor [14]. Moreover a conversion of data format is required for compatibility with an existing parallel mode of operation such as the CTR mode, but no papers have investigated an overhead of this conversion in a real platform.

In [14] we studied an optimization of AES on 64-bit Athlon64 and Pentium4 processors, where his bitsliced AES ran still (or only) 50% slower than an optimized standard AES (i.e. a code written in a usual block-by-block style). The bitsliced AES code shown in the paper was implemented on 64-bit general registers, not on 128-bit XMM registers. This was because on these processors XMM instructions were more than two times slower than the corresponding x64 instructions and hence using 128-bit instructions did not have any performance advantage. Also note that we did not include an overhead of format conversion in the cycle counts.

Our current paper gives performance figures of several highly optimized bitsliced block ciphers on Intel's new Core2 processor, which was launched into PC market last summer and has since been very widely used in desktop and mobile PCs. Core2 has several significant improvements over previous processors, of which the most advantageous one for us is that its all execution ports support full 128-bit data. Three logical 128-bit XMM instructions can now run in parallel (although some hidden stall factors still remain as previous Intel processors), which is expected to boost performance of a bitsliced cipher.

First we implement KASUMI, a UMTS/GSM standard cipher, in both standard and bitslice modes. We show an optimization technique for a single block encryption, which results in 36.3 cycles/byte. On the other side, our bitsliced code runs at the speed of 9.3 cycles/byte, four times faster, thanks to its hardware-oriented lookup tables and improved XMM instructions of the Core2 processor. Since the mode of operation adopted in the UMTS standard is not a parallel mode, this bitslice technique cannot be direct applied to a handset, but can be used in a radio network controller, which has to treat many independent data streams.

Our next target is AES in the bitslice mode, fully utilizing 128-bit XMM registers and instructions. Our optimized code has achieved the encryption speed of 9.2 cycles/byte on Core2, which is the highest speed of AES ever achieved on a PC processor. Also we present a specific code sequence for converting data between a bitslice mode and a standard mode. This format conversion is essentially an entire bitwise data reallocation, which was believed to be expensive. Our conversion algorithm fully utilizes SIMD instructions and successfully runs in less than 1 cycle/byte.

As a result, we conclude that bitsliced AES that is fully compatible with the CTR mode can run still faster than highly optimized standard AES on Core2. Moreover note that a bitslice cipher is safe against implementation attacks such as cache timing attacks [17]. We believe that the bitslice implementation is in fact very promising in real applications in current and future PC processors.

Table 1 shows our reference machines and environments.

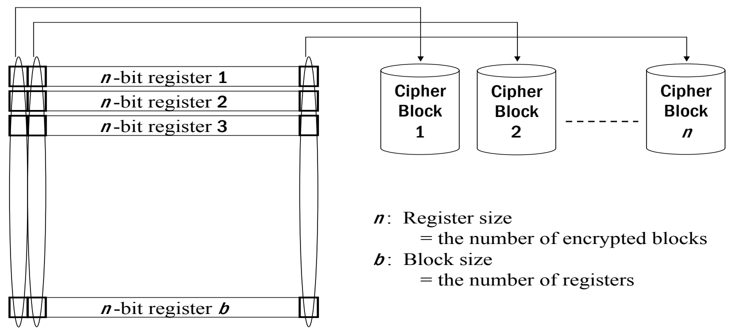

Fig. 1. The basic concept of bitslicing.

Processor Name	Intel Pentium 4 561	AMD Athlon 64 3500+	Intel Core2 Duo E6400
Core Name	Prescott	Winchester	Conroe
Clock Frequency	3.6GHz	2.2GHz	2.13GHz
Cache (Code/Data)	12Kμops / 16KB	64KB / 64KB	32KB / 32KB
Memory	1GB	1GB	1GB
Operation System	Windows XP 64-bit Edition		
Compiler	Microsoft Visual Studio 2005		

Table 1. Our reference machines and environments.

2 Core2 Architecture

This section briefly describes internal architecture of Core2 and points out what a programmer of symmetric algorithms should notice in optimizing performance on this processor. Intel has not published details of its pipeline hardware mechanism, and moreover undocumented pipeline stalls are often observed. This section hence largely comes from external resources [9][11] and our own experimental results.

Intel Core2 processor came after Pentium 4, which one-side focused on high clock frequency and reached its dead end due to an overheating problem. The pipeline of Core2 has 14 stages, significantly shorter than that of Pentium 4, aiming at higher superscalarity rather than higher frequency as shown below. The pipeline of Core2 includes the following stages:

- **Instruction Fetch and Predecoding**
  Instructions are fetched from memory and sent to the predecoder, which detects where each instruction begins. Unfortunately the predecoder can process only 16 bytes/cycle, which is very likely a performance bottleneck. So using a short instruction and a short offset is essential for optimization. For instance, three XMM "xor" instructions `xorps`, `xorpd` and `pxor` are logically equivalent, but the second and third ones are one byte longer than the first one. The same applies to `movaps`, `movapd` and `movdqa`. Another example is that using registers `xmm8` to `xmm15` leads to an additional prefix byte.

