

# 《现代密码学》实验报告

|             |                       |
|-------------|-----------------------|
| 实验名称：有限域的实现 | 实验时间：2025 年 10 月 16 日 |
| 学生姓名：谢润文    | 学号：23336262           |
| 学生班级：23 计科  | 成绩评定：                 |

## 一、实验目的

通过实现  $GF(2^{131})$  上的有限域运算（包括加、乘、平方、求逆），理解有限域元素的存储方式和运算规则，提高设计有关位运算算法的能力，通过指令优化进一步降低复杂度。

## 二、实验内容

用 C/C++ 实现  $GF(2^{131})$  上的多项式加法、多项式乘法、平方运算，以及分别用费马小定理和 Exgcd（扩展欧几里得算法，Extended gcd）实现求逆运算。

参数要求：输入由以下部分组成：uint32\_t 需要进行的运算数量，紧随着若干个运算操作，uint8\_t 进行的运算类型（包括 0x00 加法、0x01 乘法、0x02 平方、0x03 求逆），随后是 uint64\_t[3][2] 进行运算的两个有限域元素。每次运算的结果以 uint64\_t[3] 输出，不使用的高位需要全部设为 0。

## 三、实验原理

1. 有限域元素的表示： $GF(2^{131})$  的元素可以表示  $a_0 + a_1x + a_2x^2 + \dots + a_{130}x^{130}$  的形式，为此以 uint64\_t[3] 表示  $GF(2^{131})$  上的元素，第一个 uint64\_t 表示  $(a_{63}, a_{62}, \dots, a_0)$ ，第二个 uint64\_t 表示  $(a_{127}, a_{126}, \dots, a_{64})$ ，第三个 uint64\_t 以  $(0, 0, \dots, a_{130}, a_{129}, a_{128})$  表示。在 C++ 中采用结构体方式表示  $GF(2^{131})$ ，开一个 uint64\_t num[6] 的数组表示（为了方便后文提到的乘法）。

2. 有限域的加法。以 c 表示加法结果，a 和 b 作为运算元素，注意到模 2 加相当于异或，只需 `for(i:0->2) a.num[i]^b.num[i]` 即可。

3. 有限域的取模。这里仅考虑最高次为 260 的取模，是因为两元素做乘法运算后最多不超过  $130*2=260$  次。首先将 [192, 260] 位的数移位异或到 [61, 142] 位中。注意到在给定的有限域中， $x^{131} \equiv x^{13} + x^2 + x + 1$ ，同时乘以 x 的 k 次幂 ( $k \in [61, 129]$  且  $k \in \mathbb{Z}$ )，可以得到移位关系。然后，同理可将 [192, 131] 中的数移位至到低位并异或。具体算法如下：

---

**算法 1  $F_{2^{131}}$  模运算**

---

```

Input:  $e = (E[4], E[3], E[2], E[1], E[0])$ .
Output:  $b = e \bmod f(z)$ .
1 for  $i = 4 \rightarrow 3$  do
2    $T \leftarrow E[i]$ ;
3    $E[i - 3] \leftarrow E[i - 3] \oplus (T \ll 61) \oplus (T \ll 62) \oplus (T \ll 63)$ ;
4    $E[i - 2] \leftarrow E[i - 2] \oplus (T \ll 10) \oplus (T \gg 1) \oplus (T \gg 2) \oplus (T \gg 3)$ ;
5    $E[i - 1] \leftarrow E[i - 1] \oplus (T \gg 54)$ ;
6 end
7  $T \leftarrow (E[2] \& 0xFFFFFFFFFFFFFFFC)$ ;
8  $B[0] \leftarrow E[0] \oplus (T \ll 10) \oplus (T \gg 1) \oplus (T \gg 2) \oplus (T \gg 3)$ ;
9  $B[1] \leftarrow E[1] \oplus (T \gg 54)$ ;
10  $B[2] \leftarrow E[2] \& 0x7$ ;
11 return  $b \leftarrow (B[2], B[1], B[0])$ ;

```

---

图 1  $GF(2^{131})$  上的快速模运算

其中有一处错误， $0xFFFFFFFFFFFFFFFC$  应为  $0xFFFFFFFFFFFFFFF8$ ，否则会导致第 130 位也加入了第二步的模运算，实际上应当保留第 130 位。

