《现代密码学》实验报告

实验名称：AES-128的实现	实验时间：2024年11月20日
学生姓名：黄集瑞	学号：22336090
学生班级：22保密管理	

一、实验目的

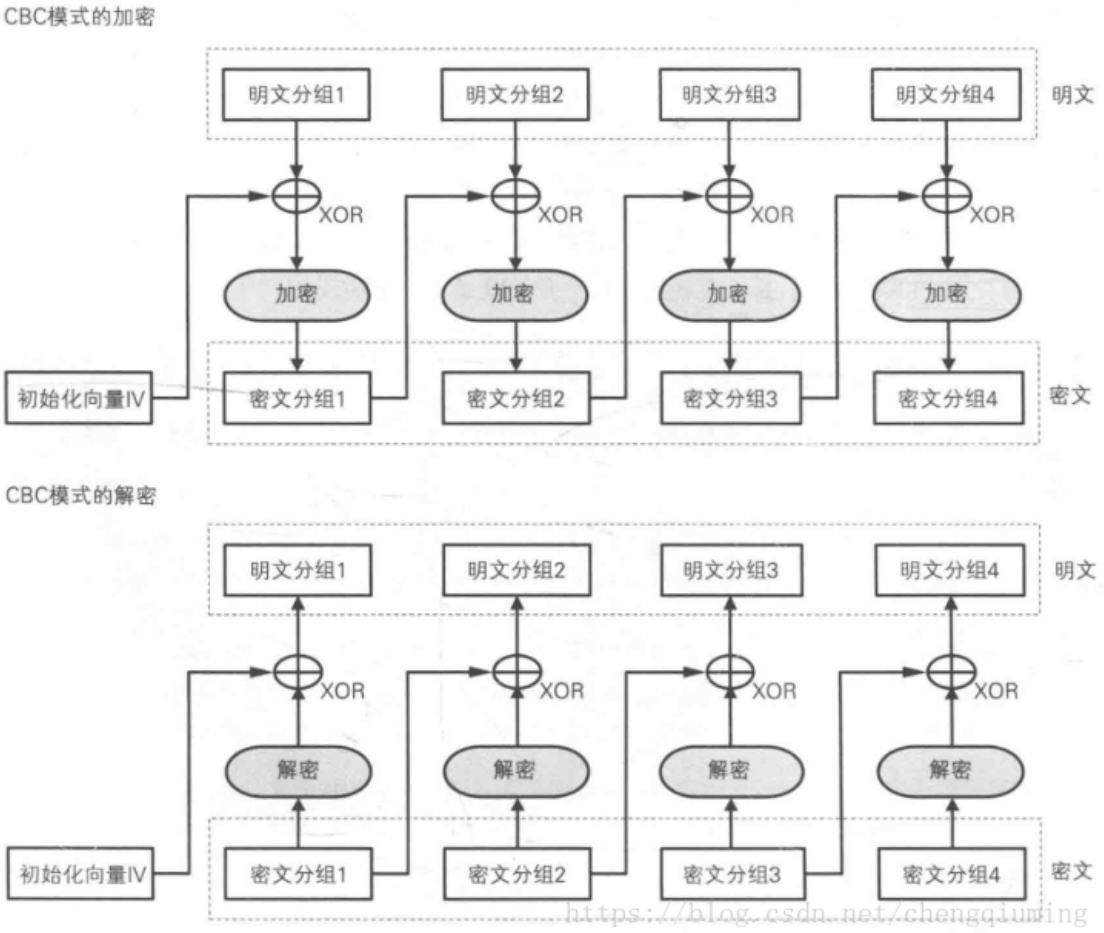
> 本次实验通过实现AES-128 CBC模式的加密与解密，帮助同学们理解并且掌握对称加密算法 AES 的核心原理，包括分组加密、密钥扩展和加密轮次操作，能够提高同学们对AES加密算法的核心步骤和CBC工作模式的理解，并且提升同学们分析问题和解决问题的能力。

