Binary Field Multiplication on ARMv8

Hwajeong Seo[1], Zhe Liu[2], Yasuyuki Nogami[3],
Jongseok Choi[1], and Howon Kim[1]⋆

[1] Pusan National University,
School of Computer Science and Engineering,
San-30, Jangjeon-Dong, Geumjeong-Gu, Busan 609–735, Republic of Korea
{hwajeong,jschoi85,howonkim}@pusan.ac.kr
[2] University of Luxembourg,
Laboratory of Algorithmics, Cryptology and Security (LACS),
6, rue R. Coudenhove-Kalergi, L–1359 Luxembourg-Kirchberg, Luxembourg
{zhe.liu}@uni.lu
[3] Okayama University,
Graduate School of Natural Science and Technology,
3-1-1, Tsushima-naka, Kita, Okayama, 700-8530, Japan
{yasuyuki.nogami}@okayama-u.ac.jp

Abstract. In this paper, we show efficient implementations of binary field multiplication over ARMv8. We exploit an advanced 64-bit polynomial multiplication (PMULL) supported by ARMv8 and conduct multiple levels of asymptotically faster Karatsuba multiplication. Finally, our method conducts binary field multiplication within 57 and 153 clock cycles for B-251 and B-571, respectively. Our proposed method on ARMv8 improves the performance by a factor of 5.5 $\sim$ 7.2 times than previous techniques on ARMv7.

Keywords: Polynomial Multiplication, Binary Field Multiplication, ARMv8, Elliptic Curve Cryptography, Karatsuba Multiplication

1 Introduction

Since binary field multiplication is an important component of elliptic curve cryptography and authenticated encryption, many researches have studied the high speed implementation of binary field multiplication in software engineering. The typical binary field multiplication over embedded processor may compute the results with bitwise-xor and logical shift operations. The other more clever approach exploits the look-up table by calculating the part of results in advance [6, 7, 10, 8, 9]. Recently, many modern embedded processors adopt the advanced built-in binary field multiplication. ARMv7 supports VMULL.P8 operation which can compute eight 8-bit wise polynomial multiplications with single instruction. In [2], author shows that efficient implementation techniques to construct the 64-bit binary field multiplication with the VMULL.P8 operation. After then

⋆ Corresponding Author

multiple levels of Karatsuba multiplication is applied to several binary field multiplications including $\mathbb{F}_{2^{251}}$, $\mathbb{F}_{2^{283}}$ and $\mathbb{F}_{2^{571}}$. The most recent processor, ARMv8, supports `PMULL` operation which can compute 64-bit wise polynomial multiplication with single instruction. In [3], author shows that compact implementation of GCM based authenticated encryption with the `PMULL` operation. Since the 64-bit multiplication is quite fast enough for 128-bit multiplication, they avoid Karatsuba multiplication. The detailed 128-bit polynomial multiplication is available in Algorithm 1. The implementations achieved 11 times faster results than ARMv7. However, the paper does not show binary field multiplication for long length operands. The long operands are required to compute ECC based cryptography. In this paper, we present efficient implementations of long bits binary field multiplication. We applied multiple levels of Karatsuba multiplication and achieved 5.5 $\sim$ 7.2 times faster than ARMv7 implementations.

Algorithm 1 128-bit Polynomial Multiplication (`mul128_p64`)

```
Require: 128-bit Operands $A$, $B$.
Ensure: 256-bit Result $C$.
ld1.16b {v2}, [x2]                 {Load 128-bit Operand $A$}
ld1.16b {v3}, [x1]                 {Load 128-bit Operand $B$}
movi.16b v6, #0                    {Clear Reg}
pmull v0.1q, v2.1d, v3.1d          {$C_L \leftarrow A_{[63:0]} \cdot B_{[63:0]}$}
pmull2 v1.1q, v2.2d, v3.2d         {$C_H \leftarrow A_{[127:64]} \cdot B_{[127:64]}$}
ext.16b v4, v3, v3, #8             {Shuffle Operand $B$}
pmull v5.1q, v2.1d, v4.1d          {$C_M \leftarrow A_{[63:0]} \cdot B_{[127:64]}$}
pmull2 v4.1q, v2.2d, v4.2d         {$C_{M'} \leftarrow A_{[127:64]} \cdot B_{[63:0]}$}
eor.16b v4, v4, v5                 {$C_M \leftarrow C_M \oplus C_{M'}$}
ext.16b v5, v6, v4, #8             {Align Result}
eor.16b v0, v0, v5                 {$C_{L[127:64]} \leftarrow C_{L[127:64]} \oplus C_{M[63:0]}$}
ext.16b v5, v4, v6, #8             {Align Result}
eor.16b v1, v1, v5                 {$C_{H[63:0]} \leftarrow C_{H[63:0]} \oplus C_{M[127:64]}$}
st1.16b {v0,v1}, [x0]              {Return 256-bit Result $C$}
```

The remainder of this paper is organized as follows. In Section 2, we recap the target ARM processor and Karatsuba method. In Section 3, we propose the efficient binary field multiplication. In Section 4, we evaluate the performance of proposed methods in terms of clock cycles. Finally, Section 5 concludes the paper.

