A Fast RSA Implementation on Itanium 2 Processor

Kazuyoshi Furukawa, Masahiko Takenaka, and Kouichi Itoh

FUJITSU LABORATORIES LTD.,
4-1-1, Kamikodanaka, Nakahara-ku, Kawasaki, 211-8588, Japan
{kazful, takenaka, kito}@labs.fujitsu.com

Abstract. We show the fastest implementation result of RSA on Itanium 2. For realizing the fast implementation, we improved the implementation algorithm of Montgomery multiplication proposed by Itoh et al. By using our implementation algorithm, pilepine delay is decreased than previous one on Itanium 2. And we implemented this algorithm with highly optimized for parallel processing. Our code can execute 4 instructions per cycle (At maximum, 6 instructions are executed per cycle on Itanium 2), and its probability of pipeline stalling is just only 5%. Our RSA implementation using this code performs 32 times per second of 4096-bit RSA decryption with CRT on Itanium 2 at 900MHz. As a result, our implementation of RSA is the fastest on Itanium2. This is 3.1 times faster than IPP, a software library developed by Intel, in the best case.

Keywords: RSA, Montgomery multiplication, software implementation, Itanium 2.

1 Introduction

The RSA [13] is one of the standard public-key cryptosystems. The security of RSA relies on a fact that factoring huge integers, which is used as a public-key in RSA, is infeasible. Thus the key-length of RSA is chosen so as to avoid such factorization. In the past, 1024-bit was enough. However, with a remarkable development of semiconductor technologies, we need longer RSA keys in the near future. For example, NIST recommends using 2048-bit or 3072-bit RSA keys after the year 2010 [16] [17]. If the key-length of RSA becomes longer, its computational cost grows with the cube of the bit length. Therefore, realizing a high-speed RSA with longer-keys is more important than ever.

Most of the RSA processing time is spent in modular multiplications. For performing modular multiplication effectively, several types of primitive algorithms was proposed by Montgomery [1], Barrett [14], Kaihara-Takagi [15] and so on. Currently, the most popular algorithm is the Montgomery's one (Montgomery multiplication). In addition, many improvements on this algorithm have been proposed. Dusse and Kaliski transformed a multiplication and reduction for long bit integers into an effective integration of multiplication and reduction for small bit integers [18]. Related with hardware architecture for scalable Montgomery multiplication, many results are known [7] [8] [9] [10] [11] [12]. Related with software implementation, Koc et al. [2] presented several effective software implementation algorithms of the Montgomery multiplication (e.g. SOS, CIOS, FIOS and more), and evaluated the required resources of these implementation algorithms. Furthermore, FIOS, one of the improvement by Koc et al. [2], was improved by Itoh et al [3]. This improved algorithm (Itoh's algorithm) is very suitable for pipelining, and allows high-performance RSA implementation on a DSP [3].

Our objective of this paper is to realize a fast implementation of RSA on Itanium 2, because it has attractive features for establishing high-performance RSA. That is, it can operate a 64-bit $\times$ 64-bit multiply instruction with very low latency (4 cycles) and low delay (1 cycle), and can execute 6 instructions in parallel at 1 cycle. Our strategy has 2 steps as follows.

In the first step, we analyzed the dependency between data calculations in a Montgomery multiplication in the Itoh's algorithm. By considering the specific conditions for Itanium 2 pipeline scheduling, the pipeline delays are evaluated. However, we found that a naive application of the Itoh's algorithm causes heavy overheads on Itanium 2. Thus we enhanced the Itoh's algorithm so as to avoid such overheads, which is one of our contributions of this paper.

In the second step, based on the pipeline scheduling found in the first step, we established an optimized code in parallel processing in assembly language of Itanium 2 by trial and errors. Our code can execute 4 instructions per cycle, while Itanium 2 can execute at maximum 6 instructions. Since the probability of pipeline stalling of our code is just only 5%, we think our code is optimal. These results show the high-performance in parallel processing for software implementation. In fact, our code performs 32 times 4096-bit RSA decryptions with CRT per second on Itanium 2 at 900MHz. Compared with IPP, an RSA software library developed by Intel, our code is 3.1 times faster in the best case.

As far as the authors know, this is the first paper which analyzes the optimizing process and presents performance results for RSA software implementation specified for Itanium 2.

The rest of this paper is organized as follows. We describe primitives of the Montgomery multiplication and previous implementation algorithms in chapter 2, our proposed algorithm in chapter 3, implementational results of our proposed algorithm in chapter 4 and a conclusion in chapter 5.