二、实验内容

> - 用C++实现AES-128 CBC工作模式的加密和解密
> - 输入如下：
>
> `uint8_t` 工作模式和加密/解密
>
> `0x01` 表示 CBC 模式加密
>
> `0x81` 表示 CBC 模式解密
>
> 最高位为 0 表示加密，1 表示解密
>
> `uint8_t[16]` 密钥
>
> `uint8_t[16]` IV
>
> `uint32_t` 明文/密文长度
>
> `uint8_t[]` 明文/密文内容
>
> - 同时在CBC工作模式下要求明文、密文的长度均为分组长度16bytes的倍数，此时需要我们使用PKCS#7 Padding
> - 注意：输入输出与以往相同均为二进制流，并且是以小端序进行存储的

三、实验原理

> 我认为对于本次实验而言，我们首先需要弄清楚CBC工作模式的原理，然后在此基础上在完成加密以及解密的具体操作。
>
> - 输入以及输出逻辑：
>
> 虽然该部分并没有很好的体现出本次实验的AES的内容以及思想，但是本人认为该部分在本次实验中也是不可或缺的，由于题目测试样例的后两个样例的数据内存都超过了题目设置的16MB，所以这便要求我们不能**直接将所有数据都读入然后进行操作后一起输出**，而是需要我们**读取一轮分组然后进行操作后输出一轮分组**。
>
> - 关于CBC工作模式的原理：该原理在课本上或者互联网上都有着很多的参考资料，也比较容易理解并且实现。



- 关于加密以及解密操作：整个加密以及解密流程算是十分复杂的，需要我们实现非常多的函数来达到我们的目标，而具体的实现流程参考是来自网站上的 NIST 的 AES 标准文档 FIPS 197，同时本人也使用SIMD指令集实现了部分函数的功能来优化性能，但是最终结果并不理想{{{(>_<)}}}。

四、实验步骤

- 输入以及输出逻辑

由于本次实验的数据仍然是二进制数据流，所以我们便沿用上次实验中的输入以及输出模板，具体模板如下所示，这里就不过多赘述。

```
#include <iostream>
#include <fstream>
#include <bitset>
#include <stdint.h>
#include <iomanip>
using namespace std;
#ifdef ONLINE_JUDGE
#include <x86intrin.h>
#define in cin
#define out cout
#else
#include <intrin.h>
std::ifstream input("D:/Cryptolab/AES/dump.bin", std::ios::binary);
std::ofstream output("output.txt");
#define in input
#define out output
```

```
#endif
```

接着，我们就需要按顺序来读入相关数据比如：模式选择、密钥内容、明/密文长度、初始向量（IV）、以及明/密文具体内容，由于我们需要采取一次读一轮的操作，具体实现代码在后面会进行详细解释。

```
uint8_t mode;                              // 模式选择
in.read((char *)&mode, sizeof(mode)); // 判断是否是加密还是解密
uint8_t key[16];
in.read((char *)key, sizeof(key));
uint8_t IV[16]; // 初始向量
in.read((char *)IV, sizeof(IV));
uint32_t length; // 明文/密文长度
in.read((char *)&length, sizeof(length));
if (mode == 0x01)
{
    uint32_t *word = new uint32_t[44];
    KeyExpansion(key, word, 4); // 密钥扩展,从word中得到所有的轮密钥
    encrypt(length, word, IV);
    delete[] word;
}
else if (mode == 0x81)
{
    uint32_t *word = new uint32_t[44];
    KeyExpansion(key, word, 4); // 密钥扩展,从word中得到所有的轮密钥
    decrypt(length, word, IV);
    delete[] word;
    // PKCS7Unpadding(data, length); // 去填充
}
else
{
    cout << "模式选择错误" << endl;
}
```