2 Related Works

2.1 ARM Processor

ARM processor is a well known family of RISC processor architectures introduced in 1985 [11]. The most recent version, ARMv8, supports both 32-bit and 64-bit processing. The 32-bit ARMv8 architecture is known as AArch32, while the 64-bit is known as AArch64. An ARMv8 processor can support both, allowing the execution of 32-bit and 64-bit applications. ARM processors support a single-instruction multiple-data (SIMD) module called the NEON engine. AArch32 features sixteen 32-bit registers (`R0-R15`) and sixteen 128-bit NEON registers (`Q0-Q15`). The NEON registers can also be viewed as pairs of 64-bit registers (`D0-D32`). For example, `D0` and `D1` are the lower and higher parts of `Q0`, respectively. AArch64 features thirty two 64-bit registers (`X0-X31`) and thirty two 128-bit NEON registers (`V0-V31`). The NEON registers can no longer be viewed as pairs of 64-bit registers. From ARMv8, two polynomial dedicated instructions, `PMULL` and `PMULL2`, are available. Both of which carry out a single 64-bit multiplication. In both cases, the inputs are 128-bit registers. Their difference is that in `PMULL` the lower 64-bit parts of the inputs are used as operands, while in `PMULL2` the higher 64-bit parts are used [3].

2.2 Karatsuba Algorithm

The basic idea of Karatsuba multiplication is to split a multiplication of two s words operands into three multiplications of size $\frac{s}{2}$, which is possible at the expense of some additions [5]. Taking the multiplication of s words operands A and B as an example, we represent the operands as $A = A_H \cdot 2^{\frac{s}{2}} + A_L$ and $B = B_H \cdot 2^{\frac{s}{2}} + B_L$. The multiplication $P = A \cdot B$ can be computed according to the Equation 1.

$$A_H \cdot B_H \cdot 2^s + [(A_H + A_L)(B_H + B_L) - A_H \cdot B_H - A_L \cdot B_L] \cdot 2^{\frac{s}{2}} + A_L \cdot B_L$$

Karatsuba method roughly executes $\frac{3s^2}{4}$ `mul` instructions to multiply two s-word operands [4]. Recently, the refined Karatsuba's algorithm from a Crypto 2009 paper by Bernstein [1] makes efficient use of the available registers to keep the low overheads from `load` and `store` instructions.

3 Proposed Method

3.1 Polynomial Multiplication

Polynomial multiplication can be implemented in ordinary or Karatsuba method. The Karatsuba multiplication is an efficient approach when size of operand is long enough than processor's word. Since the ARMv8 processor is 64-bit architecture, several parameters are not favorable with Karatsuba multiplication. In this section, we investigate the proper size of minimum Karatsuba technique.

Four and Eight Terms of Multiplications The efficient Karatsuba multiplication technqiues are highly relied on the number of terms where term is (operand size/word size). In this section we firstly explore the two, four and eight terms of multiplications. For two terms as studied in [3], ordinary multiplication is efficient than Karatsuba approach. As we can see the comparison results in Table 1. Karatsuba method can reduce the number of multiplication by 1. However, more number of other instructions including `eor` and `ext` are required. For this reason, ordinary multiplication is better choice for 128-bit polynomial multiplication.

Table 1. Comparison of 128-bit polynomial multiplication methods

Instructions	`pmull`	`eor`	`movi`	`ext`
Ordinary	4	3	1	3
Karatsuba	3	6	1	4

In case of long operands (256, 512 and etc), Karatsuba approach is better choice, because total number of instructions are smaller than ordinary approach. We use the Karatsuba algorithm for both cases and our solution is a combination of the Karatsuba algorithm and a multiplier based on `PMULL` which we have named the Karatsuba/NEON/`PMULL` multiplier (KNP).