- **Instruction Decoding**
  In this stage, an instruction is broken down into micro-operations (μops). Core2 can treat a read-modify instruction as one μop, called a fused μop, while previous processors counted it as two μops. The same applies to a memory write instruction. Since an instruction consisting of two or more μops can be decoded in only one of the four decoders of Core2, this fusion mechanism greatly improves decoding efficiency. We expect that the decoding stage is not a performance bottleneck in programming a block cipher.

- **Register Renaming**
  In this stage a register to be written or modified is renamed into a virtual register, and then μops are sent to the reordering buffer. This stage can handle up to 4μops/cycle, which is the overall performance limitation of Core2. In other words, assembly programmer's objective is to write a code that runs at the speed of (as close as possible to) 4μops/cycle. Also this stage contains another bottleneck factor called "register read stall"; i.e. only two registers can be renamed per cycle, excluding those that have been modified within the last few cycles [9]. We hence have to avoid registers that are frequently read without being written. It is however difficult to avoid this stall without causing another penalty in practice.

- **Execution Units**
  A fused μop is finally broken down into unfused μops, which are issued toward execution units. Core2 has a total of six ports; three for ALUs, one for read, one for write address, and one for write data. A very good news for

us is that all ports support the full 128-bit data and each of the three ALUs independently accept a 128-bit XMM logical instruction with throughput and latency 1. This is a remarkable improvement of Core2 over previous processors such as Pentium 4 and Athlon 64, and is the most contributing factor in high speed encryption in the bitslice mode.

Table 2 shows a list of latency (left) and throughput (right) of instructions frequently used in a block cipher code on Pentium 4, Athlon 64 and Core2. It is clearly seen that while Athlon 64 still outperforms Core2 for x64 instructions, Core2 has much stronger 128-bit ALU units; in particular three XMM logical instructions can run in parallel, which is extremely beneficial for the bitslice implementation. This list was created on the basis of our experiments, since sometimes what Intel's documents say does not agree with our experimental results. For instance, our measurements show that the throughput of `add reg,reg` never reaches 3 on Pentium 4, contrary to Intel's claim. An unknown stall factor must exist in its pipeline. Note that it is common that unexpected things happen on Intel processors. For another simple example, on Core2, a repetition of **Code1A** below runs in 2.0 cycles/iteration as expected, but **Code1B** and **Code1C** run in 2.5 and 3.0 cycles/iteration, respectively. On Athlon64 all the three codes actually work in 2.0 cycles/iteration.

Processor	Pentium4	Athlon64	Core2
Operand Type	64-bit general registers		
`mov reg,[mem]`	4, 1	3, 2	3, 1
`mov reg,reg`	1, 3	1, 3	1, 3
`add reg,reg`	1, 2.88	1, 3	1, 3
`xor/and/or reg,reg`	1, 7/4	1, 3	1, 3
`shr reg,imm`	7, 1	1, 3	1, 2
`shl reg,imm`	1, 7/4	1, 3	1, 2
`ror/rol reg,imm`	7, 1/7	1, 3	1, 1
Operand Type	128-bit XMM registers		
`movaps xmm,[mem]`	−, 1	−, 1	−, 1
`movaps xmm,xmm`	7, 1	2, 1	1, 3
`paddb/w/d xmm,xmm`	2, 1/2	2, 1	1, 2
`paddq xmm,xmm`	5, 2/5	2, 1	1, 1
`xorps/andps/orps xmm,xmm`	2, 1/2	2, 1	1, 3
`psllw/d/q xmm,imm`	2, 2/5	2, 1	2, 1
`pslldq xmm,imm`	4, 2/5	2, 1	2, 1
`punpcklbw/wd/dq xmm,xmm`	2, 1/2	2, 1	4, 1/2
`punpcklqdq xmm,xmm`	3, 1/2	1, 1	1, 1
`pmovmskb reg,xmm`	−, 1/2	−, 1	−, 1

Table 2. A list of an instruction latency and throughput.

```
and  rax,rax        and  rax,rdx        and  rax,rax
and  rbx,rbx        and  rbx,rsi        and  rbx,rax
and  rcx,rcx        and  rcx,rdi        and  rcx,rax
and  rdx,rdx        and  rdx,rax        and  rdx,rax
and  rsi,rsi        and  rsi,rbx        and  rsi,rax
and  rdi,rdi        and  rdi,rcx        and  rdi,rax

Code1A: 2.0 cycles   Code1B: 2.5 cycles   Code1C: 3.0 cycles
```