采用上面所示的代码，我们就能够读取做题所需要的全部变量。

- CBC 工作模式的实现
- 加密过程

  由上图所示，对于CBC加密过程而言，我们需要利用读入的初始化向量(IV)先与明文分组异或然后再进行加密，得到的密文分组继续作为下一个明文分组的初始化向量，具体代码如下：

  ```
  // CBC模式加密
  void encrypt(uint32_t length, uint32_t *word, uint8_t *IV)
  {
      //其中，true_len为填充后的明文长度
      //……
      for (int i = 0; i < true_len; i += 16)
      {
          //……
          in.read((char *)&tmp, 16);
          for (int j = 0; j < BLOCKSIZE; j++)
          {
              tmp[j] = tmp[j] ^ IV[j];
          }
  ```

```
        Cipher(tmp, word);
        // 将tmp赋值给IV
        for (int j = 0; j < BLOCKSIZE; j++)
        {
            IV[j] = tmp[j];
        }
        // 直接输出对应值
        out.write((char *)&tmp, sizeof(tmp));
        // PrintDataLE(tmp, 16);
    }
}
```

由以上代码可见，我们总共需要进行的加密轮数为$times = \frac{true_len}{2}$，这里用`i+=16`来代替轮数的更新，然后每一次都仅读入16字节长度的数据（不足16字节会使用填充逻辑），然后与初始向量IV进行异或后，将数据传入到`Cipher`函数中（该函数是具体加密算法的实现）；加密完成后用密文来更新IV，最后直接输入该密文分组。

- 解密过程

  总体解密过程与加密过程相类似，不同点在于我们需要先对密文分组进行解密然后将其与初始向量（IV）进行异或后才能得到对应的明文分组，且我们是**直接将得到的密文分组作为初始向量传到下一组**，具体代码如下：

```
void decrypt(uint32_t length, uint32_t *word, uint8_t *IV)
{
    int times = length / 16; // 解密次数
    uint8_t S_IV[BLOCKSIZE]; // 缓存上一次的 IV
    uint8_t tmp[BLOCKSIZE];  // 当前块数据
    // ……

    for (int i = 0; i < times; i++)
    {
        // 缓存当前的 IV
        memcpy(S_IV, IV, BLOCKSIZE);
        // 读取当前密文块
        in.read((char *)tmp, BLOCKSIZE);
        // 更新 IV 为当前密文块
        memcpy(IV, tmp, BLOCKSIZE);
        // 解密当前块
        InvCipher(tmp, word);
        // CBC 模式：将解密结果与前一块密文（或初始向量）异或
        for (int j = 0; j < BLOCKSIZE; j++)
        {
            tmp[j] ^= S_IV[j];
        }
        // ……
        // 写出解密结果
        out.write((char *)tmp, BLOCKSIZE);
    }
}
```

可以看到这里新创建了一个数组`S_IV`，该数组是用来缓存上一次的IV，然后在后面每次都会将IV更新为当前的密文`tmp`值，其他步骤就跟加密一致，这边便不过多赘述。

- 加密以及解密操作

该操作是本次实验的核心操作，具体的实现流程如下图原理所示：

- 加密

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[0, Nb-1])                  // See Sec. 5.1.4

   for round = 1 step 1 to Nr-1
      SubBytes(state)                              // See Sec. 5.1.1
      ShiftRows(state)                             // See Sec. 5.1.2
      MixColumns(state)                            // See Sec. 5.1.3
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for

   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])

   out = state
end
```

其中，`in`代表输入的明文分组，`out`代表加密后的密文分组，而`word`则是代表着扩展后的密钥内容。首先，我们将初始状态设置为输入的明文分组；然后进行一轮密钥加；紧接着执行一个循环：该循环包括了字符替换、行移位、列混合以及密钥加操作；最后则再重复一遍流程但是唯独少了一个列混合的操作，至此加密过程就完成了，具体代码如下所示：

