武汉大学国家网络安全学院

课程实验报告

分组密码 ZUC

专 业 、 班 ：	信安 9 班
课 程 名 称 ：	密码学实验
指 导 教 师 ：	王后珍
实 验 地 点 ：	C102
学 生 学 号 ：	2021302141097
学 生 姓 名 ：	李宇

2023 年 **12** 月 **22** 日

序列密码 ZUC 的实现

一、 实验目的

1. 掌握序列密码的基本概念；
2. 掌握线性移位寄存器的结构及其序列的伪随机性；
3. 熟悉非线性序列的概念与基本产生方法；
4. 了解常用伪随机性评价方法；
5. 掌握一种典型流密码（如 RC4 或 ZUC 等）。

二、 实验内容及原理

1. 掌握序列密码的实现方案
2. 掌握线性移位寄存器的构造
3. 熟悉序列伪随机性的基本测试方法
4. 实现 RC4 或 ZUC 算法（此处选用 ZUC 算法）
5. 使用本原多项式 $g1(x) = x^4 + x + 1$ 为连接多项式组成线性移位寄存器。画出逻辑图，写出输出序列及状态变迁
6. 使用本原多项式 $g2(x) = x^4 + x^3 + 1$ 为连接多项式组成线性移位寄存器。画出逻辑图，写出输出序列及状态变迁

三、 实验环境

- 操作系统：Windows11 家庭版
- 基本硬件：CPU：AMD Ryzen 5 5600H, 16G 内存
- 所用软件：Pycharm 2023.2.1 专业版
- 所用语言：python3.10

四、 实验步骤与分析

在实验步骤部分，首先给出 ZUC 加密算法的实现流程，具体涉及了**线性反馈移位寄存器**（其中包含了**初始化模式**与**工作模式**）、**比特重组**、**非线性函数**、**密钥装入**的实现。实现的算法以“GM/T 0001-2016 祖冲之序列密码算法”为标准进行实现，实验示例采用课本 171 页给出的示例进行验证。

其次，针对实验报告中要求的两个本原多项式，给出其分析过程与对应的线性移位寄存器。

4.1 ZUC 加密算法实现

ZUC 算法）是由我国学者自主设计的加密和完整性算法，包括祖冲之算法、加密算法 128-EEA3 和完整性算法 128-EIA3，已经被国际组织 3GPP 推荐为 4G 无线通信的第三套国际加密和完整性标准的候选算法。祖冲之算法结构分为三层，第一层是线性反馈移位寄存器 (LFSR)，第二层是比特重组 (BR)，最后一层是非线性函数 F。算法结构如下图所示：