4. 有限域的乘法。基于 Karatsuba 算法和 PCLMULQDQ 指令，一共需要 6 次 64 位的乘法运算。具体算法如下：

---

**算法 2  $F_{2^{131}}$  乘法**

---

```

Input:  $a = (A[2], A[1], A[0]), b = (B[2], B[1], B[0])$ .
Output:  $c = ab$ .
1  $T_0 \leftarrow A[1] \oplus A[2]$ ;
2  $T_1 \leftarrow B[1] \oplus B[2]$ ;
3  $T_2 \leftarrow A[0] \oplus A[2]$ ;
4  $T_3 \leftarrow B[0] \oplus B[2]$ ;
5  $T_4 \leftarrow A[0] \oplus A[1]$ ;
6  $T_5 \leftarrow B[0] \oplus B[1]$ ;
7  $T_0 \leftarrow \text{PCLMULQDQ}(T_0, T_1)$ ;
8  $T_1 \leftarrow \text{PCLMULQDQ}(T_2, T_3)$ ;
9  $T_2 \leftarrow \text{PCLMULQDQ}(T_4, T_5)$ ;
10  $T_3 \leftarrow \text{PCLMULQDQ}(A[0], B[0])$ ;
11  $T_4 \leftarrow \text{PCLMULQDQ}(A[1], B[1])$ ;
12  $T_5 \leftarrow \text{PCLMULQDQ}(A[2], B[2])$ ;
13  $T_6 \leftarrow T_0 \oplus T_4 \oplus T_5$ ;
14  $T_7 \leftarrow T_2 \oplus T_4 \oplus T_3$ ;
15  $T_8 \leftarrow T_1 \oplus T_3 \oplus T_4 \oplus T_5$ ;
16  $c \leftarrow (T_5, T_6, T_8, T_7, T_3)$ ;
17 return  $c$ ;

```

图 2  $GF(2^{131})$  上的快速乘法运算

5. 有限域的平方。注意到该有限域中每一个元素可以写成  $a \ll 128 + b \ll 64 + c$  的形式，而  $(a \ll 128 + b \ll 64 + c)^2 = (a^2 \ll 256) + b^2 \ll 128 + c^2 + 2 * [(a \ll 128) * (b \ll 64) + (b \ll 64) * c + (a \ll 128) * c]$   $\equiv (a^2 \ll 256) + b^2 \ll 128 + c^2 \pmod{2}$ ，只需要分别对每 64 位（即  $\text{num}[0:2]$ ）做一次 PCLMULQDQ 乘法即可，共需 3 次 PCLMULQDQ 乘法和 1 次取模运算，相比于复用普通的乘法省去了 3 次 PCLMULQDQ 乘法，更为高效。

6. 有限域的求逆。第一种方法是，利用费马小定理有  $x^{\text{pow}(2, m)-1} = 1$ ，于是可知  $x^{\text{pow}(2, m)-2} = x^{-1}$ 。即求元素的逆需要计算  $x^{\text{pow}(2, m)-2}$ 。此处  $m=131$ 。 $m-1$  的二进制表示是  $(10000010)$ ，基于 Itoh-Tsujii 算法，我们有：

$$\begin{array}{ll}
X_0 = X & X^{\text{pow}(2, 1)-1} \\
X_1 = X_0 X_0^{\text{pow}(2, 1)} & X^{\text{pow}(2, 2)-1} \\
X_2 = X_1 X_1^{\text{pow}(2, 2)} & X^{\text{pow}(2, 4)-1} \\
X_3 = X_2 X_2^{\text{pow}(2, 4)} & X^{\text{pow}(2, 8)-1} \\
X_4 = X_3 X_3^{\text{pow}(2, 8)} & X^{\text{pow}(2, 16)-1} \\
X_5 = X_4 X_4^{\text{pow}(2, 16)} & X^{\text{pow}(2, 32)-1} \\
X_6 = X_0 (X_5 X_5^{\text{pow}(2, 32)})^2 & X^{\text{pow}(2, 65)-1} \\
X_7 = X_6 X_6^{\text{pow}(2, 65)} & X^{\text{pow}(2, 130)-1}
\end{array}$$

结果就是  $X_7^2$ 。

第二种方法是利用扩展欧几里得算法（下称 Exgcd）来做。这是基于下列事实：  
 $\gcd(a, f)=1$ , 其中 a 是待求逆的多项式, f 是题目中给出的不可约多项式。那么根据 Exgcd, 一定能找出多项式 g 和 h, 使得  $ag+fh=1$ 。在 mod f 意义下, 上式找出的 g 相当于 a 的逆。  
具体算法如下:

---

**Algorithm 2.48** Inversion in  $\mathbb{F}_{2^m}$  using the extended Euclidean algorithm

---

INPUT: A nonzero binary polynomial  $a$  of degree at most  $m - 1$ .