```
// AES具体加密实现
void Cipher(uint8_t *data, uint32_t *key) // 其中，key是通过密钥扩展得出的轮密钥的集合
{
    // 初始化状态矩阵
    uint8_t state[4][4];
    // 初始化状态矩阵（将输入的data按照列填充到state中）
    for (int i = 0; i < 16; i++)
    {
        state[i % 4][i / 4] = data[i];
    }
    AddRoundKey(state, key, 0);
    for (int i = 1; i < 10; i++)
    {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, key, i);
    }
    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, key, 10);
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            data[i * 4 + j] = state[j][i];
        }
    }
```

```
}
```

接下来分别介绍每个函数：

1. `KeyExpansion`（密钥扩展函数）

   该函数的具体实现逻辑如下图所示：

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word  temp

   i = 0

   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while

   i = Nk

   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end
```

   首先，我们先初始化前NK（NK=4）个轮密钥，前 NK 个密钥轮 `w[0], w[1], ..., w[Nk-1]` 直接从主密钥 `key` 中复制过来；其次，我们在扩展其余的轮密钥，该扩展过程中涉及到了其他一些函数，例如**RotWord**将当前字的字节循环左移一位；**SubWord**将每个字节通过 S-box 进行替换（其中，AES加密方案的S-box是给定的）；**Rcon**轮常量，是一种依赖于轮数 `i` 的固定值，在解密时也同样使用，对于 AES 来说，`Rcon[i/Nk]` 代表第 `i/Nk` 轮的轮常量；最后，我们在`i>=NK`时生成新的轮密钥，具体代码如下所示：

```
uint32_t RotWord(uint32_t word)
{
// 将最高字节移到最低字节位置，其他字节依次左移
uint32_t result = (word << 8) | (word >> 24);
return result;
}
//行移位
void ShiftRows(uint8_t state[4][4])
{
for (int i = 1; i < 4; i++)  // 从第 1 行开始
{
   uint8_t temp[4];  // 用来保存当前行的移位结果
   for (int j = 0; j < 4; j++)
   {
       temp[j] = state[i][(j + i) % 4];  // 使用模运算进行循环移位
   }
   // 将结果写回到状态矩阵
   for (int j = 0; j < 4; j++)
   {
       state[i][j] = temp[j];
   }
```

```
}
}
// 密钥扩展
void KeyExpansion(uint8_t *key, uint32_t *word, int Nk)
{
int Nb = 4;
int Nr = 10;
uint32_t temp;
uint32_t Rcon[11] = {0x00000000, 0x01000000, 0x02000000, 0x04000000,
0x08000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000,
0x1b000000, 0x36000000};
for (int i = 0; i < Nk; i++)
{
   word[i] = (key[4 * i] << 24) | (key[4 * i + 1] << 16) | (key[4 * i
+ 2] << 8) | key[4 * i + 3];
   // cout << hex << word[i] << endl;
}
for (int i = Nk; i < Nb * (Nr + 1); i++)
{
   temp = word[i - 1];
   // cout << hex << temp << endl;
   if (i % Nk == 0)
   {
       temp = SubWord(RotWord(temp)) ^ Rcon[i / Nk];
       // cout << hex << Rcon[i / Nk] << endl;
       // cout << hex << temp << endl;
   }
   else if (Nk > 6 && i % Nk == 4)
   {
       temp = SubWord(temp);
   }
   word[i] = word[i - Nk] ^ temp;
}
}
```