2 Montgomery Multiplication and Itoh's Algorithm

In this chapter, we briefly introduce the Montgomery multiplication method for modular multiplications and its implementation algorithm.

The Montgomery method allows efficient modular multiplications [1]. The most crucial part of this method is the Montgomery multiplication (REDC) shown in Algorithm 1. Let N be an integer is greater than 1, and R be an integer greater than N and relatively prime to N. Also let N' be an integer such that $0 < N' < R$ and $N' = -N^{-1} \pmod{R}$. Under these notations, Algorithm 1 calculates a Montgomery multiplication $REDC(A, B) = A \times B \times R^{-1} \pmod{N}$ for integers A and B with $0 \leq A \times B \leq R \times N$.

To compute a modular multiplication $A \times B \pmod{N}$ with the Montgomery multiplication, we covert A and B to so-called the Montgomery domain in which the Montgomery multiplications are effectively computed (this conversion is done by applying an appropriate constant R). After a Montgomery multiplication, a result is re-converted to the previous domain and output. An outline is shown in Algorithm 2.

```
-input: $A, B, R, N.$
-output: $REDC(A,B) = A \times B \times R^{-1} \pmod N$
-algorithm
    $N' := -N^{-1} \pmod R$
    $Y := AB$
    $M := (Y \pmod R) \times N' \pmod R$
    $Y := Y + MN$
    $Y := Y/R$
    $if\ Y \geq N\ then\ Y := Y - N$
    $return\ Y$
```

Alg. 1. Montgomery Multiplication (REDC)

```
-input: $A, B, R, N.$
-output: $A \times B \pmod N$
-algorithm
    $A' = A \times R \pmod N$
    $B' = B \times R \pmod N$
    $C' = REDC(A', B') = A \times B \times R \times R \times R^{-1} \pmod N = A \times B \times R \pmod N$
    $C = REDC(C', 1) = A \times B \times R \times R^{-1} \pmod N = A \times B \pmod N$
    $return\ C$
```

Alg. 2. Structure of "CORE loop of REDC" in Algorithm 3

In typical implementations of REDC, input integers A, B and output integer Y are represented with multi-precision, namely an $s \times w$-bit integer A is represented as $(a_{s-1}, a_{s-2}, \ldots, a_0)$ where every $a_i (0 \leq i \leq s-1)$ is a w-bit word data. (Here, $R = 2^{s \times w}$ is used). So, in these implementations, all integers are represented with multi-precision and looped calculations for every word data are required. Thus such implementations are not efficient.

In 1996, Koc et al. presented some implementation algorithms of REDC [2]. FIOS (Finely Integrated Operand Scanning) algorithm was among these implementations and further improved by Itoh et al [3] so as to suitable for pipelining [3]. We focus on this algorithm (Itoh's algorithm) in the rest of this paper. An outline of the Itoh's algorithm is shown in Algorithm 3. And "Core loop of REDC" structure in Algorithm 3 is shown in Fig 1. Here, "Upper Computation" in Fig 1 is corresponding to the first equation of "Core loop of REDC" in Algorithm 3. And "Lower Computation" is corresponding to the second equation. To handle carries of each equation to the next loop, the first equation in the $(i + 1)$-th loop does not refer to a result of the second equation in the i-th loop. Thus Itoh's algorithm can compute the first and second equations in parallel, which is an improvement in comparison with the original FIOS algorithm. This is why the Itoh's algorithm is suitable for pipelining.

3 Enhancement on the Itoh's Algorithm

Our objective of this paper is to realize a fast implementation of RSA on Itanium 2. To do so, our strategy has 2 steps. As the first step, we analyze the dependency between data

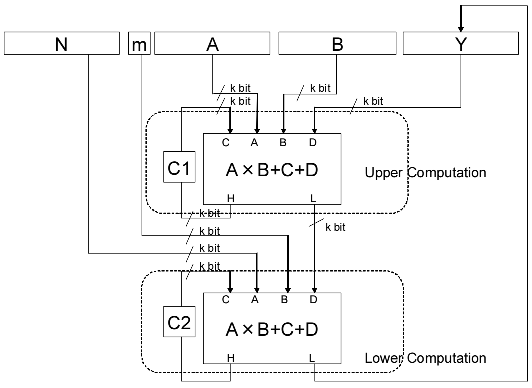

Fig. 1. Structure of "CORE loop of REDC" in Algorithm 3

calculations in the Itoh's algorithm in this chapter. Also, an enhanced Itoh's algorithm, which avoids heavy overheads on Itanium 2 are proposed in this chapter.