OUTPUT:  $a^{-1} \bmod f$ .

1.  $u \leftarrow a, v \leftarrow f$ .
  2.  $g_1 \leftarrow 1, g_2 \leftarrow 0$ .
  3. While  $u \neq 1$  do
    - 3.1  $j \leftarrow \deg(u) - \deg(v)$ .
    - 3.2 If  $j < 0$  then:  $u \leftrightarrow v, g_1 \leftrightarrow g_2, j \leftarrow -j$ .
    - 3.3  $u \leftarrow u + z^j v$ .
    - 3.4  $g_1 \leftarrow g_1 + z^j g_2$ .
  4. Return( $g_1$ ).
- 

图 3 基于 Exgcd 的求逆算法

#### 四、实验步骤（源代码）

```

#include<iostream>
#include<cstdint>
#include<fcntl.h>
#include<wmmmintrin.h>
#include<immintrin.h>
using namespace std;
/*
FILE *fp=fopen("input.bin","rb");
FILE *p=fopen("output.bin","wb");
*/

```

```

#define fp stdin
#define p stdout

struct gf{
    uint64_t num[6]={0};
};

int gf_deg(gf &c){
    for(int i=2;i>=0;--i){
        for(int k=63;k>=0;--k){
            if((c.num[i]&(1ull<<k))!=0){
                return i*64+k;
            }
        }
    }
    return 0;
}

inline void gf_print(const gf c){
    for(int i=0;i<3;++i) fwrite(&c.num[i],sizeof(uint64_t),1,p);
}

inline void gf_add(gf &c,const gf a,const gf b){
    for(int i=0;i<3;++i) c.num[i]=a.num[i]^b.num[i];
}

inline void gf_mul(gf &c,const gf a,const gf b){
    __m128i tmp[6];
    uint64_t t[9];
    t[0]=a.num[1]^a.num[2];
    t[1]=b.num[1]^b.num[2];
    t[2]=a.num[0]^a.num[2];
    t[3]=b.num[0]^b.num[2];
    t[4]=a.num[0]^a.num[1];
    t[5]=b.num[0]^b.num[1];
    __m128i a0=_mm_set_epi64x(0,a.num[0]);
    __m128i a1=_mm_set_epi64x(0,a.num[1]);
    __m128i a2=_mm_set_epi64x(0,a.num[2]);
}

```

```

__m128i b0=_mm_set_epi64x(0,b.num[0]);
__m128i b1=_mm_set_epi64x(0,b.num[1]);
__m128i b2=_mm_set_epi64x(0,b.num[2]);
__m128i t0=_mm_set_epi64x(0,t[0]);
__m128i t1=_mm_set_epi64x(0,t[1]);
__m128i t2=_mm_set_epi64x(0,t[2]);
__m128i t3=_mm_set_epi64x(0,t[3]);
__m128i t4=_mm_set_epi64x(0,t[4]);
__m128i t5=_mm_set_epi64x(0,t[5]);
tmp[0]=_mm_clmulepi64_si128(t0,t1,0x00);
tmp[1]=_mm_clmulepi64_si128(t2,t3,0x00);
tmp[2]=_mm_clmulepi64_si128(t4,t5,0x00);
tmp[3]=_mm_clmulepi64_si128(a0,b0,0x00);
tmp[4]=_mm_clmulepi64_si128(a1,b1,0x00);
tmp[5]=_mm_clmulepi64_si128(a2,b2,0x00);
t[3]=_mm_extract_epi64(tmp[3],0);
t[5]=_mm_extract_epi64(tmp[5],0);
//cout<<hex<<_mm_extract_epi64(tmp[2]^tmp[3]^tmp[4],0)<<""
"<<hex<<_mm_extract_epi64(tmp[3],1)<<endl;

t[6]=_mm_extract_epi64(tmp[0]^tmp[4]^tmp[5],0)^_mm_extract_epi64(tmp[1]
^tmp[3]^tmp[4]^tmp[5],1);

t[7]=_mm_extract_epi64(tmp[2]^tmp[3]^tmp[4],0)^_mm_extract_epi64(tmp[3],
1);
//cout<<hex<<t[7]<<endl;

t[8]=_mm_extract_epi64(tmp[1]^tmp[3]^tmp[4]^tmp[5],0)^_mm_extract_epi64
(tmp[2]^tmp[3]^tmp[4],1);
c.num[0]=t[3];
c.num[1]=t[7];
c.num[2]=t[8];
c.num[3]=t[6];
c.num[4]=t[5]^_mm_extract_epi64(tmp[0]^tmp[4]^tmp[5],1);