这样，通过这些操作我们就得到了`uint32_t word[44]`扩展后的密钥数组，这些密钥一经计算完成便不会改变，会一直参与剩下次数的加密环节。

2. `AddRoundKey`密钥加函数

密钥加函数的原理就是将当前的轮密钥与状态矩阵进行逐字节的异或，具体的实现原理如下图所示：

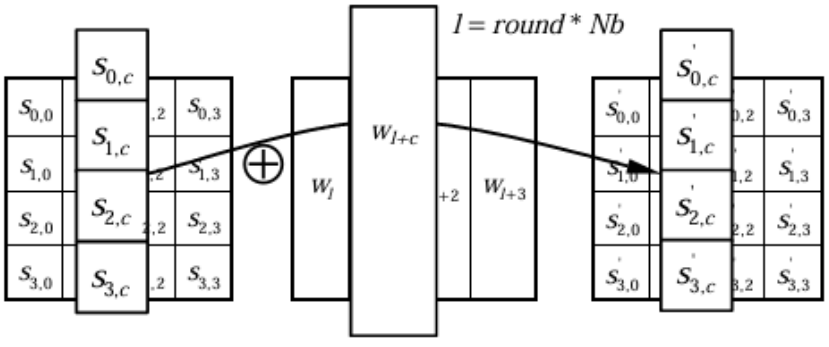


Figure 10. AddRoundKey() **XORs each column of the State with a word from the key schedule.**

> 注意到图中所示的操作是对矩阵的列进行操作，所以在实现该代码时需要比较谨慎的去计算相关数值的坐标，具体代码如下所示：
>
> ```
> void AddRoundKey(uint8_t state[4][4], uint32_t *word, int round)
> {
> // 计算当前轮密钥的起始地址
> uint8_t *roundKey = reinterpret_cast<uint8_t *>(&word[round * 4]);
> // 按列的顺序对 state 矩阵与 roundKey 进行异或
> for (int j = 0; j < 4; j++)
> { // 遍历列
>    for (int i = 0; i < 4; i++)
>    { // 遍历行
>        state[i][j] ^= roundKey[3 - i + j * 4];
>    }
> }
> }
> ```
>
> 由于我们的`word`数组是`uint32_t[44]`，所以每一轮我们都使用该数组中的4个值作为我们本轮的密钥，然后通过自己计算后得到了对于`roundKey`的坐标取值。

3. `SubBytes`字节替换函数

> 字节替换函数有多种实现的方法:第一种可以通过计算得到替换值，而由于在AES中的S盒式是公布的，所以这里采取查表的方式，通过获得该字节的高4位作为行坐标，而字节的低四位作为列坐标然后去给定的S盒查找对应的替换值，这样也可以优化我们的算法时间，具体代码如下所示：
>
> ```
> void SubBytes(uint8_t state[4][4])
> {
> for (int i = 0; i < 4; i++)
> {
>    for (int j = 0; j < 4; j++)
>    {
>        uint8_t byte = state[i][j];
>        uint8_t row = (byte >> 4) & 0x0F;     // 高4位作为行坐标
>        uint8_t col = byte & 0x0F;            // 低4位作为列坐标
>        state[i][j] = s_box[row * 16 + col]; // 在S盒中查找替换
>    }
> }
> }
> ```

4. `ShiftRows`行移位函数

> 行移位函数顾名思义就是对行进行操作：其中，第0行不移动，而第一行左移1位，第二行左移2位，然后以此类推，具体代码如下：
>
> ```
> // 实现行移位
> void ShiftRows(uint8_t state[4][4])
> {
> uint8_t tmp[4][4];
> for (int i = 0; i < 4; i++)
> {
>    for (int j = 0; j < 4; j++)
>    {
>        tmp[i][j] = state[i][j];
> ```

```
    }
}
// 行移位规则
// Row 0: No shift 所以不用操作

// Row 1: Left shift by 1
state[1][0] = tmp[1][1];
state[1][1] = tmp[1][2];
state[1][2] = tmp[1][3];
state[1][3] = tmp[1][0];
// Row 2: Left shift by 2
state[2][0] = tmp[2][2];
state[2][1] = tmp[2][3];
state[2][2] = tmp[2][0];
state[2][3] = tmp[2][1];
// Row 3: Left shift by 3
state[3][0] = tmp[3][3];
state[3][1] = tmp[3][0];
state[3][2] = tmp[3][1];
state[3][3] = tmp[3][2];
}
```

5. `MixColumns`列混合函数

在AES中，`MixColumns`是通过GF(2^8)进行的矩阵乘法。矩阵运算可以增强数据的扩散性，使每个字节对加密结果的影响范围更广，而其中矩阵乘法公式如下：