One of the block cipher algorithms that can have the biggest benefit of Core2 is 128-bit block cipher Serpent[2]. Serpent was designed in a 32-bit bitslice style; specifically, it internally applies 32 lookup tables with 4-bit input/output in parallel in a single round, which can be coded with 32-bit logical and shift instructions only. Table 3 demonstrates that our four-block parallel encryption code using XMM instructions dramatically improves its performance on Core2 as compared with a highly optimized single block encryption program written by Gladman[10]. Serpent was known as a block cipher with a high security margin and a low encryption speed but our result shows that Serpent will be categorized into fast ciphers on future processors.

Processor	Pentium 4		Athlon 64		Core2	
Style	4-Parallel	Single [10]	4-Parallel	Single [10]	4-Parallel	Single [10]
Cycles/block	681	689	466	569	**243**	749
Cycles/byte	42.6	43.1	29.1	35.6	**15.2**	46.8
Instrs/cycle	0.71	1.98	1.03	2.40	**1.98**	1.83

Table 3. Performance of Serpent in single-block and four-block parallel modes.

3 KASUMI

KASUMI [1] is a 64-bit block cipher with 128-bit key that forms the heart of UMTS confidentiality algorithm f8 and integrity algorithm f9. KASUMI has been also adopted as one of GSM standard ciphers for confidentiality. KASUMI was designed on the basis of MISTY1 block cipher with 64-bit block and 128-bit key [13], which has been included in the ISO-18033 standard [12]. Since these ciphers highly focus on hardware platforms, we can naturally expect that they achieve high performance when implemented in a bitslice style. In this section, we start with discussing an optimization of a single block encryption for comparison, and then move to the bitslice implementation

3.1 KASUMI and MISTY1

Both of KASUMI and MISTY1 have an eight-round Feistel structure, whose round function is called FO function, and additionally a small component called FL function is inserted several times outside the FO functions. The FO function itself has a ladder structure with three inner rounds, each of which is called FI function. Therefore these ciphers have a total of 24 FI functions, which dominate their encryption performance.

The left side of figure 2 shows the detailed structure of the FI function of KASUMI. The FI has again a ladder structure with two lookup tables S7 and S9, which are internally applied two times each. Unlike KASUMI, the FI of MISTY1 has only three rounds (S9 - S7 - S9) with slightly different S7 and S9. S7 and S9 (for both of KASUMI and MISTY1) are linearly equivalent to a power function over Galois field $GF(2^7)$ and $GF(2^9)$, and their algebraic degree is 3 and 2, respectively. These low degree tables significantly contribute to small hardware in real applications.

The key scheduling part of KASUMI is extremely simple, consisting of 16-bit rotate shifts by a constant size and xor operations with a constant value only, which is compactly implemented in hardware. Also the key scheduling part of MISTY1 is not costly, consisting of eight parallel FI functions. For more details, see [1] and [13].

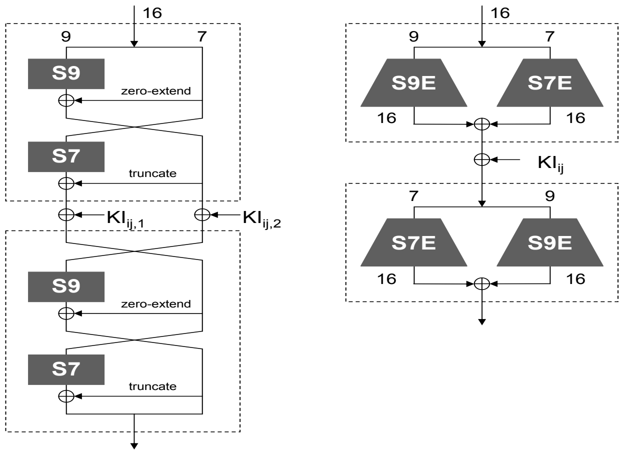

Fig. 2. Equivalent forms of the FI function of KASUMI.