```

```

}

inline void gf_mod(gf &c){
    gf b;
    uint64_t tmp;
    for(int i=4;i>=3;i--){
        tmp=c.num[i];
        c.num[i-3]=c.num[i-3]^(tmp<<61)^(tmp<<62)^(tmp<<63);
        c.num[i-2]=c.num[i-2]^(tmp<<10)^(tmp>>1)^(tmp>>2)^(tmp>>3);
        c.num[i-1]=c.num[i-1]^(tmp>>54);
    }
    tmp=(c.num[2]&0xFFFFFFFFFFFFFF8); //not 0xFFFFFFFFFFFFFFFC!!!
    c.num[0]=c.num[0]^(tmp<<10)^(tmp>>1)^(tmp>>2)^(tmp>>3);
    c.num[1]=c.num[1]^(tmp>>54);
    c.num[2]=c.num[2]&0x7;
    c.num[3]=c.num[4]=0x0;
}

inline void gf_pow2(gf &c,const gf a){
    __m128i tmp0,tmp1,tmp2;
    __m128i a0=_mm_set_epi64x(0,a.num[0]);
    __m128i a1=_mm_set_epi64x(0,a.num[1]);
    __m128i a2=_mm_set_epi64x(0,a.num[2]);
    tmp0=_mm_clmulepi64_si128(a0,a0,0x00);
    tmp1=_mm_clmulepi64_si128(a1,a1,0x00);
    tmp2=_mm_clmulepi64_si128(a2,a2,0x00);
    c.num[0]=_mm_extract_epi64(tmp0,0);
    c.num[1]=_mm_extract_epi64(tmp0,1);
    c.num[2]=_mm_extract_epi64(tmp1,0);
    c.num[3]=_mm_extract_epi64(tmp1,1);
    c.num[4]=_mm_extract_epi64(tmp2,0);
}

inline void gf_inv(gf &c,const gf a){
    gf tmp;
    for(int i=0;i<3;++i) tmp.num[i]=a.num[i];
    for(int i=1;i<130;++i){

```

```

        gf_pow2(tmp,tmp);
        gf_mod(tmp);
        gf_mul(tmp,tmp,a);
        gf_mod(tmp);
    }

    gf_pow2(c,tmp);
    gf_mod(c);

}

inline void gf_inv_it(gf &c,const gf a){

    gf x1,x2,x3,x4,x5,x6,x7,tmp;
    gf_pow2(x1,a);
    gf_mod(x1);
    gf_mul(x1,x1,a);
    gf_mod(x1);
    for(int i=0;i<3;++i) tmp.num[i]=x1.num[i];
    for(int i=1;i<=2;++i){
        gf_pow2(tmp,tmp);
        gf_mod(tmp);
    }
    gf_mul(x2,tmp,x1);
    gf_mod(x2);
    for(int i=0;i<3;++i) tmp.num[i]=x2.num[i];
    for(int i=1;i<=4;++i){
        gf_pow2(tmp,tmp);
        gf_mod(tmp);
    }
    gf_mul(x3,tmp,x2);
    gf_mod(x3);
    for(int i=0;i<3;++i) tmp.num[i]=x3.num[i];
    for(int i=1;i<=8;++i){
        gf_pow2(tmp,tmp);
        gf_mod(tmp);
    }
    gf_mul(x4,tmp,x3);
}