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

但是，为了节省代码的运算时间，所以先把对应的值都计算好，然后再使用查表的方法来赋值即可，具体代码如下：

```
void MixColumns(uint8_t state[4][4])
{
unsigned char a, b, c, d;

for (int i = 0; i < 4; i++)
{
   a = state[0][i];
   b = state[1][i];
   c = state[2][i];
   d = state[3][i];
   state[0][i] = prodTable[1][a] ^ prodTable[2][b] ^ prodTable[0][c]
^ prodTable[0][d];
   state[1][i] = prodTable[0][a] ^ prodTable[1][b] ^ prodTable[2][c]
^ prodTable[0][d];
   state[2][i] = prodTable[0][a] ^ prodTable[0][b] ^ prodTable[1][c]
^ prodTable[2][d];
   state[3][i] = prodTable[2][a] ^ prodTable[0][b] ^ prodTable[0][c]
^ prodTable[1][d];
}
}
```

6. `Padding`填充函数

在本次实验中，由于要匹配实验的输入输出，所以没有具体实现一个填充函数，而是将填充的逻辑添加到代码的CBC工作模式中。对于加密操作而言，我们的填充操作需要满足以下两个要求：

- 若明文本身刚好是16bytes的倍数，需要填充16个`0x10`
- 若需要填充Nbytes后长度才是16bytes的倍数，此时需要填充N个`0x0N`

具体代码如下：

```
int true_len, pad_len;
if (length % 16 != 0)
{
   true_len = (length / 16 + 1) * 16;
   pad_len = true_len - length;
}
else
{
   true_len = length + 16;
   pad_len = 16;
}
```

首先，我们根据读取到的长度来计算最终需要填充的长度，如果不是16字节的倍数就直接计算出差多少字节将其记为`pad_len`即可。

```
// 最后一组我们需要进行特殊处理
if (i == true_len - 16)
{
   if (length % 16 == 0)
   {
       for (int j = 0; j < 16; j++)
       {
           tmp[j] = 0x10;
       }
   }
   else
   {
       in.read((char *)&tmp, length % 16);
       for (int j = length % 16; j < 16; j++)
       {
           tmp[j] = pad_len;
       }
   }
}
else
{
   in.read((char *)&tmp, 16);
}
```

注意到，我们对于最后一个分组而言需要进行特殊处理，因为主要的填充就是发生在这一部分。如果明文长度本身就是16bytes的倍数，那么就不进行数据读入而直接填充`0x10`即可；如果长度不为16bytes的倍数，那么我们就会先读入剩余的内容，最后在补充上相应填充即可。

至此，加密逻辑都已经实现完毕。

- 解密

```
InvCipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte  state[4,Nb]

   state = in

   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1]) // See Sec. 5.1.4

   for round = Nr-1 step -1 downto 1
      InvShiftRows(state)                    // See Sec. 5.3.1
      InvSubBytes(state)                     // See Sec. 5.3.2
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
      InvMixColumns(state)                   // See Sec. 5.3.3
   end for

   InvShiftRows(state)
   InvSubBytes(state)
   AddRoundKey(state, w[0, Nb-1])

   out = state
end
```

解密操作的步骤与加密操作的步骤大部分相似，只是一些函数需要使用逆函数来求解， 其中，`in`代表输入的密文分组，`out`代表解密后的明文分组，而`word`则是代表着扩展后的密钥内容（这个与加密操作结果相同）。首先，我们将初始状态设置为输入的密文分组；然后进行一轮密钥加（不过选取密钥的顺序并不相同）；紧接着执行一个循环：该循环包括了逆行移位、逆字节替换、密钥加操作以及逆列混合；最后则再重复一遍流程但是仍然少了一个逆列混合的操作，至此解密过程就完成了，具体代码如下所示：