3.2 Single Block Implementation

First we show our implementation of KASUMI in a usual single block encryption style. As stated above, the complexity of the FI function dominates the entire performance of the KASUMI algorithm. A straightforward implementation of the FI on Core2 (or any other PC processors) requires approximately 16 instructions. However by preparing the following two new tables S7E and S9E, we can create a simpler form that is equivalent to the FI function as shown in the right side of figure 2.

```
S9E[x] = ((S9[x]<<9)^S9[x]) & 0xffff              ; 9-bit -> 16-bit
S7E[x] = ((S7[(x&0x7f)]^(x&0x7f))<<9) ^ (x&0x7f)   ; 8-bit -> 16-bit
```

Use of S7E and S9E reduces the number of instructions of the FI function down to 10. **Code2** shows the specific 10-line implementation. Note that S7E must accept an eight-bit input (and ignore its highest bit), which results in a saving of one instruction at the beginning of the code. Since an output of S7E and S9E is stored in a 32-bit entry in practice, a total size of the new tables is $2^8 \times 4$ bytes (S7E) + $2^9 \times 4$ bytes (S9E) = 3072 bytes.

```
movzx  esi,al              ; extract right 8 bits
shr    eax,7               ; extract left  9 bits
mov    eax,S9E[rax*4]
xor    eax,S7E[rsi*4]
xor    eax,[key]           ; xor subkey
mov    esi,eax
shr    esi,9               ; extract left  7 bits
and    eax,01ffh           ; extract right 9 bits
mov    eax,S9E[rax*4]
xor    eax,S7E[rsi*4]
```

`Code2: An optimized code of the FI function of KASUMI.`

Note that in an x64 environment we can equivalently use 64-bit registers instead of 32-bit registers, say, `shr rsi,9` instead of `shr esi,9`, but this should be avoided in general because use of a 64-bit general register as a data register makes an instruction length longer. Also since two adjacent FI functions are mutually independent even if they are not contained in the same FO function, interleaving two FI functions contributes to further speeding-up.

As a result, our optimized codes for the full KASUMI and MISTY1 can run at the speed of 290 cycles/block and 214 cycles/block, respectively. The difference in performance comes from the fact that the former applies $4 \times 3 \times 8 = 96$ table lookups and the latter does $3 \times 3 \times 8 = 72$. The key scheduling part of KASUMI, instead, works of course much faster than that of MISTY1.

3.3 Bitslice implementation

In this subsection we deal with an implementation of KASUMI and MISTY1 in the bitslice mode, that is, 128-block parallel encryption, fully utilizing 16 128-bit XMM registers of the Core2 processor. The performance of bitsliced KASUMI and MISTY1 is largely determined by the number of instructions of lookup tables S7 and S9. Below is our (hand-optimized) bit-level logic of S7 and S9 at the time of writing, where output bits y_i are computed sequentially in our code. Boldface terms, which always appear pairwisely (or more), are stored into registers in advance in order to reduce the number of instructions.

MISTY S9:

$y_0 = x_0(x_4+x_5) + x_1(x_5+x_6) + x_2(x_6+x_7) + x_3(x_7+x_8) + \mathbf{x_4x_8} + 1$

$y_1 = x_3(1+x_2+\mathbf{x_1}+\mathbf{x_4}+\mathbf{x_8}) + x_0(\mathbf{x_2}+\mathbf{x_6}+\mathbf{x_8}) + x_5(\mathbf{x_4}+\mathbf{x_8}) + x_2x_6 + x_7 + 1$

$y_2 = x_4(1+x_3+\mathbf{x_0}+\mathbf{x_2}+\mathbf{x_5}) + x_1(x_0+x_3+x_7) + x_6(\mathbf{x_0}+\mathbf{x_5}) + x_3x_7 + x_8$

$y_3 = x_5(1+x_4+\mathbf{x_1}+\mathbf{x_3}+\mathbf{x_6}) + x_2(\mathbf{x_1}+\mathbf{x_4}+\mathbf{x_8}) + x_7(\mathbf{x_1}+\mathbf{x_6}) + x_0 + \mathbf{x_4x_8}$

$y_4 = x_6(1+x_5+\mathbf{x_2}+\mathbf{x_4}+\mathbf{x_7}) + x_3(\mathbf{x_0}+\mathbf{x_2}+\mathbf{x_5}) + x_8(\mathbf{x_2}+\mathbf{x_7}) + x_1 + x_0x_5$
$y_5 = x_7(1+x_6+\mathbf{x_3}+\mathbf{x_5}+\mathbf{x_8}) + x_4(\mathbf{x_1}+\mathbf{x_3}+\mathbf{x_6}) + x_0(\mathbf{x_3}+\mathbf{x_8}) + x_2 + x_1x_6$
$y_6 = x_8(1+x_7+\mathbf{x_0}+\mathbf{x_4}+\mathbf{x_6}) + x_5(\mathbf{x_2}+\mathbf{x_4}+\mathbf{x_7}) + x_1(\mathbf{x_0}+\mathbf{x_4}) + x_3 + x_2x_7 + 1$
$y_7 = x_1(1+x_0+\mathbf{x_2}+\mathbf{x_6}+\mathbf{x_8}) + x_7(\mathbf{x_0}+\mathbf{x_4}+\mathbf{x_6}) + x_3(\mathbf{x_2}+\mathbf{x_6}) + x_0x_4 + x_5 + 1$
$y_8 = x_0(1+x_1+x_5+x_7+x_8) + x_6(\mathbf{x_3}+\mathbf{x_5}+\mathbf{x_8}) + x_2(x_1+x_5) + x_4 + x_3x_8 + 1$