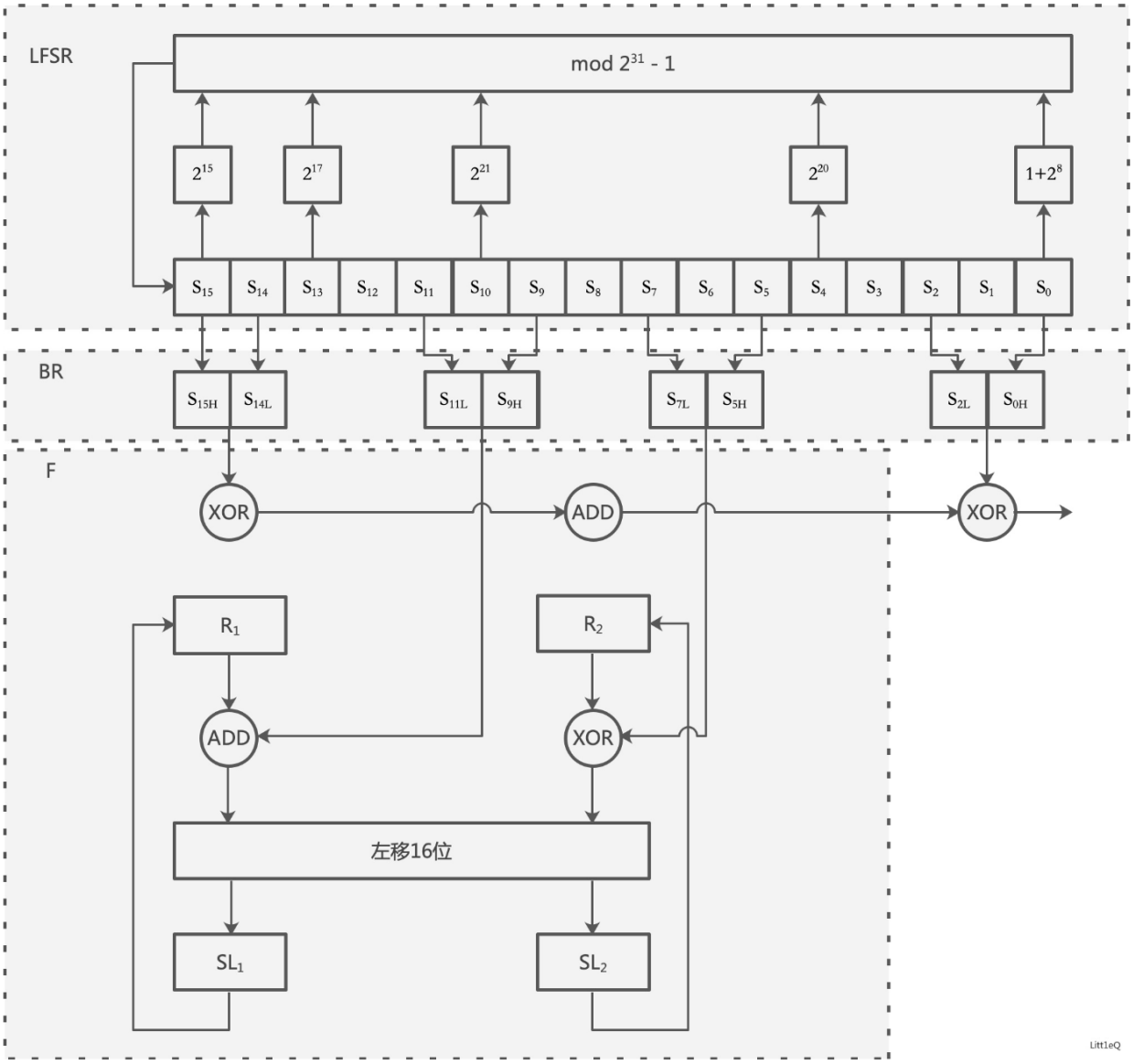

图 1: ZUC 算法结构图

ZUC 算法三层结构逻辑清晰，分别负责完成不同的任务：

1. **第一层：线性反馈移位寄存器（LFSR）**

   线性反馈移位寄存器是祖冲之密码算法的第一层。它是由一系列的比特位组成的寄存器，通过线性的反馈函数进行更新。具体而言，LFSR 的输出会作为下一层的输入，同时它也会根据反馈函

数进行位移和异或操作，以生成新的比特序列。LFSR 的设计使得其产生的比特序列具有伪随机性，增加了密码的安全性。

2. **第二层：比特重组（BR）**

   比特重组是祖冲之密码算法的中间层。它的作用是对来自 LFSR 层的比特序列进行重新排列和组合，以增加密码的复杂性和安全性。比特重组的具体过程包括将输入的比特序列按照特定的规则进行切割、重排和连接，生成新的比特序列。这种重组方式使得攻击者难以通过统计分析等方法来猜测原始的密钥信息。

3. **第三层：非线性函数 F**

   非线性函数 F 是祖冲之密码算法的最后一层。它是一个将输入比特序列映射为输出比特序列的函数，具有非线性的特点。非线性函数的设计使得密码算法能够抵抗各种已知的攻击方法，如差分分析和线性分析等。具体而言，非线性函数 F 会将来自比特重组层的输入进行一系列复杂的运算和操作，最终生成密文的输出。这些运算和操作通常包括模加、异或、置换等，旨在增加密码的混乱和扩散效果，从而提高密码的安全性。

其中线性反馈移位寄存器存在两种运行模式—**初始化模式**和**工作模式**，两者实际上差距不大，不同之处在于是否接收一个 31bit 的参数输入。算法结构的 python 伪代码如下所示，下面给出 python 代码实现的具体过程。

```
# 初始化阶段
for i in range(32):
   BitReconstruction()
   W = F(X_0, X_1, X_2)
   LFSRWithInitializationMode(W >>1)

# 工作阶段
BitReconstruction()
F(X_0, X_1, X_2)
LFSRWithWorkMode()

for i in range(n):
   BitReconstruction()
   z_i = F(X_0, X_1, X_2) ^ X_3
   LFSRWithWorkMode()
```