Four terms (256-bit) polynomial multiplication is executed with 1 level of Karatsuba multiplication. The inner 128-bit multiplication is readily established with Algorithm 1. The detailed four terms polynomial multiplication is available in Algorithm 2. In Step 1 and 2, 256-bit operands are loaded from memory. In Step 3 and 4, lower parts of operands are bit-wise exclusive-ored with higher parts. In Step 5, 128-bit register (`v14`) is cleared. In Step 6, 7 and 8, three 128-bit wise polynomial multiplications ($C_L \leftarrow A_L \cdot B_L$ and $C_H \leftarrow A_H \cdot B_H$ and $C_M \leftarrow (A_L \oplus A_H) \cdot (B_L \oplus B_H)$) are conducted. The first two input variables of `mul128_p64` are results and third and fourth variables represent operands and the last three operands indicate temporal storages. From Step 9 to 14, intermediate results are bit-wise exclusive-ored. At step 15, total 512-bit results are stored into memory.

Eight terms (512-bit) polynomial multiplication consists of 2 levels of Karatsuba multiplication. The first level consists of 256-bit polynomial multiplications and the multiplication is following the Algorithm 2. The detailed 512-bit multiplication is written in Algorithm 3. In Step 1 and 2, 256-bit operands are loaded from memory. In Step 3 and 6, lower parts of operands are bit-wise exclusive-ored with higher parts. In Step 7, 128-bit register (`v14`) is cleared. In Step 8, 9 and 10, three 128-bit wise polynomial multiplications ($C_L \leftarrow A_L \cdot B_L$ and $C_H \leftarrow A_H \cdot B_H$ and $C_M \leftarrow (A_L \oplus A_H) \cdot (B_L \oplus B_H)$) are conducted. The first two input variables of `mul256_p64` are results and third and fourth variables represent operands and the last three operands indicate temporal storages. From Step 11 to 22, intermediate results are bit-wise exclusive-ored. At step 23 and 24, total 1024-bit results are stored into memory.

Three, Six and Nine Terms of Multiplications In this section we explore the three, six and nine terms of multiplications. For three terms, ordinary mul-

Algorithm 2 256-bit Polynomial Multiplication (mul256_p64)

```
Require: 256-bit Operands $A$, $B$.
Ensure: 512-bit Result $C$.
ld1.16b {v6, v7}, [x2]                                  {Load 256-bit Operand $A$}
ld1.16b {v9, v10}, [x1]                                 {Load 256-bit Operand $B$}
eor.16b v8, v6, v7                                      {$A_L \oplus A_H$}
eor.16b v11, v9, v10                                    {$B_L \oplus B_H$}
movi.16b v14, #0                                        {Clear Reg}
mul128_p64 v0, v1, v6, v9, v12, v13, v14                {$C_L \leftarrow A_L \cdot B_L$}
mul128_p64 v2, v3, v7, v10, v12, v13, v14               {$C_H \leftarrow A_H \cdot B_H$}
mul128_p64 v4, v5, v8, v11, v12, v13, v14 {$C_M \leftarrow (A_L \oplus A_H) \cdot (B_L \oplus B_H)$}
eor.16b v4, v4, v0                  {$C_{M[127:0]} \leftarrow C_{M[127:0]} \oplus C_{L[127:0]}$}
eor.16b v5, v5, v1            {$C_{M[255:128]} \leftarrow C_{M[255:128]} \oplus C_{L[255:128]}$}
eor.16b v4, v4, v2                  {$C_{M[127:0]} \leftarrow C_{M[127:0]} \oplus C_{H[127:0]}$}
eor.16b v5, v5, v3            {$C_{M[255:128]} \leftarrow C_{M[255:128]} \oplus C_{H[255:128]}$}
eor.16b v1, v1, v4              {$C_{L[255:128]} \leftarrow C_{L[255:128]} \oplus C_{M[127:0]}$}
eor.16b v2, v2, v5                {$C_{H[127:0]} \leftarrow C_{H[127:0]} \oplus C_{M[255:128]}$}
st1.16b {v0,v1,v2,v3}, [x0]                             {Return 512-bit Result $C$}
```

tiplication is the more efficient method than that of Karatsuba. The comparison results are drawn in Table 2. Three terms of Karatsuba multiplication reduces the number of multiplication from 9 to 6. However, additional 8 and 5 times of `eor` and `ext` instructions are required. For this reason, ordinary multiplication is better choice for 192-bit polynomial multiplication.