MISTY S7:
$y_0 = x_0 + \mathbf{x_0x_3x_4} + x_1(x_3+\mathbf{x_0x_6}) + x_2(x_0x_5+x_6) + x_5(x_4+\mathbf{x_3x_6}) + \mathbf{x_5}(\mathbf{x_1}+\mathbf{x_0x_6}) + 1$
$y_1 = x_2(x_0+\mathbf{x_4x_5}) + \overline{x_0}x_6 + \overline{x_2}\mathbf{x_3x_6} + x_4(x_0+x_3+x_1x_6) + \mathbf{x_5}(\mathbf{x_1}+\mathbf{x_0x_6}) + 1$
$y_2 = x_2(x_1+\mathbf{x_0x_3}) + x_4 + x_0((x_1+x_5)x_4+x_5) + x_4(x_2x_6+x_1) + x_3(\mathbf{x_4x_5}+x_6) + \mathbf{x_6}(\mathbf{x_0x_3}+\mathbf{x_4}+\mathbf{x_1})$
$y_3 = (\mathbf{x_0}+\mathbf{x_1}+\mathbf{x_0}(\mathbf{x_1x_2}+\mathbf{x_3})) + x_6(x_2+x_5+\mathbf{x_1x_3}) + x_4(x_2+\mathbf{x_0x_6}) + x_1\mathbf{x_4x_5} + 1$
$y_4 = x_4(x_0+\mathbf{x_1x_3}) + x_5 + \overline{x_1}x_2x_5 + x_3(x_2+\mathbf{x_0x_5}) + x_6((x_1+x_4)x_5+x_1) + 1$
$y_5 = (\mathbf{x_0}+\mathbf{x_1}+\mathbf{x_0}(\mathbf{x_1x_2}+\mathbf{x_3})) + x_2 + \mathbf{x_1x_2x_3} + x_4(x_1+x_0x_2) + x_0(x_1x_5+x_6) + \mathbf{x_5}(\mathbf{x_0}+\mathbf{x_3}+\mathbf{x_2x_6})$
$y_6 = \overline{x_0}x_3 + x_2(x_3x_4+x_5) + x_1(x_0+x_3x_5) + \mathbf{x_1x_2x_6} + \mathbf{x_6}(\mathbf{x_0x_3}+\mathbf{x_4}+\mathbf{x_1}) + \mathbf{x_5}(\mathbf{x_0}+\mathbf{x_3}+\mathbf{x_2x_6})$

KASUMI S9
$y_0 = x_7(x_0+x_1+x_2+x_8) + x_5(\mathbf{x_2}+\mathbf{x_6}+\mathbf{x_8}) + x_4x_8 + x_0x_2 + x_3 + 1$
$y_1 = x_1(1+x_0+x_4+x_7) + x_5(x_0+\mathbf{x_3}+\mathbf{x_8}) + x_2(x_3+x_7) + x_0x_4 + x_6 + 1$
$y_2 = x_6(x_2+x_3+\mathbf{x_5}+\mathbf{x_7}) + x_0(x_5+\mathbf{x_3}+\mathbf{x_8}) + x_7(x_4+x_5) + x_3x_4 + x_1 + 1$
$y_3 = x_0(1+x_6+\mathbf{x_3}+\mathbf{x_8}) + x_1(\mathbf{x_2}+\mathbf{x_6}+\mathbf{x_8}) + x_4(x_2+x_7) + x_7x_8 + x_5$
$y_4 = x_0(\mathbf{x_1}+\mathbf{x_5}+\mathbf{x_7}) + x_3(x_1+x_6+x_8) + x_8(x_1+x_2) + x_6x_7 + x_4$
$y_5 = x_6(x_0+x_8+\mathbf{x_1}+\mathbf{x_7}) + x_4(\mathbf{x_1}+\mathbf{x_5}+\mathbf{x_7}) + x_7(\mathbf{x_3}+\mathbf{x_8}) + x_5x_8 + x_2 + 1$
$y_6 = x_5(x_1+x_4+\mathbf{x_2}+\mathbf{x_6}+\mathbf{x_8}) + x_3(\mathbf{x_2}+\mathbf{x_6}+\mathbf{x_8}) + x_8(\mathbf{x_1}+\mathbf{x_7}) + x_4x_6 + x_0 + x_7$
$y_7 = x_2(x_0+\mathbf{x_3}+\mathbf{x_6}+\mathbf{x_1}+\mathbf{x_7}) + x_3(1+x_0+x_6) + x_5(x_4+x_7) + x_0x_1 + x_8 + 1$
$y_8 = x_1(x_0+x_2+x_5+x_6) + x_2(1+x_5+x_8) + x_4(\mathbf{x_3}+\mathbf{x_6}) + x_3x_8 + x_7$