3.1 Analysis of "Core Loop of REDC" in the Itoh's Algorithm

For improving the Itoh's algorithm, analyzing "Core loop of REDC" part in Algorithm 1 is required. Figure 2 shows our dependency analysis of "Core loop of REDC" in which the i-th and the $(i+1)$-th loops are discussed. Here "Upper *MAA*" corresponds to "Upper Computation" in Fig. 1 and "Lower *MAA*" to "Lower Computation" in Fig. 1 (*MAA* means one multiplication and two additions).

In Fig.2, "Lower *MAA*" needs a temporal result of "Upper *MAA*" before starting its computation. We call this relation "dependency" and represent as directions in Fig.2. Similarly, three following dependencies can be found Fig.2. Here a representation $X \rightarrow Y$ means that Y needs a result of X to start its computation. We call this relation as "Y depends to X".

(1-i) Upper *MAA* $\rightarrow$ Lower *MAA*
(1-ii) i-th Upper *MAA* $\rightarrow$ $(i + 1)$-th Upper *MAA*
(1-iii) i-th Lower *MAA* $\rightarrow$ $(i + 1)$-th Lower *MAA*

REDC implementations with FIOS cannot avoid a dependency between the i-th Lower *MAA* and the $(i + 1)$-th Upper *MAA*. On the other hand, the Itoh's algorithm computes i-th Lower *MAA* and $(i + 1)$-th Upper *MAA* in parallel, because there is no dependency between them. Thus, the Itoh's algorithm is very suitable for pipelining.

For implementing the Itoh's algorithm on Itanium 2, some specific conditions for Itanium 2 pipeline scheduling shold be considered. As the first condition, Itanium 2 cannot operate one multiplication and two additions (multiply-add-add) within 1 instruction, but can operate one multiplication and one addtion (multiply-add) within 1 instruction. By taking this restriction into account, a pipeline scheduling of the Itoh's algrithm is modified as in Fig.3, where a symbol *MA* represents a multiply-add operation and a symbol A' represents an add operation. Note that *MA* can be operated within 1 instruction, while A' cannot within 1 instruction.

```
-input: $A, B, R, N$.
-output: $REDC(A,B) = A \times B \times R^{-1} \pmod N$
-Multi-precision integers
$A=(a_{s-1}, \ldots, a_0)$
$B=(b_{s-1}, \ldots, b_0)$
$N=(n_{s-1}, \ldots, n_0)$
$Y=(y_s, \ldots, y_0)$
$N' = (n'_{s-1}, \ldots, n'_0)$
-w-bit word data.
$tmp, C1, C2, m$

- algorithm
  $Y := 0$
  for $j=0$ to $s-1$
    $(C1, tmp) := y_0 + a_0 \times b_j$
    $m := tmp \times n'_0 \pmod r$          /* $r$ is $2^w$ */
    $(C2, tmp) := tmp + m \times n_0$
      for $i=0$ to $s-1$
        $(C1, tmp) := y_i + C1 + a_i \times b_j$        } Core loop of REDC
        $(C2, y_{i-1}) := tmp + C2 + m \times n_i$
    next $i$
    $(C2, C1) := C1 + C2 + y_s$
    $y_{s-1} := C1$
    $y_s := C2$
  next $j$
if $Y >= N$ then $Y := Y - N$
return $Y$
```

Alg. 3. Itoh's algorithm for REDC

In the pipeline scheduling of Fig.3, there are three dependencies shown in the following (2-i)–(2-iii).

(2-i) Upper A' $\rightarrow$ Lower MA
(2-ii) i-th Upper MA $\rightarrow$ $(i + 1)$-th Upper MA
(2-iii) i-th Lower A $\rightarrow$ $(i + 1)$-th Lower A'

In Fig.3, MA can be operated within 1 instruction (XMA instruction of Itanium 2). However, there is another specific condition of Itanium 2 that XMA instructions are executed only to floating-point registers for input and output data. Since Upper A' (integer addition) is executed to general registers, not to floating-point registers, we require a trick like follows:

(a1) One idea is to operate an Upper A' by an XMA instruction. This can be done by substituting integers in general registers to floating-point registers and by executing an XMA instruction with a dummy multiplication (namely, a multiplication with 1). In total, two XMA instructions are required, which cause overheads because XMA instructions are heavy operations compared to integer addition instructions to general registers.
(a2) Another idea is to operate an Upper A' as integer addition. This can be done by converting from general registers to floating-point registers before an XMA instruction, and by converting floating-point registers to general registers after the instruction. Again, this idea causes performance overheads due to two data-conversion instructions, which are much heavier than XMA instructions on Itanium 2.