Table 2. Comparison of 192-bit polynomial multiplication methods

Instructions	`pmull`	`eor`	`movi`	`ext`
Ordinary	6	16	1	8
Karatsuba	9	7	1	3

The detailed three terms of polynomial multiplication is available in Algorithm 4. In Step 1 and 2, 192-bit operands (A and B) are loaded from memory. By using option 8b, we loaded operands by sequential 64-bit format to the 128-bit registers, which lefts higher 64-bit as an empty. In Step 3 $\sim$ 6, four multiplications including ($C_L \leftarrow A_{[63:0]} \cdot B_{[63:0]}$, $T_L \leftarrow A_{[63:0]} \cdot B_{[127:64]}$, $C_M \leftarrow A_{[63:0]} \cdot B_{[191:128]}$ and $Temp \leftarrow A_{[127:64]} \cdot B_{[63:0]}$) are conducted. After then results ($A_{[127:64]} \cdot B_{[63:0]}$) are added to T_L. From Step 8 to 15, remaining multiplications are conducted and then added to the intermediate results. After then the results are aligned and accumulated to the intermediate results in Step 17 $\sim$ 22. Finally, total 384-bit results are stored into memory.

For six terms of polynomial multiplication, we firstly split six terms largely into two parts consisting of three terms. The two part is computed with 1 level of Karatsuba multiplication. The inner three-term of multiplication is executed

Algorithm 3 512-bit Polynomial Multiplication

```
Require: 512-bit Operands $A$, $B$.
Ensure: 1024-bit Result $C$.
ld1.16b {v0, v1, v2, v3}, [x2]                                      {Load 512-bit Operand $A$}
ld1.16b {v6, v7, v8, v9}, [x1]                                      {Load 512-bit Operand $B$}
eor.16b v4, v0, v2                                                  {$A_{L[127:0]} \oplus A_{H[127:0]}$}
eor.16b v5, v1, v3                                                  {$A_{L[255:128]} \oplus A_{H[255:128]}$}
eor.16b v10, v6, v8                                                 {$B_{L[127:0]} \oplus B_{H[127:0]}$}
eor.16b v11, v7, v9                                                 {$B_{L[255:128]} \oplus B_{H[255:128]}$}
movi.16b v16, #0                                                    {Clear Reg}
mul256_p64 v18, v19, v20, v21, v30, v31, v0, v1, v12, v6, v7, v13,
  v14, v15, v16                                                     {$C_L \leftarrow A_L \cdot B_L$}
mul256_p64 v22, v23, v24, v25, v30, v31, v2, v3, v12, v8, v9, v13,
  v14, v15, v16                                                     {$C_H \leftarrow A_H \cdot B_H$}
mul256_p64 v26, v27, v28, v29, v30, v31, v4, v5, v12, v10, v11, v13,
  v14, v15, v16                                                     {$C_M \leftarrow (A_L \oplus A_H) \cdot (B_L \oplus B_H)$}
eor.16b v26, v26, v18                                               {$C_{M[127:0]} \leftarrow C_{M[127:0]} \oplus C_{L[127:0]}$}
eor.16b v27, v27, v19                                               {$C_{M[255:128]} \leftarrow C_{M[255:128]} \oplus C_{L[255:128]}$}
eor.16b v28, v28, v20                                               {$C_{M[383:256]} \leftarrow C_{M[383:256]} \oplus C_{L[383:256]}$}
eor.16b v29, v29, v21                                               {$C_{M[511:384]} \leftarrow C_{M[511:384]} \oplus C_{L[511:384]}$}
eor.16b v26, v26, v22                                               {$C_{M[127:0]} \leftarrow C_{M[127:0]} \oplus C_{H[127:0]}$}
eor.16b v27, v27, v23                                               {$C_{M[255:128]} \leftarrow C_{M[255:128]} \oplus C_{H[255:128]}$}
eor.16b v28, v28, v24                                               {$C_{M[383:256]} \leftarrow C_{M[383:256]} \oplus C_{H[383:256]}$}
eor.16b v29, v29, v25                                               {$C_{M[511:384]} \leftarrow C_{M[511:384]} \oplus C_{H[511:384]}$}
eor.16b v20, v20, v26                                               {$C_{L[383:256]} \leftarrow C_{L[383:256]} \oplus C_{M[127:0]}$}
eor.16b v21, v21, v27                                               {$C_{L[511:384]} \leftarrow C_{L[511:384]} \oplus C_{M[255:128]}$}
eor.16b v22, v22, v28                                               {$C_{H[127:0]} \leftarrow C_{H[127:0]} \oplus C_{M[383:256]}$}
eor.16b v23, v23, v29                                               {$C_{H[255:128]} \leftarrow C_{H[255:128]} \oplus C_{M[511:384]}$}
st1.16b {v18, v19, v20, v21}, [x0], #64                             {Return 512-bit Result $C_{[511:0]}$}
st1.16b {v22, v23, v24, v25}, [x0], #64                             {Return 512-bit Result $C_{[1023:512]}$}
```