KASUMI S7
$y_0 = x_4(x_0x_1+\mathbf{x_3x_5}+\mathbf{x_2x_6}) + \overline{x_2}x_5 + x_6(1+x_0+x_1+x_3+x_5(x_1+x_4)) + \mathbf{x_1x_3} + x_4$
$y_1 = x_0(x_1+x_4+\mathbf{x_3x_5}+\mathbf{x_2x_6}) + \overline{x_3}x_6 + x_5(1+\mathbf{x_1x_2}) + x_4(x_2+x_5x_6) + 1$
$y_2 = x_0(\overline{x_4}x_3+\overline{x_1}x_6) + x_6(x_2+x_4) + x_5(x_1+\mathbf{x_0x_2}) + x_2(x_3+\mathbf{x_1x_4}) + x_0 + 1$
$y_3 = \overline{x_5}\mathbf{x_1x_4} + \overline{x_1}\mathbf{x_0x_5} + x_6(x_2+\mathbf{x_1x_3}) + x_3(x_2x_5+x_4) + x_1\mathbf{x_0x_2} + x_1$
$y_4 = \overline{x_0}x_3x_6 + \overline{x_0}\mathbf{x_1x_4} + \overline{x_4}\mathbf{x_0x_5} + x_3(1+x_1+x_2x_4) + x_5(x_6+\mathbf{x_1x_3}) + \mathbf{x_0x_2} + x_1x_6 + 1$
$y_5 = x_0(\overline{x_4}x_2+ \overline{x_6}x_3+x_5) + x_5(\overline{x_6}x_2+x_4) + \mathbf{x_1x_2}(x_3+x_6) + x_6(x_1+x_3x_4) + x_2 + 1$
$y_6 = x_6(1+x_1(x_0+x_4)+x_2x_3+\mathbf{x_0x_5}) + x_0(x_4+\mathbf{x_1x_3}) + x_5(x_1+x_3) + \mathbf{x_1x_2}$

	KASUMI		MISTY1	
Lookup tables	S9	S7	S9	S7
Number of instructions	149	153	148	144

Table 4. The number of instructions of S7 and S9.

Processor	Pentium 4		Athlon 64		Core2	
Style	Bitslice	Single	Bitslice	Single	Bitslice	Single
KASUMI						
Cycles/block	241	300	241	272	**74**	290
Cycles/byte	30.1	75.0	30.1	34.0	**9.25**	36.3
Instrs/cycle	0.71	1.69	0.71	1.86	**2.31**	1.75
Cycles/Keysch	8	104	7	64	**2**	78
MISTY1						
Cycles/block	185	234	195	203	**59**	214
Cycles/byte	23.1	29.3	24.4	25.4	**7.38**	26.8
Instrs/cycle	0.72	1.82	0.68	2.10	**2.26**	1.99
Cycles/Keysch	57	244	57	240	**16**	178

Table 5. Performance of our implementation of KASUMI and MISTY1.

Tables 4 and 5 show our implementation results of KASUMI and MISTY1 on Pentium 4, Athlon 64 and Core2. Instructions for S7 and S9 occupy 69% and 61% of the entire code of KASUMI and MISTY1, respectively. It is seen that both ciphers achieve an overwhelming performance; three to four times faster in the bitslice mode on Core2. Also the key scheduling of KASUMI can be carried out almost with no cost due to the nature of its structure.

4 AES and Camellia

4.1 Bitslice Implementation

How to implement bitsliced AES [8] and Camellia [3] on x64 platforms was first reported in [14]. The codes shown in the paper were written not using 128-bit XMM registers but using 64-bit general registers, because XMM instructions had poor performance for bitslicing on its target processors (Pentium 4 and Athlon 64). In fact these processors internally treated a 128-bit instruction as two 64-bit operations. In this subsection, we discuss performance of bitsliced AES and Camellia fully utilizing 128-bit XMM instructions on the Core2 processor.

The dominant part of these bitsliced ciphers is the lookup table S, which is linearly equivalent to an inversion function over $GF(2^8)$. The known smallest hardware design (i.e. most suitable for the bitslice implementation) of S is to use a subfield of index two; that is, we represent an inverse of $GF(2^{2n})$ as a combination of operations on $GF(2^n)$ recursively [6][16][19]. The essence of this technique is to select $(1, a)$ as a basis of $GF(2^{2n})$ over $GF(2^n)$ for an $a \in GF(2^{2n})$ such that $Tr_{GF(2^{2n})/GF(2^n)}(a) = 1$. Then for any $x, y, z, u \in GF(2^n)$, we have

$$(x + ya)(z + ua) = (xz + yuNr_{GF(2^{2n})/GF(2^n)}(a)) + ((x + y)(z + u) + xz)a,$$

which means that a multiplication of $GF(2^{2n})$ can be designed with three multiplications of $GF(2^n)$ like the Karatsuba algorithm.