4.1.1 数据结构定义

为了便利之后算法的实现过程，在这里使用类的方法定义所需的数据结构。在 ZUC 中需要频繁使用到的 16 进制字符串的拼接、相加、异或、取模相加、取 31 位 2 进制字符串的高 16 位、取 31 位 2 进制字符串的低 16 位、循环移位等操作，因此直接定义一个融合了上述功能的字符串类可以让我们之后的字符串处理操作更加便利。

具体定义了一个名为 CustomString 的类，代码如下，各个函数的功能已经在代码注释中给出：

```python
class CustomString:
    def __init__(self, value):
        self.value = str(value)

    def __xor__(self, other):
        # 自定义字符串异或运算
        result = hex(int(self.value, 16) ^ int(str(other), 16))[2:]
        return CustomString(result)

    def __str__(self):
        # 定义字符串对象的打印形式
        return self.value

    def __add__(self, other):
        # 重构加法运算符为字符串首尾拼接
        result = self.value + str(other)
        return CustomString(result)

    def mod_pow_2_32_add(self, other):
        # 两个字符串对应的值相加，并模2**32
        t=int(self.value, 16) + int(str(other), 16)
        result = (int(self.value, 16) + int(str(other),16)) % (2 ** 32)
        return CustomString(hex(result)[2:])

    def low_bits(self):
        # 取出一个2进制字符串的低16位（即16进制字符串低4位）
        t = self.value[-4:]
        return CustomString(self.value[-4:])

    def high_bits(self):
        # 取出一个2进制字符串的高16位（即16进制字符串高4位）
        binary_result = bin(int(self.value, 16))[2:].zfill(31)[:16]
        hex_result = hex(int(binary_result, 2))[2:]
        return CustomString(hex_result)

    def process_other_string(self, other_str):
        # 处理另一个字符串
        other_value = int(str(other_str.value[-1]), 16)
        flag = other_value % 2
        temp = self.value
        temp = hex(int(temp[0], 16) + flag * 8)[2:] + temp[1:]

```

```
        other_result = other_str.right_shift(1)
        return CustomString(other_result.value + temp)

    def rotate_left(self, x):
        # 循环左移 x 位
        value_int = int(self.value, 16)
        rotated_value = ((value_int «x) | (value_int »(32 - x))) & 0xFFFFFFFF
        result = hex(rotated_value)[2:]
        return CustomString(result)

    def right_shift(self, x):
        # 右移 x 位
        result = hex(int(self.value, 16) »x)[2:]
        return CustomString(result)

    def drop_lowest_bit_and_to_str(self):
        # 舍弃字符串的二进制下最低位，然后转为字符串输出--获得31比特字u
        value_int = int(self.value, 16)
        result = hex(value_int »1)[2:]
        return CustomString(result)
```

可以看到，针对 CustomString 字符串类，异或运算符和加法运算符均被重载；同时定义了各种字符串之间的操作。另外，由于 ZUC 的实现过程中多次出现了 **31 位比特字**，不同于 32 位比特字（8 字节），31 位比特字的处理要稍加麻烦，需要将其转转化为真实的 2 进制数值，再舍去最高位/最低位后，再转化为 16 进制字符串。

完成定义所需的数据类型后，下面给出伪代码中各个函数的具体实现，顺序按照算法执行流程给出。

4.1.2 密钥装入

在 ZUC 算法中，密钥装入过程将**初始密钥**和**初始向量**扩展为多个比特，作为**线性反馈移位寄存器（LFSR）的初始状态**。这个过程的具体步骤如下：

1. 引入一个 240 比特的常量 D，将其分为 16 个 15 比特的子串。
2. 将每个子串转换为 8 比特字节，然后将这些字节拼接起来，形成密钥。
3. 将初始向量也扩展为与密钥相同长度的比特序列。
4. 将密钥和初始向量按照一定的方式进行拼接，形成 LFSR 寄存器单元变量 s_0,s_1,⋯,s_{15} 的初始状态。