Algorithm 4 192-bit Polynomial Multiplication (mul192_p64)

```
Require: 192-bit Operands $A$, $B$.
Ensure: 384-bit Result $C$.
ld1.8b {v0, v1, v2}, [x2]             {Load 192-bit Operand $A$}
ld1.8b {v3, v4, v5}, [x1]             {Load 192-bit Operand $B$}
pmull v6.1q, v0.1d, v3.1d             {$C_L \leftarrow A_{[63:0]} \cdot B_{[63:0]}$}
pmull v9.1q, v0.1d, v4.1d             {$T_L \leftarrow A_{[63:0]} \cdot B_{[127:64]}$}
pmull v7.1q, v0.1d, v5.1d             {$C_M \leftarrow A_{[63:0]} \cdot B_{[191:128]}$}
pmull v11.1q, v1.1d, v3.1d            {$Temp \leftarrow A_{[127:64]} \cdot B_{[63:0]}$}
eor.16b v9, v9, v11                   {$T_L \leftarrow T_L \oplus Temp$}
pmull v11.1q, v1.1d, v4.1d            {$Temp \leftarrow A_{[127:64]} \cdot B_{[127:64]}$}
eor.16b v7, v7, v11                   {$C_M \leftarrow C_M \oplus Temp$}
pmull v10.1q, v1.1d, v5.1d            {$T_H \leftarrow A_{[127:64]} \cdot B_{[191:128]}$}
pmull v11.1q, v2.1d, v3.1d            {$Temp \leftarrow A_{[191:128]} \cdot B_{[63:0]}$}
eor.16b v7, v7, v11                   {$C_M \leftarrow C_M \oplus Temp$}
pmull v11.1q, v2.1d, v4.1d            {$Temp \leftarrow A_{[191:128]} \cdot B_{[127:64]}$}
eor.16b v10, v10, v11                 {$T_H \leftarrow T_H \oplus Temp$}
pmull v8.1q, v2.1d, v5.1d             {$C_H \leftarrow A_{[191:128]} \cdot B_{[191:128]}$}
movi.16b v11, #0                      {Clear Reg}
ext.16b v11, v11, v9 , #8             {Align Result}
ext.16b v9, v9, v10 , #8              {Align Result}
ext.16b v10, v10, v11, #8             {Align Result}
eor.16b v6, v6, v11                   {$C_{L[127:64]} \leftarrow C_{L[127:64]} \oplus T_{L[63:0]}$}
eor.16b v7, v7, v9                    {$C_M \leftarrow C_M \oplus \{T_{H[63:0]} || T_{L[127:64]}\}$}
eor.16b v8, v8, v10                   {$C_{H[63:0]} \leftarrow C_{H[63:0]} \oplus T_{H[127:64]}$}
st1.16b {v6,v7,v8}, [x0]              {Return 384-bit Result $C$}
```

with ordinary multiplication described in Algorithm 4. The detailed six terms of polynomial multiplication is available in Algorithm 5. In Step 1 $\sim$ 4, 384-bit operands are loaded from memory. In Step 5 $\sim$ 10, lower parts of operands are bit-wise exclusive-ored with higher parts. In Step 11, 128-bit register (`v14`) is cleared. In Step 12, 13 and 14, three 192-bit wise polynomial multiplications ($C_L \leftarrow A_L \cdot B_L$ and $C_H \leftarrow A_H \cdot B_H$ and $C_M \leftarrow (A_L \oplus A_H) \cdot (B_L \oplus B_H)$) are conducted. The first six input variables of `mul192_p64` are operands and seventh, eighth and nine-th variables represent temporal storages and the last three operands indicate results. From Step 15 to 28, intermediate results are bit-wise exclusive-ored. At step 29 and 30, total 768-bit results are stored into memory.