Using 16 XMM registers, instead of general registers, also reduces "register pressure", which results in a smaller number of instructions of S in software. Our optimized codes for S consist of 201 and 199 instructions for AES and Camellia, respectively, which are 2% smaller than those shown in [14].

Table 6 shows our implementation results of AES and Camellia with 128-bit key in bitslice and non-bitslice modes. "Bs128" and "Bs64" denote the bitslice mode using 128-bit XMM instructions and 64-bit general instructions, respectively. "Single" and "Double" indicate a usual single block encryption and a double-block parallel encryption by interleaving two blocks, respectively. For both algorithms, the obtained encryption speed, 9.2 cycles/block and 8.4 cycles/block on Core2, respectively, is the highest speed ever achieved in a PC platform, where the previous record was 10.6 cycles/block and 10.9 cycles/block on Athlon 64 as shown in [14]. In addition, to our best knowledge, this is the first result where performance of AES in the bitslice mode has exceeded that in an ordinary block-by-block encryption mode.

Processor	Pentium 4			Athlon 64			Core2		
AES									
Style	Bs128	Bs64[14]	Single[14]	Bs128	Bs64[14]	Single[14]	Bs128	Bs64	Single
Cycles/block	491	418	256	560	250	170	**147**	307	232
Cycles/byte	30.7	26.1	16.0	35.0	15.6	10.6	**9.19**	19.2	14.5
Instrs/cycle	0.80	1.66	1.81	0.70	2.75	2.74	**2.66**	2.27	2.00
Camellia									
Style	Bs128	Bs64[14]	Double[14]	Bs128	Bs64[14]	Double[14]	Bs128	Bs64	Double
Cycles/block	467	415	457	510	243	175	**135**	272	208
Cycles/byte	29.2	25.9	28.6	31.9	15.2	10.9	**8.44**	17.0	13.0
Instrs/cycle	0.72	1.61	0.94	0.65	2.74	2.46	**2.47**	2.44	2.07

Table 6. Performance of our implementation of AES and Camellia with 128-bit key.

4.2 Format Conversion

The bitsliced cipher uses a non-standard input/output data format. This is not a problem in a standalone application such as a file encryption utility or a password recovery program. However a format conversion is required if a file encrypted in the bitslice mode must be decrypted in an existing parallel mode of operation such as the CTR mode. This conversion is essentially an entire rearrangement of bit positions, which is generally costly in software, and its performance overhead cannot be ignorable.

This paper for the first time discusses a specific implementation algorithm of data conversion between a bitslice format and an ordinary format. The following piece of code (**Code3**) shows our basic step creating a byte sequence formatted in a bitsliced style pointed by `rdx` from an ordinary byte sequence pointed by `rcx` for a 128-bit block cipher.

```
movaps          xmm0,  0[rcx]
punpck[l|h]bw   xmm0, 16[rcx]   ; xxxxxxxx xxxxxx10
movaps          xmm1, 32[rcx]
punpck[l|h]bw   xmm1, 48[rcx]   ; xxxxxxxx xxxxxx32
movaps          xmm2, 64[rcx]
punpck[l|h]bw   xmm2, 80[rcx]   ; xxxxxxxx xxxxxx54
.....
.....
movaps          xmm7,224[rcx]
punpck[l|h]bw   xmm7,240[rcx]   ; xxxxxxxx xxxxxxFE

punpck[l|h]wd   xmm0,xmm1       ; xxxxxxxx xxxx3210
punpck[l|h]wd   xmm2,xmm3       ; xxxxxxxx xxxx7654
punpck[l|h]wd   xmm4,xmm5       ; xxxxxxxx xxxxBA98
punpck[l|h]wd   xmm6,xmm7       ; xxxxxxxx xxxxFEDC

punpck[l|h]dq   xmm0,xmm2       ; xxxxxxxx 76543210
```

```
24   punpck[l|h]dq   xmm4,xmm6      ; xxxxxxxx FEDCBA98
25
26   punpck[l|h]qdq xmm0,xmm4      ; FEDCBA98 76543210
27
28   pmovmskb        eax,xmm0       ; 16 7-th bits of xmm0
29   mov             112[rdx],ax
30   paddb           xmm0,xmm0
31   pmovmskb        eax,xmm0       ; 16 6-th bits of xmm0
32   mov             96[rdx],ax
33   paddb           xmm0,xmm0
34   pmovmskb        eax,xmm0       ; 16 5-th bits of xmm0
35   mov             80[rdx],ax
36   paddb           xmm0,xmm0
..   .....
..   .....
48   paddb           xmm0,xmm0
49   pmovmskb        eax,xmm0       ; 16 0-th bits of xmm0
50   mov             0[rdx],ax
```

