A Fast Implementation of the Optimal Ate Pairing over BN curve on Intel Haswell Processor

Shigeo MITSUNARI [*]

June 11, 2013

Abstract

We present an efficient implementation of the Optimal Ate Pairing on Barreto-Naehrig curve over a 254-bit prime field on Intel Haswell processor. Our library is able to compute the optimal ate pairing over a 254-bit prime field, in just 1.17 million of clock cycles on a single core of an Intel Core i7-4700MQ(2.4GHz) processor with TurboBoost technology disabled.

Keywords: optimal ate pairing, efficient implementation, Haswell

1 Introduction

Bilinear maps on elliptic curves are important tools for generating many interesting encryption protocols. This paper provides an efficient software implementation of asymmetric bilinear pairings at high security levels. We present a library that performs the optimal ate pairing over a 254-bit Barreto-Naehrig (BN) curve in just 1.17 million of clock cycles on a single core of an Intel i7-4700MQ 2.4GHz (Haswell) processor with TurboBoost technology disabled.

Haswell processor supports a new instruction named `mulx`, which performs an unsigned multiplication of 64-bit integer without writing the arithmetic flags unlike `mul` instruction. We apply `mulx` instruction to the straightforward multiplication of two 256-bit integers (producing a 512-bit integer), then the timings of the pairing are reduced from 1.33M cycles to 1.17M cycles.

The full source code of our implementation is available from `https://github.com/herumi/ate-pairing/`.

[*] Cybozu Labs, Inc.

2 Parameters of BN Curves

We use BN curves[1] defined by the equation $E : y^2 = x^3 + 2$ over $\mathbb{F}_p$, where p is defined as follows[3]:

$$z = -(2^{62} + 2^{55} + 1), \quad p = 36z^4 + 36z^3 + 24z^2 + 6z + 1.$$

p is a 254-bit prime, which implies that the security level achieved for such a BN curve would be of approximately 127 bits. Next, we represent $\mathbb{F}_{p^{12}}$ using the following extensions[3]:

$$\begin{aligned}
\mathbb{F}_{p^2} &= \mathbb{F}_p[u]/(u^2 + 1), \\
\mathbb{F}_{p^6} &= \mathbb{F}_{p^2}[v]/(v^3 - \xi), \quad \xi = -u - 1, \\
\mathbb{F}_{p^{12}} &= \mathbb{F}_{p^6}[w]/(w^2 - v).
\end{aligned}$$

3 Operation Costs

Let **mul256** denote the multiplication of two 256-bit integers, producing a 512-bit integer, and **red512** the Montgomery modular reduction of 512-bit integers to $\mathbb{F}_p$. Let m_u and r denote the cost of **mul256** and **red512** respectively, then the cost of field multiplication in $\mathbb{F}_p$, denoted as m is $m_u + r$. The cost of multiplication and squaring in $\mathbb{F}_{p^2}$ is $3m_u + 2r$ and $2m_u + 2r$ respectively. Here, we omit the costs of addition operations.

We compare the operation costs for different implementations of the optimal ate pairing of the same parameters. Table 1 shows the operation costs of the pairing of [3], our previous work[4], and this work. Table 2 in the Section 6 of [3] does not contain the cost of m and r of $\mathbb{F}_p$, then we add the costs of $(282m + 6m_u + 4r)$, $(30m + 75m_u + 50r)$ to the costs of [3] for the Miller loop and the final exponentiation respectively.

Table 1: Operation Cost of the pairing

Phase	Aranha *et al.*[3]	our previous[4]/this work
Miller loop	$6792m_u + 3022r$	$6785m_u + 3022r$
Final exp.	$3753m_u + 2006r$	$3526m_u + 1932r$
Optimal Ate pairing	$10545m_u + 5028r$	$10311m_u + 4954r$

4 Implementation

This section shows an efficient implementation of **mul256** and **red512** for Haswell processor. Let **mul256x64** denote the multiplication of 256-bit integer and 64-bit integer, producing a 320-bit integer. An efficient implementation of **mul256x64** is important because each **mul256** and **red512** call **mul256x64** four times.

Let x_i, y, z_i, and t_i denote 64-bit general purpose registers, and `mul`, `add`, and `adc` a multiplication instruction, an addition instruction, an addition instruction with carry flag (CF) of two 64-bit registers respectively. The registers named rax and rdx are special registers for destination of `mul` instruction.

4.1 Our previous implementation

This section shows a detail of the implementation of **mul256x64** in our previous work[4]. It is necessary for `adc` instruction to keep CF generated by other `add` and `adc` instruction. However, `mul` instruction changes the arithmetic flags such as CF, then it is difficult to deal with `mul` and `adc` simultaneously. Moreover the destination registers of `mul` instruction are fixed to rax and rdx register.