For nine terms of polynomial multiplication, we applied three terms of Karatsuba algorithm. The three terms of Karatsuba multiplication replaces two s words operands into six multiplications of size $\frac{s}{3}$, which is possible at the expense of some additions [5]. Taking the multiplication of s words operands A and B as an example, we represent the operands as $A = A_H \cdot 2^{\frac{2s}{3}} + A_M \cdot 2^{\frac{s}{3}} + A_L$ and $B = B_H \cdot 2^{\frac{2s}{3}} + B_M \cdot 2^{\frac{s}{3}} + B_L$. The multiplication $P = A \cdot B$ can be computed according to the Equation 2.

$$\begin{aligned}(A_H \cdot B_H \cdot 2^{\frac{2s}{3}} + A_M \cdot B_M \cdot 2^{\frac{s}{3}} + A_L \cdot B_L) \cdot (2^{\frac{2s}{3}} + 2^{\frac{s}{3}} + 1) +\\ (A_L + A_M) \cdot (B_L + B_M) \cdot 2^s + (A_H + A_L) \cdot (B_H + B_L) \cdot 2^{\frac{2s}{3}} +\\ (A_M + A_L) \cdot (B_M + B_L) \cdot 2^{\frac{s}{3}}\end{aligned} \tag{1}$$

For nine terms of polynomial multiplication, we firstly split nine terms largely into three parts consisting of three terms. The three parts are computed with 1 level of three terms of Karatsuba algorithm. The inner three-term of multiplication is executed with ordinary multiplication described in Algorithm 4. The detailed three terms of Karatsuba multiplication follows the Equation 2.

3.2 Binary Field Multiplication

s word of binary field multiplication produce values of degree at most $2s - 2$, which must be reduced modulo $f(z) = z^m + r(z)$. The usual approach is to multiply the higher parts by $r(z)$ using shift and xors. For small polynomials $r(z)$ we can exploit the `PMULL` instruction to carry out 64-bit multiplication by $r(z)$.

Curve B-251 The modulo of binary field $\mathbb{F}_{2^{251}}$ is defined by ($r(z)$= $z^7 + z^4 + z^2 + 1$). The detailed reduction method is available in Algorithm 6. In Step 1 $\sim$ 3, intermediate results are shifted by 59 to the right. In Step 4 and 5, 60 $\sim$ 64-th bits are cleared. In Step 6 and 7, intermediate results are shifted by 5 to the left. In Step 8 $\sim$ 10, the results are aligned and then the shifted results are accumulated. In Step 13 and 14, the lower 8-bit of register `v15` is set to `0x95`. In Step 15 $\sim$ 19, the higher intermediate results are multiplied by $r(z)$. In Step 20

Algorithm 5 384-bit Polynomial Multiplication

```
Require: 384-bit Operands $A$, $B$.
Ensure: 768-bit Result $C$.
ld1.8b {v0, v1, v2}, [x2], #24              {Load 192-bit Operand $A_{[191:0]}$}
ld1.8b {v3, v4, v5}, [x2], #24              {Load 192-bit Operand $A_{[383:192]}$}
ld1.8b {v9 , v10, v11}, [x1], #24           {Load 192-bit Operand $B_{[191:0]}$}
ld1.8b {v12, v13, v14}, [x1], #24           {Load 192-bit Operand $B_{[383:192]}$}
eor.16b v6, v0, v3                          {$A_{L[63:0]} \oplus A_{H[63:0]}$}
eor.16b v7, v1, v4                          {$A_{L[127:64]} \oplus A_{H[127:64]}$}
eor.16b v8, v2, v5                          {$A_{L[191:128]} \oplus A_{H[191:128]}$}
eor.16b v15, v9, v12                        {$B_{L[63:0]} \oplus B_{H[63:0]}$}
eor.16b v16, v10, v13                       {$B_{L[127:64]} \oplus B_{H[127:64]}$}
eor.16b v17, v11, v14                       {$B_{L[191:128]} \oplus B_{H[191:128]}$}
movi.16b v18, #0                            {Clear Reg}
mul192_p64 v0, v1, v2, v9, v10, v11, v19, v20, v21, v25, v26, v27
                                            {$C_L \leftarrow A_L \cdot B_L$}
mul192_p64 v3, v4, v5, v12, v13, v14, v19, v20, v21, v28, v29, v30
                                            {$C_H \leftarrow A_H \cdot B_H$}
mul192_p64_normal v6, v7, v8, v15, v16, v17, v19, v20, v21, v0, v1, v2
                                            {$C_M \leftarrow (A_L \oplus A_H) \cdot (B_L \oplus B_H)$}
eor.16b v0, v0, v25                         {$C_{M[127:0]} \leftarrow C_{M[127:0]} \oplus C_{L[127:0]}$}
eor.16b v1, v1, v26                         {$C_{M[255:128]} \leftarrow C_{M[255:128]} \oplus C_{L[255:128]}$}
eor.16b v2, v2, v27                         {$C_{M[383:256]} \leftarrow C_{M[383:256]} \oplus C_{L[383:256]}$}
eor.16b v0, v0, v28                         {$C_{M[127:0]} \leftarrow C_{M[127:0]} \oplus C_{H[127:0]}$}
eor.16b v1, v1, v29                         {$C_{M[255:128]} \leftarrow C_{M[255:128]} \oplus C_{H[255:128]}$}
eor.16b v2, v2, v30                         {$C_{M[383:256]} \leftarrow C_{M[383:256]} \oplus C_{H[383:256]}$}
ext.16b v4, v18, v0, #8                     {Align Result}
ext.16b v5, v0, v1, #8                      {Align Result}
ext.16b v6, v1, v2, #8                      {Align Result}
ext.16b v7, v2, v18, #8                     {Align Result}
eor.16b v26, v26, v4                        {$C_{L[255:191]} \leftarrow C_{L[255:191]} \oplus C_{M[63:0]}$}
eor.16b v27, v27, v5                        {$C_{L[383:256]} \leftarrow C_{L[383:256]} \oplus C_{M[191:64]}$}
eor.16b v28, v28, v6                        {$C_{H[127:0]} \leftarrow C_{H[127:0]} \oplus C_{M[319:192]}$}
eor.16b v29, v29, v7                        {$C_{H[191:128]} \leftarrow C_{H[191:128]} \oplus C_{M[383:320]}$}
st1.16b {v25,v26,v27}, [x0], #48            {Return 384-bit Result $C_{[383:0]}$}
st1.16b {v28,v29,v30}, [x0], #48            {Return 384-bit Result $C_{[767:384]}$}
```