通过密钥装入，ZUC 算法能够从一个可靠的起始状态开始运行，并产生一系列的加密比特序列。这些加密比特序列随后会被用于数据的加密和解密过程，以确保数据的机密性和完整性。

密钥装入过程，将会把初始密钥 k(128 bits) 和 iv(128 bits) 扩展成 16 个 31 位整数，作为 LFSR 的初始状态。$k = k_0 \| k_1 \| k_2 \| \ldots \| k_{15}$、$iv = iv_0 \| iv_1 \| \ldots \| iv_{15}$，这里的 k_i 和 iv_i 均为 8 位的字节，然后构造 D，D 为 240 比特的常量，可以按照如下方式分成 16 个 **15 位**的子串（注意是 15 位）：

$$\begin{aligned}
d_0 &= 100010011010111_2 \\
d_1 &= 010011010111100_2 \\
d_2 &= 110001001101011_2 \\
d_3 &= 001001101011110_2 \\
d_4 &= 101011110001001_2 \\
d_5 &= 011010111100010_2 \\
d_6 &= 111000100110101_2 \\
d_7 &= 000100110101111_2 \\
d_8 &= 100110101111000_2 \\
d_9 &= 010111100010011_2 \\
d_{10} &= 110101111000100_2 \\
d_{11} &= 001101011110001_2
\end{aligned} \tag{1}$$

那么 $D = d_0 \| d_1 \| \ldots \| d_{15}$。对于 s_i，有 $s_i = k_i \| d_i \| iv_i$，其中 || 代表两个字符首尾拼接。代码如下所示：

```
def init_S(key, IV):
    d = [CustomString("44D7"), CustomString("26BC"), CustomString("626B"),
        CustomString("135E"),
        CustomString("5789"), CustomString("35E2"), CustomString("7135"),
            CustomString("09AF"),
        CustomString("4D78"), CustomString("2F13"), CustomString("6BC4"),
            CustomString("1AF1"),
        CustomString("5E26"), CustomString("3C4D"), CustomString("789A"),
            CustomString("47AC")]

    t = [d[i] + IV[i] for i in range(16)]
    return [t[i].process_other_string(key[i]) for i in range(16)]
```

可以看到，通过之前定义好的类，字符串之间的操作变得十分简单。

4.1.3 比特重组 BitReconstruction

中间层是一个比特重组算法。它从 LFSR 中提取出来 128 比特，然后组成 4 个 32 位的字 X_0, X_1, X_2, X_3，前三个字是提供给底层非线性 F 函数使用的，最后一个字用来生成密钥流。重组后的 X_0, X_1, X_2, X_3，

如下所示：

$$
\begin{aligned}
X_0 &= s_{15H} \| s_{14L} \\
X_1 &= s_{11L} \| s_{9H} \\
X_2 &= s_{7L} \| s_{5H} \\
X_3 &= s_{2L} \| s_{0H}
\end{aligned} \tag{2}
$$

注意：这里的 s_i 是 *31* 比特的整数类型，所以 s_{iH} 表示的是 s_i 的 $30 \ldots 15$ 位而不是 $31 \ldots 16$ 位。

示意图如下所示：

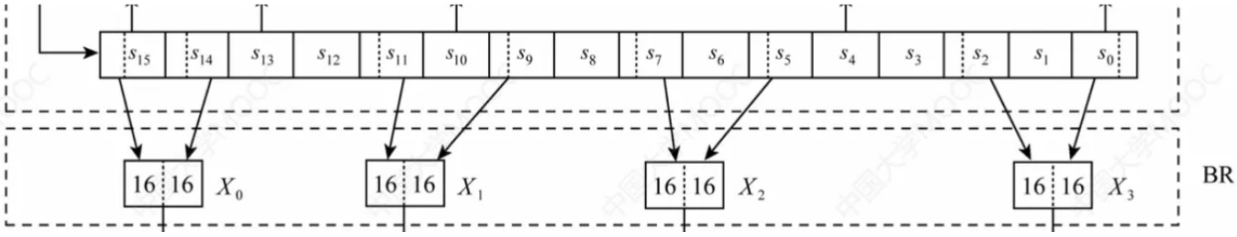

图 2: 比特重组

实现代码如下所示：

```
def BitReconstruction(S):
    x0 = S[15].high_bits() + S[14].low_bits()
    x1 = S[11].low_bits() + S[9].high_bits()
    x2 = S[7].low_bits() + S[5].high_bits()
    x3 = S[2].low_bits() + S[0].high_bits()
    return x0, x1, x2, x3
```