Code3: A format conversion code creating 128 converted bits

The first part (lines 1 to 26) creates a 16-byte data on `xmm0`, whose n-th byte corresponds to a byte in memory at the addresses $16n + m$ ($n = 0, 1, .., 15$). Also m ($m = 0, 1, .., 15$) can be controlled by a choice of "unpack" instructions `punpck[l|h]bw`, `punpck[l|h]wd`, `punpck[l|h]bdq`, `punpck[l|h]dqd` (low `l` or high `h`); specifically, `llll` for $m = 0$, `lllh` for $m = 1$ and `llhl` for $m = 2$, etc. The latter part (lines 28 to 50) creates 16-bit data on `ax` consisting of 16 bits at bit positions $8i+j$ of `xmm0` ($i = 0, 1, ..., 15$) using a special `pmovmskb` instruction, and then it is written into memory, which is repeated 8 times ($j = 0, 1, .., 7$).

Basically the full format conversion of 128 bits $\times$ 128 blocks = 2KB data can be done by repeating **Code3** 128 times (with changing unpack instructions, `rcx` and `rdx`). However by keeping intermediate values in temporary registers for later use, the number of memory reads is significantly reduced. Table 7 demonstrates performance figures of our format conversion code. It is seen that the conversion works very fast, in less than one byte per cycle, which shows that bitsliced AES/Camellia runs still faster than non-bitsliced AES/Camellia on Core2 even if an overhead of data format conversion is included in the bitsliced code.

Processor	Pentium 4	Athlon 64	Core2
Cycles/block	41.5	28.1	15.4
Cycles/byte	2.59	1.76	0.96
Instrs/cycle	0.72	1.06	1.96

Table 7. Measured performance of our format conversion code.

5 Conclusions

This paper explored the state-of-the-art implementation techniques for speeding up block ciphers on Intel's new Core2 microprocessor. We have shown that the bitslicing technique is actually promising on a PC platform from practical points of view. A bitsliced AES code that is fully compatible with the CTR mode can be now faster than a non-bitsliced AES code on Core2. Another importance of the bitslice mode is that a bitsliced code is secure against cache timing attacks since it does not use any lookup tables whose address is dependent on secret information. We believe that bitsliced ciphers will be much more widely used in real applications in very near future.

References

[1] 3GPP TS 35.202 v6.1.0, "3G Security; Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2:KASUMI Specification (Release 6)", 3rd Generation Partnership Project, 2005.

[2] R. Anderson, E. Biham, L. Knudsen: "Serpent: A proposal for the Advanced Encryption Standard", Available at `http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/serpent.pdf`

[3] 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.

[4] R. Bhaskar, P. Dubey, V. Kumar, A. Rudra: "Efficient galois field arithmetic on SIMD architectures", Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, pp.256-257, ACM Press, 2003.

[5] 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.

[6] D. Canright: "A Very Compact S-Box for AES", Proceedings of CHES 2005, Lecture Notes in Computer Science, Vol.3659, pp.441-455, Springer-Verlag, 2005.

[7] The distributed.net project: Available at `http://www.distributed.net/index.php.en`

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

[9] A. Fog: "Software optimization resources", Available at `http://www.agner.org/optimize/`

[10] B. Gladman: "Serpent Performance", Available at `http://fp.gladman.plus.com/cryptography_technology/serpent/`

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

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

[13] M.Matsui, "New encryption algorithm MISTY," Proceedings of Fast Software Workshop FSE'97 , Lecture Notes in Computer Science, Vol.1267, pp.54-68, Springer-Verlag, 1997.

[14] M. Matsui: "How Far Can We Go on the x64 Processors?", Proceedings of Fast Software Workshop FSE2006, Lecture Notes in Computer Science, Vol.4047, pp.341-358, Springer-Verlag, 2006.

[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] N. Mentens, L. Batina, B. Preneel, I. Verbauwhede: "A Systematic Evaluation of Compact Hardware Implementations for the Rijndael S-Box", Proceedings of CT-RSA 2005, Lecture Notes in Computer Science, Vol.3376, pp.323-333, Springer-Verlag, 2005.

[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] T. Shimoyama, S. Amada, S. Moriai: "Improved fast software implementation of block ciphers," Proceedings of the First International Conference on Information and Communication Security, pp.269-273, Springer-Verlag, 1997.