```

```
gf_mod(x4);

for(int i=0;i<3;++i) tmp.num[i]=x4.num[i];

for(int i=1;i<=16;++i){

    gf_pow2(tmp,tmp);

    gf_mod(tmp);

}

gf_mul(x5,tmp,x4);

gf_mod(x5);

for(int i=0;i<3;++i) tmp.num[i]=x5.num[i];

for(int i=1;i<=32;++i){

    gf_pow2(tmp,tmp);

    gf_mod(tmp);

}

gf_mul(tmp,tmp,x5);

gf_mod(tmp);

gf_pow2(tmp,tmp);

gf_mod(tmp);

gf_mul(x6,tmp,a);

gf_mod(x6);

for(int i=0;i<3;++i) tmp.num[i]=x6.num[i];

for(int i=1;i<=65;++i){

    gf_pow2(tmp,tmp);

    gf_mod(tmp);

}

gf_mul(x7,tmp,x6);

gf_mod(x7);

gf_pow2(c,x7);

gf_mod(c);

}

/*
void gf_pow(gf &c,const gf a,int power){

    if(power==0){

        c.num[0]=0x1;

        for(int i=1;i<5;++i) c.num[i]=0;
```

```

        return;
    }

    if(power==1){
        for(int i=0;i<5;++i) c.num[i]=a.num[i];
        return;
    }

    else{
        gf _a;
        for(int i=0;i<5;++i){
            _a.num[i]=a.num[i];
            c.num[i]=0;
        }
        c.num[0]=0x1;
        while(power){
            if(power&1){
                gf_mul(c,c,_a);
                gf_mod(c);
            }
            gf_pow2(_a,_a);
            gf_mod(_a);
            power>>=1;
        }
        gf_mod(c);
    }
}

void gf_inv_gcd(gf &c,const gf a){
    gf u,v,g1,g2,z;
    int j,degu=0,degv=0;
    for(int i=0;i<3;++i) u.num[i]=a.num[i];
    v.num[0]=0x2007;
    v.num[2]=0x8;
    g1.num[0]=0x1;
    while(1){

```

```

        if(u.num[0]==0x1 && u.num[1]==0 && u.num[2]==0){
            for(int i=0;i<3;++i) c.num[i]=g1.num[i];
            return;
        }
        degu=gf_deg(u);
        degv=gf_deg(v);
        j=degu-degv;
        if(j<0){
            j=-j;
            gf tmp2;
            for(int i=0;i<3;++i){
                tmp2.num[i]=u.num[i];
                u.num[i]=v.num[i];
                v.num[i]=tmp2.num[i];
                tmp2.num[i]=g1.num[i];
                g1.num[i]=g2.num[i];
                g2.num[i]=tmp2.num[i];
            }
        }
        gf tmp,tmp1,tmp2;
        tmp.num[j/64]=1ull<<(j%64);
        gf_mul(tmp1,tmp,v);
        gf_mod(tmp1);
        gf_add(u,u,tmp1);
        gf_mul(tmp2,tmp,g2);
        gf_mod(tmp2);
        gf_add(g1,g1,tmp2);
        gf_mod(u);
        gf_mod(g1);
        gf_mod(g2);
    }
}

int main(){
    #ifdef _WIN32

```

```

setmode(fileno(stdin), O_BINARY);
setmode(fileno(stdout), O_BINARY);
#endif
uint32_t n,dat_n;
uint8_t type,dat_type;
uint64_t dat[8];
gf a,b,c;
fread(&dat_n,sizeof(uint32_t),1,fp);
n=dat_n;
while(n--){
    fread(&dat_type,sizeof(uint8_t),1,fp);
    type=dat_type;
    fread(&dat,sizeof(uint64_t),6,fp);
    for(int i=0;i<3;++i){
        a.num[i]=dat[i];
        b.num[i]=dat[i+3];
    }
    if(type==0) gf_add(c,a,b);
    else if(type==1){
        gf_mul(c,a,b);
        //gf_print(c);
        gf_mod(c);
    }
    else if(type==2){
        gf_pow2(c,a);
        //gf_pow(c,a,2);
        //gf_mul(c,a,a);
        gf_mod(c);
    }
    else{
        //gf_inv(c,a);
        gf_inv_it(c,a);
        //gf_inv_gcd(c,a);
    }
}

```

```

        gf_print(c);
    }
}

```

## 五、实验结果

```

#include<iostream>
#include<cstdint>
#include<fcntl.h>
#include<wmmmintrin.h>
#include<jmmmintrin.h>
using namespace std;

FILE *fp=fopen("input.bin","rb");
FILE *p=fopen("output.bin","wb");

/*
#define fp stdin
#define p stdout
*/

```

UI File Edit View Workspace Extras Help | output.bin (\*)