4.1.4 非线性函数 F

非线性函数 F 有两个 32 比特的记忆单元 R_1 和 R_2，输入为 3 个 32 比特的字：X_0, X_1, X_2，输出一个 32 比特的字 W。因此，非线性函数是一个把 96 比特压缩为 32 比特的一个非线性压缩函数。具体描述如下，其中 $\boxplus$ 为模 2^{32} 的加法：

$$
\begin{aligned}
W &= (X_0 \oplus R_1) \boxplus R_2 \\
W_1 &= (R_1 \boxplus X_1) \\
W_2 &= (R_2 \oplus X_2) \\
R_1 &= S\left(L_1\left(W_{1L} \| W_{2H}\right)\right) \\
R_2 &= S\left(L_2\left(W_{2L} \| W_{1H}\right)\right)
\end{aligned} \tag{3}
$$

非线性函数的示意图如下所示：

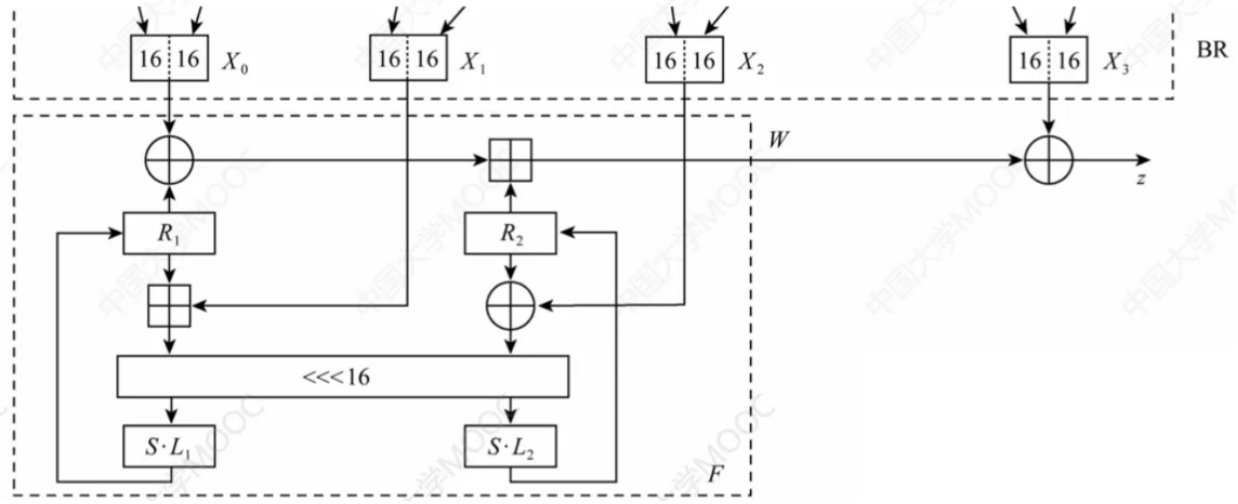

图 3: 非线性函数示意图

在代码实现上，异或运算、模加已经在类中实现了，重点是循环移位 L1,L2 以及 S 盒变换的实现。循环移位已经在之前的实验中多次实现，L1,L2 代码如下；

```
def L1(string):
   return string^string.rotate_left(2)^string.rotate_left(10)^ \
         string.rotate_left(18)^string.rotate_left(24)

def L2(string):
   return string^string.rotate_left(8)^string.rotate_left(14)^ \
      string.rotate_left(22)^string.rotate_left(30)
```

S 盒变换代码如下所示，需要注意：要确保 S 盒的输出的长度为 2，即 9 需要表示为 09，这样才能在之后的运算中不出错，代码中的体现即 $zfill(2)$。

```
def S_transform(word):
   word=CustomString(word.value.zfill(8))
   string=word.value
   s0=string[:2]
   s1=string[2:4]
   s2=string[4:6]
   s3=string[6:8]
   t1=s_box_0(s0)
   t2=s_box_1(s1)
   t3=s_box_0(s2)
   t4=s_box_1(s3)
   return t1+t2+t3+t4

def s_box_0(s):
```

```
    v=int(s,16)
    return CustomString(hex(S0[v//16][v%16])[2:].zfill(2))

def s_box_1(s):
    v = int(s, 16)
    return CustomString(hex(S1[v // 16][v % 16])[2:].zfill(2))
```