Algorithm 1 shows our previous implementation[4] with `mul` instruction to implement **mul256x64**, which needs five temporary registers t_0, …, t_4 generated by `mul` instructions. Therefore, it requires many `mov` instructions to keeps them.

```
Algorithm 1 : mul256x64 without mulx
input: $[x_3{:}x_2{:}x_1{:}x_0]$ : 256-bit integer, $y$ : 64-bit integer
output: $[z_4{:}z_3{:}z_2{:}z_1{:}z_0]$ : 320-bit integer
$[\mathrm{rdx}{:}\mathrm{rax}] \leftarrow \mathtt{mul}(x_0, y)$ , $[t_0{:}z_0] \leftarrow [\mathrm{rdx}{:}\mathrm{rax}]$
$[\mathrm{rdx}{:}\mathrm{rax}] \leftarrow \mathtt{mul}(x_1, y)$ . $[t_2{:}t_1] \leftarrow [\mathrm{rdx}{:}\mathrm{rax}]$
$[\mathrm{rdx}{:}\mathrm{rax}] \leftarrow \mathtt{mul}(x_2, y)$ , $[t_4{:}t_3] \leftarrow [\mathrm{rdx}{:}\mathrm{rax}]$
$[\mathrm{rdx}{:}\mathrm{rax}] \leftarrow \mathtt{mul}(x_3, y)$
$(z_1,\mathrm{CF}) \leftarrow \mathtt{add}(t_0, t_1)$
$(z_2,\mathrm{CF}) \leftarrow \mathtt{adc}(t_2, t_3, \mathrm{CF})$
$(z_3,\mathrm{CF}) \leftarrow \mathtt{adc}(t_4, \mathrm{rax}, \mathrm{CF})$
$z_4 \leftarrow \mathtt{adc}(\mathrm{rdx}, 0, \mathrm{CF})$
return $[z_4{:}z_3{:}z_2{:}z_1{:}z_0]$
```

4.2 Our implementation

On the other hand, `mulx` instruction[5] supported by Haswell processor does not affect to CF, then we can use `mulx` and `adc` instruction simultaneously. Moreover we can select any registers for destination of `mulx`.

Algorithm 2 shows an implementation of **mul256x64** with `mulx` instructions, which needs two temporary registers t_0, t_1. As a result, we can remove some `mov` instructions to implement **mul256x64**, therefore Algorithm 2 reduces 36 `mov` instructions to implement **mul256** and **red512** compared with Algorithm 1.

Algorithm 2 : **mul256x64** with mulx

```
input: $[x_3{:}x_2{:}x_1{:}x_0]$ : 256-bit integer, $y$ : 64-bit integer
output: $[z_4{:}z_3{:}z_2{:}z_1{:}z_0]$ : 320-bit integer
$[t_0{:}z_0] \leftarrow \texttt{mulx}(x_0, y)$
$[\mathrm{rax}{:}t_1] \leftarrow \texttt{mulx}(x_1, y)$
$(z_1,\mathrm{CF}) \leftarrow \texttt{add}(t_0, t_1)$
$[t_1{:}t_0] \leftarrow \texttt{mulx}(x_2, y)$
$(z_2,\mathrm{CF}) \leftarrow \texttt{adc}(\mathrm{rax}, t_0, \mathrm{CF})$
$[\mathrm{rax}{:}t_0] \leftarrow \texttt{mulx}(x_3, y)$
$(z_3,\mathrm{CF}) \leftarrow \texttt{adc}(t_1, t_0, \mathrm{CF})$
$z_4 \leftarrow \texttt{adc}(\mathrm{rax}, 0, \mathrm{CF})$
return $[z_4{:}z_3{:}z_2{:}z_1{:}z_0]$
```

5 Benchmark

Table 2 shows a comparison of operation counts for different implementations of the optimal Ate pairing. According to the score at Core i5 in the Table 2, our previous implementation[4] is slightly faster than that of Aranha *et al.*[3] and this work is 13% faster than our previous implementation on a same Haswell processor.

6 Conclusion

We applied the new instruction `mulx` supported with Haswell to an implementation of the optimal Ate pairing, and our implementaion, which runs in 1.17M cycles on Haswell processor, improves that result in 13%.

Table 2: Cycle counts of the operations for different implementation of the optimal Ate pairing

implementation	Aranha *et al.*[3]	our previous work[4]			this work
CPU	Core i5[a]	Core i5[b]	Core i7[c]	Haswell[d]	Haswell[d] with `mulx`
TurboBoost	on	on	on	off	off
m_u	–	69	50	42	38
r	–	110	85	69	65
Miller lp.	0.978	0.97	0.83	0.82	0.71
Final exp.	0.710	0.62	0.54	0.51	0.46
Opt Ate	1.688	1.59	1.37	1.33	1.17

[a] Core i5 M540 on Linux

[b] Core i5 M520 on Windows 7

[c] Core i7 2600K 3.4GHz on Windows 7

[d] Core i7 4700MQ 2.4GHz on Linux

References

[1] P.S.L.M. Barreto and M. Naehrig. Pairing-friendly elliptic curves of prime order. In B. Preneel and S. Tavares, editors, Selected Areas in Cryptography SAC 2005, volume 3897 of LNCS, pp.319–331. Springer, 2006.

[2] J.L. Beuchat, J.E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. Pairing 2010, pp. 21–39, 2010.

[3] D.F. Aranha, K. Karabina, P. Longa, C.H. Gebotys, and J. López. Faster explicit formulas for computing pairings over ordinary curves. EUROCRYPT'11, pp. 48–68, 2011. `http://eprint.iacr.org/2010/526`

[4] S. Mitsunari, T. Teruya, and E. Okamoto. Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves. SCIS2012(In Japanese)

[5] Intel Architecture Instruction Set Extensions Programming Reference 319433-014 Aug. 2012