$\sim$ 27, the computed results are aligned and then accumulated to intermediate results. From Step 28 to 41, one more round of reduction process is conducted.

Curve B-571 The modulo of binary field $\mathbb{F}_{2^{571}}$ is defined by ($r(z)= z^{10} + z^5 + z^2 + 1$). We can also conduct 64-bit multiplication by $r(z)$ as a similar way described in Algorithm 6 to get the reduced results.

4 Evaluation

In order to test ARMv8 instruction set, we set the development environment as follows. We used Xcode (ver 6.3.2) as a development IDE and tested the program over iPad Mini2 (iOS 8.4). The iPad Mini2 supports Apple A7 with 64-bit architecture operated in 1.3GHz. In Table 3, we show performance results onv various length of polynomial multiplications. For 192-bit, we used ordinary multiplication and the other parameters exploit the Karatsuba multiplication.

Table 3. Performance of polynomial multiplication, (2): two-term, (3): three-term

Length	192	256	384	512	576
Karatsuba	-	1-level(2)	1-level(2)	2-level(2)	1-level(3)
Clock Cycles	11	21	49	75	99

In Table 4, the comparison results of binary field multiplication is available. Previous works by [2] uses KNV method. This method is quiet slow because it adopts eight vectorized 8-bit polynomial multiplication namely `VMULL`. Unlike previous works, we exploit new 64-bit polynomial multiplication namely `PMULL`. This method significantly improves the performance by a factor of 5.5 $\sim$ 7.2 times than previous works. However, we couldn't compare our results with the methods on same ARMv8 architecture because this is the first implementation for ECC friendly binary field multiplication. The recent work by [3] only explores the short 128-bit binary field multiplication.

Table 4. Comparison results of binary field multiplication

Algorithm	Architecture	Processor	$\mathbb{F}_{2^{251}}$	$\mathbb{F}_{2^{571}}$
KNV [2]	Cortex-A8	ARMv7	385	1,506
KNV [2]	Cortex-A9	ARMv7	491	1,889
KNV [2]	Cortex-A15	ARMv7	317	1,103
Proposed Method (KNP)	Apple-A7	ARMv8	57	153

Algorithm 6 251-bit Polynomial Multiplication

Require: 502-bit Operands A.