F 函数的整体代码如下：

```
def F(x0, x1, x2, R1, R2):
    t=x0 ^ R1
    W = (t).mod_pow_2_32_add(R2)
    W1 = ((R1.mod_pow_2_32_add(x1)).value).zfill(8)
    W2 = ((R2^(x2)).value).zfill(8)
    R1 = S_transform(L1(CustomString(W1[-4:]+W2[:4])))
    R2 = S_transform(L2(CustomString(W2[-4:]+W1[:4])))
    return W, R1, R2
```

4.1.5 LSFR 阶段 $LFSRWithInitializationMode(w >> 1)$

在初始化模式中，LFSR 接收一个 31 比特字 u, u 是由非线性函数 F 的 32 比特输出 W 通过舍弃最低位比特得到。在初始化模式下，LSFR 计算过程如下：

$$
\begin{aligned}
&v = 2^{15}s_{15} + 2^{17}s_{13} + 2^{21}s_{10} + 2^{20}s_4 + \left(1 + 2^8\right) s_0 \bmod \left(2^{31} - 1\right) \\
&s_{16} = (v + u) \bmod \left(2^{31} - 1\right) \\
&if\ s_{16} = 0,\ then\ set\ s_{16} = 2^{31} - 1 \\
&(s_1, s_2, \ldots s_{16}) \rightarrow (s_0, s_1, \ldots, s_{15})
\end{aligned}
\tag{4}
$$

根据上述过程，实现代码如下：

```
def LFSRWithInitialisationMode(W,S):
    v=t_2_15*int(S[15].value,16) + t_2_17*int(S[13].value,16)+
        t_2_21*int(S[10].value,16)
        +t_2_20*int(S[4].value,16)+(t_2_8+1)*int(S[0].value,16)
    v=v%(t_2_31-1)
    s16=(int((W.drop_lowest_bit_and_to_str()).value,16)+v)%(t_2_31-1)
    if s16==0:
       s16=(t_2_31-1)
    S=S[1:]
    S.append(CustomString(hex(s16)[2:]))

    return S
```

在这里对 2^31-1 运算均是采用的直接计算的方式，很显然对于此类较大数值的计算采用这种方式是低效的。分析模乘，模加过程，一个数 X 与 2^y 相乘，并模 2^31-1。可以考虑如下两种情况：

- **乘积结果小于模数** 2^31-1

  此时运算结果即 x 左移 y 位。

- **乘积结果大于等于模数** 2^31-1

  可以先考虑模数为 2^31 的情况，此时相当于一旦乘积的第 32 位为 1 了之后就将此位置为 0，实际上也就是当乘积大于模数后减去 2^31；再考虑模数为 2^31-1 的情况，相当于于一旦乘积的第 32 位为 1 了之后就将此位置为 0 并在最低位上 +1。实际上的效果就是 x 在 31 位上循环左移 y 位，超出的高位补在低位上，可以发现乘积结果小于模数 2^31-1 也符合这样的结论。

因此可以把上述低效的大数相乘再取模的运算转换为高效的循环移位运算，代码如下所示：

```
t1=S[15].rotate_left_31(15)
t2=S[13].rotate_left_31(17)
t3=S[10].rotate_left_31(21)
t4=S[4].rotate_left_31(20)
t5=(S[0].rotate_left_31(8)).mod_pow_2_31_add(S[0])
v=(((t1.mod_pow_2_31_add(t2)).mod_pow_2_31_add(t3)).
mod_pow_2_31_add(t4)).mod_pow_2_31_add(t5)
s16=int(((W.drop_lowest_bit_and_to_str()).mod_pow_2_31_add(v)).value,16)
```

其中 rotate_left_31 为循环移位函数（31）位，同时将取结果的前 31 位，代码如下所示：

```
def rotate_left_31(self, x):
    # 循环左移 x 位
    value_int = int(self.value, 16)
    rotated_value = ((value_int «x) | (value_int »(31 - x))) & 0x7FFFFFFF
    result = hex(rotated_value)[2:]
    return CustomString(result)
```