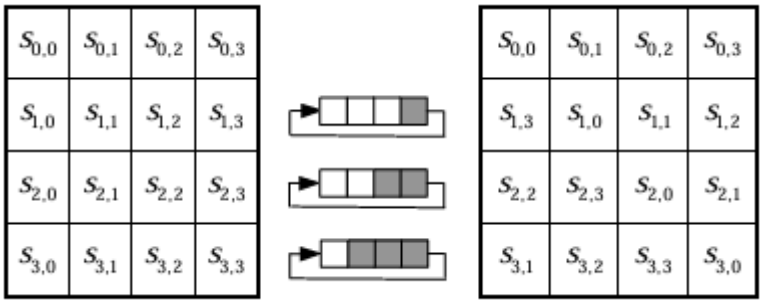
```
void InvCipher(uint8_t *data, uint32_t *key)
{
    // 初始化状态矩阵
    uint8_t state[4][4];
    // 初始化状态矩阵（将输入的data按照列填充到state中）
    for (int i = 0; i < 16; i++)
    {
        state[i % 4][i / 4] = data[i];
    }
    AddRoundKey(state, key, 10);
    for (int i = 9; i > 0; i--)
    {
        InvShiftRows(state);
        `(state);
        AddRoundKey(state, key, i);
        InvMixColumns(state);
    }
    InvShiftRows(state);
    InvSubBytes(state);
    AddRoundKey(state, key, 0);
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4; j++)
        {
            data[i * 4 + j] = state[j][i];
        }
```

```
    }
}
```

接下来介绍一些具体函数：

1. `InvShiftRows` 逆行移位函数：

该函数实现逻辑与加密时的行移动函数的实现方法相同，不过在移位上作出了不同的操作，具体原理如下所示：



代码如下所示：

```
void InvShiftRows(uint8_t state[4][4])
{
uint8_t tmp[4][4];
for (int i = 0; i < 4; i++)
{
   for (int j = 0; j < 4; j++)
   {
       tmp[i][j] = state[i][j];
   }
}
// 行移位规则
// Row 0: No shift 所以不用操作

// Row 1: Right shift by 1
state[1][0] = tmp[1][3];
state[1][1] = tmp[1][0];
state[1][2] = tmp[1][1];
state[1][3] = tmp[1][2];
// Row 2: Right shift by 2
state[2][0] = tmp[2][2];
state[2][1] = tmp[2][3];
state[2][2] = tmp[2][0];
state[2][3] = tmp[2][1];
// Row 3: Right shift by 3
state[3][0] = tmp[3][1];
state[3][1] = tmp[3][2];
state[3][2] = tmp[3][3];
state[3][3] = tmp[3][0];
}
```

2. `InvSubBytes` 逆字节替换函数

该函数也是通过查表得到，具体实现逻辑与加密时相同，这里便不再赘述。

3. `InvMixColumns` 逆列混合函数

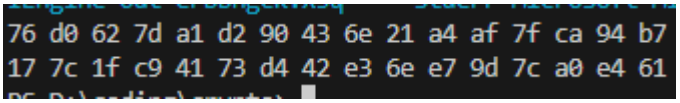
> 该函数也是通过查表得到，具体实现逻辑与加密时相同，这里便不再赘述。

4. `unpadding`去填充函数

> 在本次实验中，由于要匹配实验的输入输出，所以没有具体实现一个去填充函数，而是将去填充的逻辑添加到代码的CBC工作模式中。对于解密操作而言，我们仅需要实现去填充函数即可，具体代码如下所示：
>
> ```
> // 对最后一个分组进行特殊处理
> if (i == times - 1)
> {
> isLastBlock = true;
> for (int j = 0; j < BLOCKSIZE; j++)
> {
>    lastBlock[j] = tmp[j];
> }
> }
> else
> {
> out.write((char *)&tmp, sizeof(tmp));
> }
> // 处理最后一个分组
> if (isLastBlock)
> {
> uint8_t padding = lastBlock[15];
> if (padding < 16)
> {
>    vaildlen -= padding;
>    out.write((char *)&lastBlock, sizeof(lastBlock) - padding);
> }
> ```
>
> 由于我们的去填充操作仅会发生在最后一个分组中，所以我们对最后一个分组进行特殊处理，在处理的时候通过判断其最后一个字节的内容（`lastBlock[15]`）来判断其填充的长度，之后我们再相应的减去这个长度即可；而对于其他请况就直接输出明文分组即可。