Ensure: 251-bit Result C.

```
ushr.2d v4, v1, #59                   {Right Shift by #59}
ushr.2d v5, v2, #59                   {Right Shift by #59}
ushr.2d v6, v3, #59                   {Right Shift by #59}
shl.2d v1, v1, #5                     {Clear High Bits}
ushr.2d v1, v1, #5                    {Clear High Bits}
shl.2d v7, v2, #5                     {Left Shift by #5}
shl.2d v8, v3, #5                     {Left Shift by #5}
ext.16b v4, v4, v5, #8                {Align Result}
ext.16b v5, v5, v6, #8                {Align Result}
ext.16b v6, v6, v14, #8               {Align Result}
eor.16b v4, v4, v7                    {Accumulate Intermediate Result}
eor.16b v5, v5, v8                    {Accumulate Intermediate Result}
movi.16b v15, #149                    {Set 0x95}
ushr.2d v15, v15, #56                 {Clear High 56-bit}
pmull v16.1q, v4.1d, v15.1d           {Multiply Modulo}
pmull2 v17.1q, v4.2d, v15.2d          {Multiply Modulo}
pmull v18.1q, v5.1d, v15.1d           {Multiply Modulo}
pmull2 v19.1q, v5.2d, v15.2d          {Multiply Modulo}
pmull v20.1q, v6.1d, v15.1d           {Multiply Modulo}
ext.16b v21, v14, v17, #8             {Align Result}
ext.16b v22, v17, v19, #8             {Align Result}
ext.16b v23, v19, v14, #8             {Align Result}
eor.16b v16, v16, v21                 {Accumulate Intermediate Result}
eor.16b v18, v18, v22                 {Accumulate Intermediate Result}
eor.16b v2, v20, v23                  {Accumulate Intermediate Result}
eor.16b v0, v0, v16                   {Accumulate Intermediate Result}
eor.16b v1, v1, v18                   {Accumulate Intermediate Result}
ushr.2d v4, v1, #59                   {Right Shift by #59}
ushr.2d v5, v2, #59                   {Right Shift by #59}
shl.2d v1, v1, #5                     {Clear High Bits}
ushr.2d v1, v1, #5                    {Clear High Bits}
shl.2d v7, v2, #5                     {Left Shift by #5}
ext.16b v4, v4, v5, #8                {Align Result}
eor.16b v4, v4, v7                    {Accumulate Intermediate Result}
pmull v16.1q, v4.1d, v15.1d           {Multiply Modulo}
pmull2 v17.1q, v4.2d, v15.2d          {Multiply Modulo}
ext.16b v21, v14, v17, #8             {Align Result}
ext.16b v22, v17, v14, #8             {Align Result}
eor.16b v0, v0, v16                   {Accumulate Intermediate Result}
eor.16b v0, v0, v21                   {Accumulate Intermediate Result}
eor.16b v1, v1, v22                   {Accumulate Intermediate Result}
```

5 Conclusion

In this paper, we show efficient implementation techniques for binary field multiplication on ARMv8. Our proposed method improves the performance by a factor of $5.5 \sim 7.2$ times than previous techniques on ARMv7. Our future works are efficient binary field elliptic curve cryptography with proposed techniques.

References

1. D. J. Bernstein. Batch binary edwards. In *Advances in Cryptology-CRYPTO 2009*, pages 317–336. Springer, 2009.
2. D. Câmara, C. P. Gouvêa, J. López, and R. Dahab. Fast software polynomial multiplication on arm processors using the neon engine. In *Security Engineering and Intelligence Informatics*, pages 137–154. Springer, 2013.
3. C. P. Gouvêa and J. López. Implementing gcm on armv8. In *Topics in Cryptology—CT-RSA 2015*, pages 167–180. Springer, 2015.
4. J. Großschädl, R. M. Avanzi, E. Savaş, and S. Tillich. Energy-efficient software implementation of long integer modular arithmetic. In *Cryptographic Hardware and Embedded Systems–CHES 2005*, pages 75–90. Springer, 2005.
5. A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, page 595, 1963.
6. J. López and R. Dahab. High-speed software multiplication in f2m. In *Progress in CryptologyINDOCRYPT 2000*, pages 203–212. Springer, 2000.
7. L. B. Oliveira, D. F. Aranha, C. P. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab. Tinypbc: Pairings for authenticated identity-based non-interactive key distribution in sensor networks. *Computer Communications*, 34(3):485–493, 2011.
8. H. Seo, Y. Lee, H. Kim, T. Park, and H. Kim. Binary and prime field multiplication for public key cryptography on embedded microprocessors. *Security and Communication Networks*, 7(4):774–787, 2014.
9. H. Seo, Z. Liu, J. Choi, and H. Kim. Karatsuba–block-comb technique for elliptic curve cryptography over binary fields. *Security and Communication Networks*, 2015.
10. M. Shirase, Y. Miyazaki, T. Takagi, and D.-G. HAN. Efficient implementation of pairing-based cryptography on a sensor node. *IEICE transactions on information and systems*, 92(5):909–917, 2009.
11. Steve Ranger. Internet of things and wearables drive growth for ARM. Available for download at `http://www.zdnet.com/article/internet-of-things-and-wearables-drive-growth-for-arm/`, Apr. 2014.