4.1.6 LSFR 工作阶段 $LFSRWithWorkMode$

工作阶段相比初始化阶段没有太大的改变，只是简化掉了输入 u 的过程，计算过程如下：

$$
\begin{aligned}
&s_{16}=2^{15}s_{15}+2^{17}s_{13}+2^{21}s_{10}+2^{20}s_4+\left(1+2^8\right)s_0 \bmod \left(2^{31}-1\right)\\
&if\ s_{16}=0,\ then\ set\ s_{16}=2^{31}-1\\
&\left(s_1,s_2,\ldots s_{16}\right)\rightarrow\left(s_0,s_1,\ldots,s_{15}\right)
\end{aligned}
\tag{5}
$$

代码也改动不大，如下所示：

```
def LFSRWithWorkMode(S):
  # v=t_2_15*int(S[15].value,16) + t_2_17*int(S[13].value,16)+
     t_2_21*int(S[10].value,16)
     +t_2_20*int(S[4].value,16)+(t_2_8+1)*int(S[0].value,16)
```

```
    # v=v%(t_2_31-1)

    t1 = S[15].rotate_left_31(15)
    t2 = S[13].rotate_left_31(17)
    t3 = S[10].rotate_left_31(21)
    t4 = S[4].rotate_left_31(20)
    t5 = (S[0].rotate_left_31(8)).mod_pow_2_31_add(S[0])
    v =
        int(((((t1.mod_pow_2_31_add(t2)).mod_pow_2_31_add(t3)).mod_pow_2_31_add(t4)).
    mod_pow_2_31_add(t5)).value,16)

    s16=v
    if s16==0:
       s16=(t_2_31-1)
    S=S[1:]
    S.append(CustomString(hex(s16)[2:]))

    return S
```

4.1.7 ZUC 顶层代码

完成了所有部分后，先经过初始化阶段（循环 32 次），再进入工作阶段，代码组织如下所示：

```
S=init_S(key,IV)
disp(S)
for i in range(32):
    x0,x1,x2,x3=BitReconstruction(S)
    W,R1,R2=F(x0,x1,x2,R1,R2)
    S=LFSRWithInitialisationMode(W,S)

for i in range(3):
    x0, x1, x2, x3 = BitReconstruction(S)
    W, R1, R2 = F(x0, x1, x2, R1, R2)
    S = LFSRWithWorkMode(S)
    z=W^x3
    print("密钥流：Z_{}：".format(i),z)
```

五、 实验结果与总结

5.1 实验结果

在此部分执行编写好的代码，以课本 171 页为例，密钥 k 与初始向量 IV 如下图所示：执行代码，

```
key = [CustomString("3d"), CustomString("4c"), CustomString("4b"), CustomString("e9")
    , CustomString("6a"), CustomString("82"), CustomString("fd"), CustomString("ae")
    , CustomString("b5"), CustomString("8f"), CustomString("64"), CustomString("1d")
    , CustomString("b1"), CustomString("7b"), CustomString("45"), CustomString("5b")]

IV = [CustomString("84"), CustomString("31"), CustomString("9a"), CustomString("a8")
    , CustomString("de"), CustomString("69"), CustomString("15"), CustomString("ca")
    , CustomString("1f"), CustomString("6b"), CustomString("da"), CustomString("6b")
    , CustomString("fb"), CustomString("d8"), CustomString("c7"), CustomString("66")]

R1 = CustomString("0")
R2 = CustomString("0")
```

图 4: 算法输入

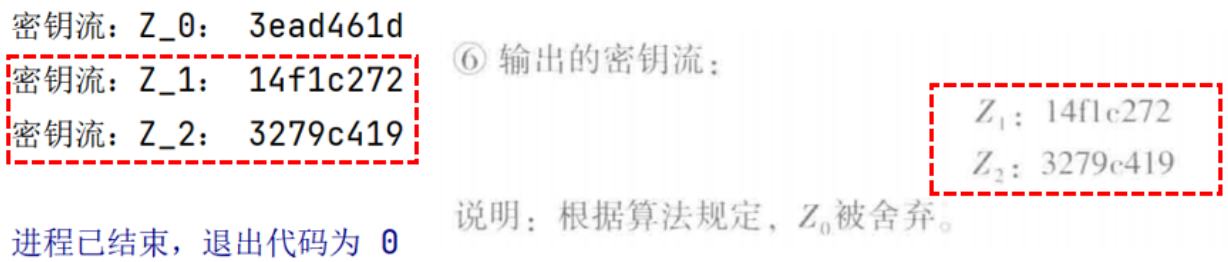

图 5: 执行结果

最终输出的密钥流：可以看到与课本结果对比结果完全一致，说明代码成功实行了 ZUC 算法。复现代码已经上传至 *Github* 之中：祖冲之算法 *python* 复现代码