In the next section, we enhance the Itoh's algorithm and propose an efficient algorithm for "Core loop of REDC" based on (a2) rather than (a1). This is because integer additions are very light instructions on Itanium 2.

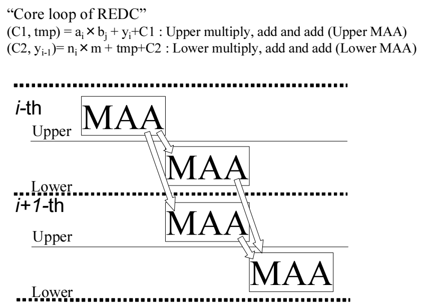
"Core loop of REDC"
(C1, tmp) = a_i × b_j + y_i+C1 : Upper multiply, add and add (Upper MAA)
(C2, y_{i-1})= n_i × m + tmp+C2 : Lower multiply, add and add (Lower MAA)
i-th
Upper
Lower
i+1-th
Upper
Lower
MAA
MAA
MAA
MAA

Fig. 2. Pipeline scheduling of "Core loop of REDC" in the Itoh's algorithm (not optimized to Itanium 2)

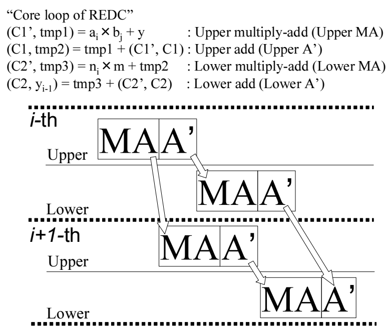
"Core loop of REDC"
(C1', tmp1) = a_i × b_j + y : Upper multiply-add (Upper MA)
(C1, tmp2) = tmp1 + (C1', C1) : Upper add (Upper A')
(C2', tmp3) = n_i × m + tmp2 : Lower multiply-add (Lower MA)
(C2, y_{i-1}) = tmp3 + (C2', C2) : Lower add (Lower A')
i-th
Upper
Lower
i+1-th
Upper
Lower
MA A'
MA A'
MA A'
MA A'

Fig. 3. Pipeline scheduling of "Core loop of REDC" in the Itoh's algorithm (optimized to Itanium 2)

3.2 Proposed Algorithm

As described in the previous section, two approaches (a1) and (a2) have heavy overheads. A main reason of these overheads is a pipeline delay of Lower *MA* by the dependency (2-i). Especially in (a2), a pipeline delay caused by Upper A' is large enough, because it consists of 1 conversion instruction (floating-point registers to general registers), 1 addition instruction for general registers and another 1 conversion instruction (general registers to floating-point registers).

Our approach for this problem is to break the dependency between Upper A' and Lower *MA*. Since this dependency is a result of the Itoh's algorithm, we go back to the Itoh's algorithm and enhance it so as to break the dependency. In fact, we established an enhanced version of the Itoh's algorithm in Fig.4 and Fig.5, in which Lower *MA* does not depend on Upper A' anymore but Lower *MA* depends on that of the previous loop. By this enhancement, overheads by the pipeline delay are eliminated.

In Fig.4, two symbols *MA* and A'' are used, where a symbol *MA* is same as in Fig.3 while a symbol A'' represents an addition of 4 values, the carry (C3), data stored on

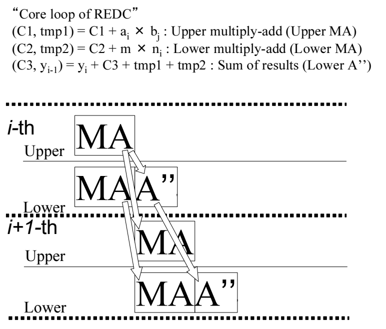
"Core loop of REDC"
(C1, tmp1) = C1 + a_i × b_j : Upper multiply-add (Upper MA)
(C2, tmp2) = C2 + m × n_i : Lower multiply-add (Lower MA)
(C3, y_{i-1}) = y_i + C3 + tmp1 + tmp2 : Sum of results (Lower A'')
i-th
Upper
MA
MA
A''
Lower
i+1-th
Upper
MA
MA
A''
Lower

Fig. 4. Pipeline scheduling of "Core loop of REDC" in our enhanced Itoh's algorithm (optimized to Itanium 2)

memory (yi) and calculated results by Upper and Lower *MA* (tmp1 and tmp2). The dependencies of Fig.4 are shown in the following (3-i)–(3-iv).