Hex editor

| Address   | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F | ASCII             |
|-----------|---|-------------------|
| 00000000: | 0C C7 7B 9A C3 3B F2 D4 4F 19 C5 8A FB E2 A1 AA | ...{...;...0..... |
| 00000010: | 02 00 00 00 00 00 00 00 A9 23 11 91 DF 9F 7B 76 | .....#....{v      |
| 00000020: | A7 C3 5A CD 55 CF 34 E7 01 00 00 00 00 00 00 00 | .Z.U.4.....       |
| 00000030: | C9 64 E3 1E 4D C9 0A F4 2F 9D F7 2B 69 BE 39    | .d..M./...+1.9    |
| 00000040: | 02 00 00 00 00 00 00 00 41 84 2B AA 50 AA F6 02 | .....A.+.P...     |
| 00000050: | AB DF 0E 36 FF 82 05 C2 05 00 00 00 00 00 00 00 | ....6.....        |
| 00000060: | E4 72 92 AA EE B8 79 3E DF 84 B5 EE 5E 83 62 1B | .r....y>....^..b. |
| 00000070: | 06 00 00 00 00 00 00 00 0D E4 29 D7 7E 9E 7F DF | .....).~....      |
| 00000080: | B4 14 90 B7 C9 12 5A A7 05 00 00 00 00 00 00 00 | .....Z.....       |

图 4、5 本地运行结果及二进制输出展示

对应样例 2，从图中可以看出加法、乘法、平方、取模、求逆运算均正确，程序无异常。

这里求逆采用的是 Itoh-tsujii 算法。利用 Exgcd 在 OJ 上的提交如下：

|    |            |           |          |         |
|----|------------|-----------|----------|---------|
| #3 | 81.001 ms  | 370.5 KB  | Accepted | 1m 加法   |
| #4 | 65.323 ms  | 569.25 KB | Accepted | 500k 乘法 |
| #5 | 66.574 ms  | 569.25 KB | Accepted | 500k 平方 |
| #6 | 371.748 ms | 555.25 KB | Accepted | 20k 求逆  |

图 6 利用 Exgcd 求逆运行结果

## 六、思考题

1. 使用 Itoh-Tsujii 算法给出的迭代公式求逆，需要进行多少次乘法和平方？检查一下你实现的乘法和平方运算进行一次平均需要多少时间，然后根据计算量估计一次求逆需要的时间，并与实际的执行时间进行对比。

从前述算法不难注意到需要 8 次乘法和  $1+65+32+16+8+4+2+1+1=130$  次平方。下面我的提交记录中，假设所有时间都在运算上，可以看到平均 1 次乘法时间为  $48.229\text{ms}/500k=96.458\text{ns}$ ，平均 1 次平方时间为  $55.076\text{ms}/500k=110.152\text{ns}$ ，一次求逆总和为  $96.458\text{ns}*8+110.152\text{ns}*130=15.0914$  微秒。实际上平均一次求逆时间为  $41.057\text{ms}/20k=2.0529$  微秒，显然出了问题。这个问题在于没有考虑输入输出及预处理的时间消耗瓶颈。我们不妨假设加法运算不太需要耗时，仅仅是输入输出及预处理的时间较长。在 1m 加法中耗时 74.4ms，意味着平均一组输入输出及预处理时间为  $74.4\text{ms}/1m=74.4\text{ns}$ ，除去这部分时间，真正的平均一次乘法时间为  $96.458\text{ns}-74.4\text{ns}=22.058\text{ns}$ ，平均一次平方时间为  $110.152\text{ns}-74.4\text{ns}=35.752\text{ns}$ ，这样计算得到的求逆总和为 4824.224ns 即 4.8242 微秒。这与平均一次求逆的时间比较接近。

|    |           |           |          |         |
|----|-----------|-----------|----------|---------|
| #3 | 74.4 ms   | 370.5 KB  | Accepted | 1m 加法   |
| #4 | 48.229 ms | 569.25 KB | Accepted | 500k 乘法 |
| #5 | 55.076 ms | 569.25 KB | Accepted | 500k 平方 |
| #6 | 41.057 ms | 555.25 KB | Accepted | 20k 求逆  |

图 7 利用 Itoh-tsujii 算法结果

2. 还有什么其他的方法从  $x$  乘到  $x^{\text{pow}(2, 131)-2}$ ？写出迭代过程，然后比较一下它们的计算量和实际运行时间有什么区别。