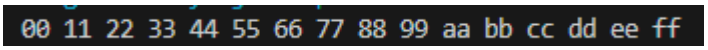
至此解密过程也全部实现。

五、实验结果

> - 加密：
>
>   输入如下所示：
>
>   ```
>   01
>   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
>   00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
>   10 00 00 00
>   00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF
>   ```
>
>   输出结果：
>
>   
>
>
> - 解密：
>
>   输入如下所示：

```
81
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
20 00 00 00
76 D0 62 7D A1 D2 90 43 6E 21 A4 AF 7F CA 94 B7
17 7C 1F C9 41 73 D4 42 E3 6E E7 9D 7C A0 E4 61
```

输出结果：



可以看到逻辑实现完全正确

六、实验总结

通过本次实验，我成功实现了AES-128 CBC工作模式下的运作，对其的理解更加深了一步，并且我也同时学会了分组模式加密的填充以及去填充机制。在本次实验中，本人跟着文档里面给出的参考文献一步一步调试，慢慢的解决掉每个函数的问题，也提高了本人调试的能力以及技巧；同时，通过阅读参考文献也掌握了一些SIMD指令的使用办法，虽然最后优化的效果并不是很好，总归也是掌握了一个新能力。总结来说，本次实验的难度挑战较大，但是再通过查阅资料、询问同学以及自己的努力下也是成功克服了种种问题，是一次收获较大实验。

七、思考题

1. 除了课堂上介绍的五种模式以外，分组加密还有一些其他的工作模式。考虑下面的两个场景，使用什么工作模式可以满足它们的需求？这个工作模式的原理是什么？
   - 你需要检查密文是否被篡改（不需要单独使用 Hash 函数）。
   - 你需要加密一块磁盘。磁盘上的数据是以扇区（通常为 512 Bytes）为单位读写的，操作系统可以通过扇区号随机访问扇区。磁盘加密的密文长度必须要和明文相同，不能使用额外的空间。（有兴趣的话，可以尝试使用一下磁盘加密软件，例如开源的 VeraCrypt）

   答：

   - 在需要检查密文是否被篡改时，可以采用**GCM**模式。它是一种基于计数器模式（Counter Mode, CTR）的加密模式，结合了消息认证代码（MAC）功能。它不仅提供加密功能，还能通过认证标签（Authentication Tag）校验密文是否被篡改。它的工作流程分为三部分：一个是加密过程，在该过程中使用CTR模式对数据进行加密；另一个是认证部分：GCM 使用 Galois 字段上的多项式运算，基于密文和附加认证数据（AAD）计算出认证标签（Tag），而Tag 能够确保密文完整性，任何篡改都会导致解密时校验失败；最后一个就是完整性校验部分：解密时可以通过重新计算并对比认证标签，判断密文是否被篡改。
   - 在进行磁盘加密时，可以采用**XTS**模式。这个模式是为磁盘加密设计的一种分组加密模式。它结合了两次 AES 加密和一个变形（tweak）函数，确保加密块在磁盘上的随机可寻址特性其具体工作方式有以下几点：第一，分块加密：它将每个扇区的数据分成多个 16 字节的块，逐块进行加密；第二，变形操作：根据扇区号和块索引生成一个变形值（tweak），与明文块异或后再加密。这使得即使明文内容相同，不同扇区的密文也完全不同；第三，保持长度不变：加密后的密文与明文长度相同，满足磁盘空间固定的需求；第四，支持随机访问：因为加密块之间独立，每个扇区可以单独加解密，支持随机访问。