(3-i) i-th Upper $MA \rightarrow i$-th Lower A''
(3-ii) i-th Upper $MA \rightarrow (i + 1)$-th Upper MA
(3-iii) i-th Lower $MA \rightarrow (i + 1)$-th Lower MA
(3-iv) i-th Lower $A'' \rightarrow (i + 1)$-th Lower A''

By changing the dependency (2-i) to the dependency (3-iii), we succeeded to cancel the pipeline delay by Upper A' in Fig.3.

4 Implementation of Our New Algorithm on Itanium 2

In this chapter, as the second step for realizing fast RSA implementation, we describe implementational aspects of our new algorithm proposed in section 3.2 including the experimental performance results of RSA decryptions on Itanium 2.

4.1 Characteristics of Itanium 2

Itanium 2 belongs to a processor family called IPF (Itanium Processor Family), which is developed by Intel and Hewlett Packard, and its architecture is based on that of IA-64. The greatest characteristic of IPF is the EPIC (Explicitly Parallel Instruction Computing) technology, which does not support out-of-order executions unlike IA-32 architecture processors. In the out-of-order executions, instruction scheduling was dynamically done by a processor. But in IPF, instruction scheduling is done by the compiler. So the effectiveness of the compiler is directly reflected to the performance of software implementation. Other characteristics of Itanium 2 are listed in followings:

- Executes 6 instructions within 1 cycle at maximum. In other words, maximum IPC (Instruction Per Cycle) is 6.
- Provides many ports for executing various types of instructions (4 ports for memory, 2 ports for general, 2 ports for floating-point and 3 ports for blanch)

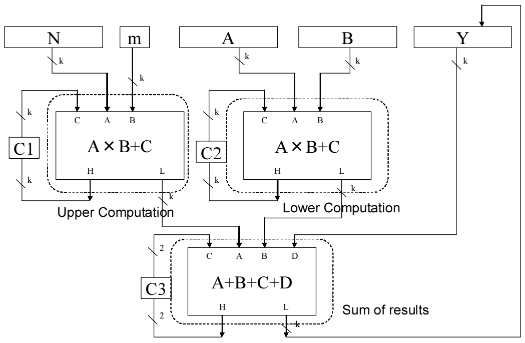

Fig. 5. New structure of "CORE loop of REDC" in our enhanced Itoh's algorithm

- Has 128 of 64-bit general registers.
- Has 128 of 82-bit floating-point registers.
- Provides software pipelining by CPU (MSL, Modulo Schedule Loop).
- Provides instructions for 64-bit fixed-point multiply-add instructions with 4 cycles latency and 1 cycle delay.

4.2 Implementation Environment

We used an hp workstation zx 2000 for implementation, whose specifications are summarized in the followings:

- CPU and frequency: Itanium 2 at 900 MHz
- Size of DRAM : 2 Gigabytes of memory
- Size of L1 cache: 16 K byte (Instruction) / 16K byte (Data)
- Size of L2 cache: 256 K byte
- Size of L3 cache: 1.5 M byte
- OS: Red Hat Enterprise Linux 4
- Compiler: gcc 3.4 and icc 9.0

In the following sections, we implemented REDC and RSA based on our enhanced Itoh's algorithm described in the previous sections. At the optimization, the performance of 1-time calculation of 4096-bit REDC is mainly considered. Our analys is used three factors, namely the number of total instructions, the probability of pipeline stalling and averaged IPC of total instructions. These factors are measured by the "performance monitoring counter" provided for Itanium 2. In this counter, the number of total instructions does not include NOP instructions, and averaged IPC of total instructions are obtained with dividing the number of total instructions by total cycles. We did not consider cache hit-miss here because its occasion is hard to be monitored.

4.3 Implementation and Optimization of REDC

In this section, we describe implementation aspects of REDC with C language and assembly language.