仍然采用二进制展开，注意到  $\text{pow}(2, 130)-1$  在二进制表示下是 130 个 1，那么采用快速幂的思想，伪代码如下（省去取模过程）：

```

for (i:1->130)

    ans=pow2(ans)

    ans=mul (ans, a)

ans=pow2(ans)

```

需要 130 次乘法和 131 次平方运算，共需 261 次，相比于 Itoh-tsujii 算法的 138 次，

可以得出结论：运行时间大概为 Itoh-tsujii 算法的两倍。在 OJ 上跑的结果如下：

| 测试点 | 时间        | 内存        | 结果       | 提示      |
|-----|-----------|-----------|----------|---------|
| #4  | 49.107 ms | 569.25 KB | Accepted | 500k 乘法 |
| #5  | 75.384 ms | 569.25 KB | Accepted | 500k 平方 |
| #6  | 84.07 ms  | 555.25 KB | Accepted | 20k 求逆  |

图 8 利用快速幂思想的算法结果

可以看到大致符合预期。

### 3. 其他的问题？

1) Exgcd 的平均复杂度？从图 6 可以看出大概是 Itoh-tsujii 算法的 9 倍左右。

在最差情况下，Exgcd 算法会循环 260 次，每一次循环有 2 次乘法运算以及 2 次找度数的运算。找度数的运算如果采用枚举的算法，最坏情况下需要循环 130 次。大概需要  $260 \times 130 + 259 \times 130 + \dots + 2 \times 1 + 1 \times 1 \approx 2 * (130^2 + 129^2 + \dots + 1) \approx 2 * 740805 = 1481610$  次基本运算。

2) PCLMULQDQ 运算时是 128 位的，在乘法运算中应该如何保留结果？

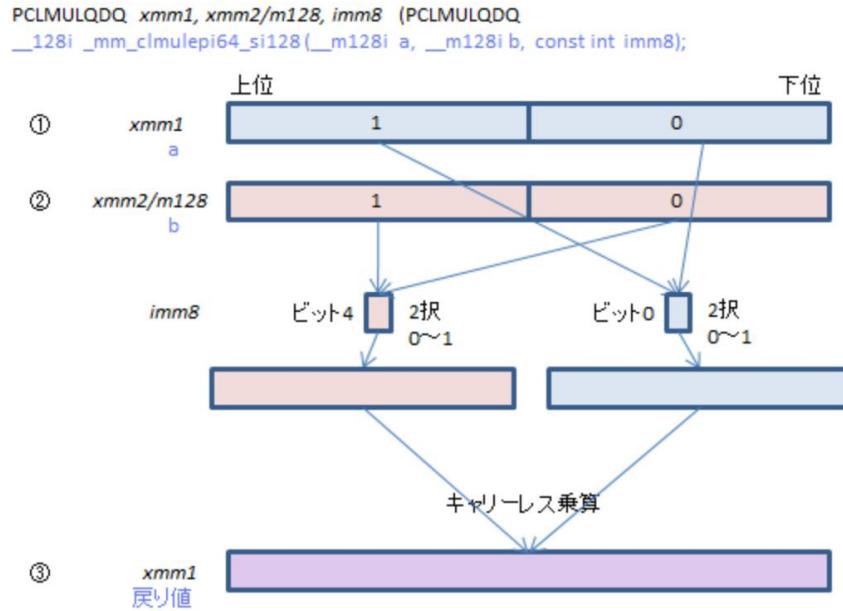


图 9 PCLMULQDQ

结合图 9，我们可以把 64 位的整数均填在低位，然后 *imm8* 的高位选择 0，低位也选择 0。最后的返回值是一个 128 位的整数，这个数的低 64 位可以保留，高 64 位需要与更高次的 64 位进行异或（相当于进位）。

## 七、实验总结

本次实验主要考察有限域的运算，包含二进制流文件读/写、二进制比特串的基本处理、模运算乘法、平方和求逆，其中求逆包含基于费马小定理的算法和 Exgcd 的算法，旨在巩固有限域知识和快速运算、优化算法的思想，为后面 AES 实验提供基础。

在算法设计方面，实现的代码基本上按照论文提供的较优算法，在求逆算法的选择上也参照 Itoh-tsujii 算法从而优化运行时间，得到了比较理想的结果。目前时间的瓶颈可能在二进制流的输入/输出和预处理上。如果提前得出 fread 最大读取的字符数（定长读取），再进行解析，而不是边读入边处理，可能有助于时间的进一步优化。是否如此有待进一步实验验证。