Implementation with C language. At the first step of the optimization, we implemented our new algorithm with C language. Here, instructions to operate multiply-add (XMA instruction) cannot be directly used with C language. So we used intrinsic function of Intel C Compiler 9.0 (icc 9.0) to operate XMA instruction. Grammar of the intrinsic function is represented as follows:

```
__int64 _m64_xmalu(__int64 a, __int64 b, __int64 c)
__int64 _m64_xmahu(__int64 a, __int64 b, __int64 c)
```

We show the evaluation result for optimization of our C language code in table 2. We obtained probability of pipeline stalling comes up to 14, most of which are waiting cycles to execute XMA instrutions. We found we can eliminate some instructions of assemble codes output by icc 9.0. An example of this elimination is shown in Table 1 which shows the assemble code of `D=_m64_xmalu(A,B,C);` by icc 9.0 and its eliminated result by hand-assembled code. We note `grA`, `grB` and `grC` represent the interger registers and `frA`, `frB` and `frC` represents the floating-point registers.

Table 1. Assembly list output by icc 9.0 (left column) and result of elimination by hand-assembled code (right column)

`ld8 grA = [&A];`	`ldf8 frA=[&A];`
`ld8 grB = [&B];`	`ldf8 frB=[&B];`
`ld8 grC = [&C];`	`ldf8 frC=[&C];`
`setf.sig.frA = grA;`	`xma.lu.frD=frA,frB,frC;`
`setf.sig.frB = grB;`	`stf8[&D] = frD;`
`setf.sig.frC = grC;`	
`xma.lu.frD=frA,frB,frC;`	
`getf.sig.grD=frD;`	
`st8[&D]=grD;`	

Implementation with hand-assembled code. In section 4.3.1, we implemented our new REDC algorithm with C language and found out we can eliminate some instructions. So we optimized the code with extreme technique of hand-assembling by eliminating the total number of instructions, tuning the software pipeline schedule based on the instruction latencies. Finally, we attained the result that average IPC of REDC is 4.02. Especially, average IPC in the "Core loop of REDC" of is 5.25 which is extremely good result because maximum IPC of Itanium 2 is 6. In table 2, we show the result of the evaluation for optimization of the hand-assembled code with comparison that of C language for 4096-bit our REDC algorithm.

4.4 Implementation of RSA

We measured the performance of RSA decryption with CRT by using the hand-assembled optimized code of our new REDC algorithm described in section 4.3.2. At the implementation of RSA modular exponentiation, we used the technique of sliding-window method with 5-bit window size. Our result showed very fast performance, that

Table 2. Comparison of evaluation results for two types of implemented code for 4096-bit REDC of our proposal, one is with C language and another is with hand-assembled code

	C language	Hand-assembled code
Total cycles	65,111	18,801
Total instructions (NOT including NOP instruction)	177,199	75,560
Probability of pipeline stalling	14%	5%
Average IPC of total instructions	2.57	4.02

Table 3. Measured total time of the 1-time execution of our REDC algorithm in our environment

Bit length	Total time on our environment (μ sec)
512	0.62
768	1.15
1024	1.81
2048	5.88
3072	12.24
4096	20.87

is, our implementation attained 1,090 times of 1024-bit RSA CRT decryptions on Itanium 2 at 900MHz.

We compared our result of performance measurement with that of Intel Performance Primitive (IPP) which is software RSA library developed by Intel. This library is well known as fast library on Intel processors series. Especially, it is the fastest RSA library of marketed products on Itanium 2. We show the comparison in Table 4. In this table, results of IPP are obtained by our measurement. Our results are 1.19 – 3.1 times faster than IPP, and are the fastest on Itanium 2.

Table 4. Measured total time of the 1-time execution of our REDC algorithm in our environment

Bit length	INTEL (μ sec)	Our Implementation	Ratio
4096	95,482	30,829	3.09
3072	44,383	14,277	3.10
2048	5,984	4,759	1.25
1024	1,099	917	1.19
768	659	512	1.28
512	313	237	1.32

5 Concluding Remarks

In this paper, we proposed new implementation algorithm of the primitive of Montgomery multiplication (REDC) by improving the Itoh's algorithm. Our new algorithm is suitable for pipeline scheduling on Itanium 2 which has an instruction to operate multiply-add (XMA instruction). And we implemented our new algorithm on Itanium 2. In the implementation, we optimized for parallel processing of REDC with extremely technique of hand-assembled code. By using our optimized code, average IPC of REDC is 4.02. Especially, average IPC in the "Core loop of REDC" is 5.25, which is an extremely good result because maximum IPC of Itanium 2 is 6, and its probability of pipeline stalling of our implementation is just only 5%. We also implemented RSA decryption with CRT based on our optimized code of REDC and technique of sliding-window method with 5-bit window size. Our implementation result attained 32 times of 4096-bit RSA decryption of CRT on Itanium 2 at 900MHz. Our REDC and RSA implementation can process variable bit length of RSA. And our RSA implementation performs 3.1 times faster than Intel's library in the best case.

A motivation of this paper is to realize fast implementation of RSA with long (say 4096-bit) keys. For such keys, Karatsuba and/or FFT algorithms may work better than conventional Montgomery's approach. Comparing such implementations (especially on Itanium 2) will be our future work.

Acknowledgement. The authors would like to thank to Kouichi Kumon for his technical advice of code optimazation on Itanium 2, and Tetsuya Izu for his editorial suggestions of the paper. We also thank anonymous referees of this paper for their comments.

References

[1] P.L.Montgomery, "Modular Multiplication without Trial Division", Mathematics of Computation, Vol.44, No.170, pp.519-521, 1985.

[2] C.K.Koc, T.Acar and B.S.Kaliski Jr., "Analyzing and Comparing Montgomery Multiplication Algorithms", IEEE Macro, Vol.16, No.3, pp.26-33, June 1996.

[3] K.Itoh, M.Takenaka, N.Torii, S.Temma and Y. Kurihara, "Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201", Cryptographic Hardware and Embedded Systems -CHES '99,(LNCS1717), pp.61-72, 1999.

[4] Itanium 2 Processor Reference Manual for Software Development and Optimization, Intel, 2002.6

[5] Intel Itanium Architecture Software Developer's Manuals, Intel, 2005

[6] Intel C++ Compiler for Linux Reference, Intel, 2005

[7] A.F.Tenca, G.Todorov and C.K.Koc, "High-Radix Design of a Scrable Modular Muitiplier", Cryptographic Hardware and Embedded Systems - CHES 2001, (LNCS 2162), pp. 185-201, 2001.

[8] C.D.Walter, "Systolic Modular Multiplication", IEEE Trans. Computers, vol.42, no.3, pp. 376-378, Mar, 1993.

[9] G.Orlando and C.Paar, "A Scalable GF(p) Elliptic Curve Processor Archtechture for Programable Hardware" Cryptographic Hardware and Embedded Systems - CHES 2001 (LNCS 2162), pp. 348-363, 2001.

[10] S.E.Eldridge and C.D.Walter, "Hardware Implementation of Montgomery's Modular Multiplication Algorithm", IEEE Transactions on Computers, vol. 42, no. 6, July 1993, pp. 693699.

[11] C.D.Walter, "Montgomery's Multiplication Technique: How to Make It Smaller and Faster", Cryptographic Hardware and Embedded Systems - CHES'99 (LNCS 1717), pp. 80-93, 1999.

[12] A.F.Tenca and C.K.Koc, “A Scalable Archtecture for Montgomery Multiplication”, Cryptographic Hardware and Embedded Systems - CHES ’99 (LNCS 1717), pp. 94-108, 1999.
[13] R.L.Rivest, A.Shamir and L.Adleman, “A Method of obtaining digital signature and public key cryptosystems”, Comm.of ACM, Vol.21, No.2, pp.120-126, Feb.1978.
[14] Paul Barrett, “Implementing the Rivest, Shamir, and Adleman Public-Key Encryption Algorithm on a Standard Digital Signal Processor”, Advances in Cryptology-CRYPTO’86(LNCS263), pp.311-323, 1987.
[15] M.E.Kaihara and N.Takagi, “Bipartile Modular Multiplication”, Cryptographic Hardware and Embedded Systems - CHES 2005 (LNCS 3659), pp. 185-210, 2005.
[16] National Institute for Standards and Technology (NIST), SP 800-57: Recommendation on Key Management, 2005.
[17] National Institute for Standards and Technology (NIST), SP 800-78: Cryptographic Algorithms and Key Sizes for Personal Identity Verification, 2005.
[18] S.R.Dusse and B.S.Kaliski Jr., “A Crytographic Library for the Motorola DSP56000”, Advances in Cryptology - EUROCRYPTO ’90 (LNCS 473), pp.230-244, 1990.