5.2 本原多项式 1 分析

$g1(x) = x^4 + x + 1$ 对应的线性移位寄存器如下图所示：

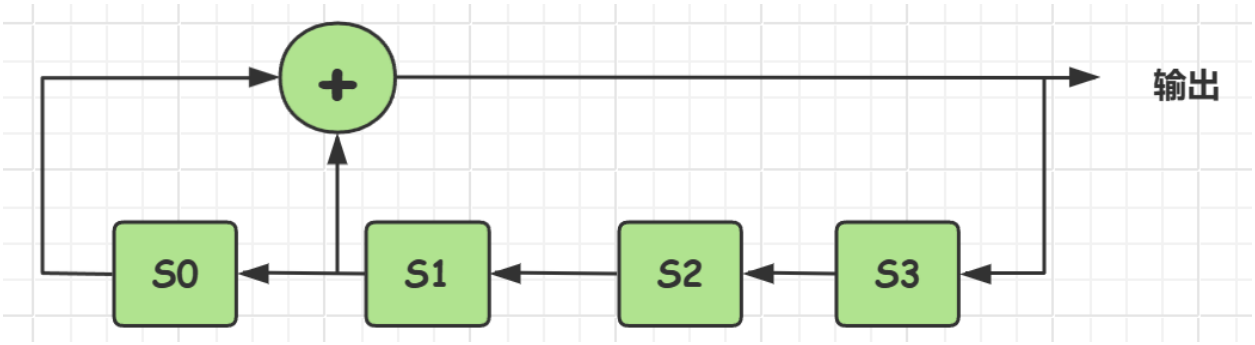

图 6: g1 对应的线性移位寄存器

输出序列为：*0011010111110001....*

状态变迁 (S0,S1,S2,S3)：*0001->0010->0100->1001->0011->0110->1101->1010->0101->1011->0111->1111->1110->1100->1000->0001*，周期为 15.

5.3 本原多项式 2 分析

$g2(x) = x^4 + x^3 + 1$ 对应的线性移位寄存器如下图所示：

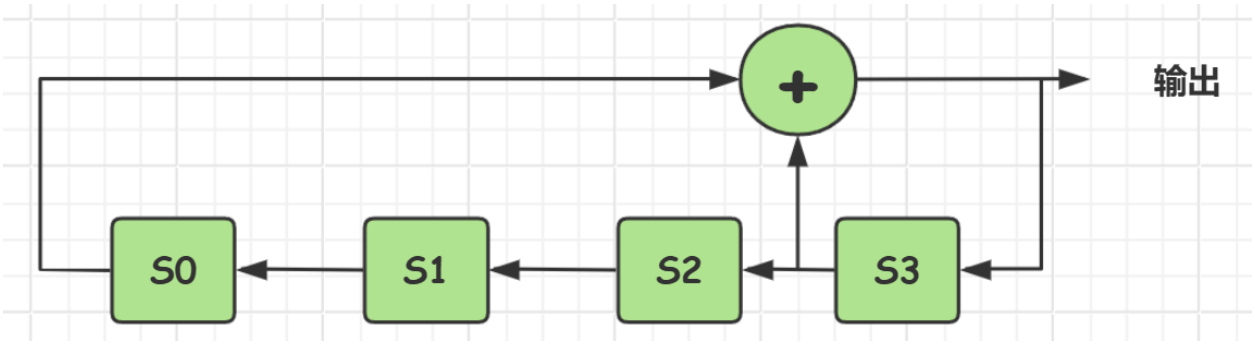

图 7: g2 对应的线性移位寄存器

输出序列为：*111010110010001....*

状态变迁 (S0,S1,S2,S3)：*0001->0011->0111->1111->1110->1101->1010->0101->1011->0110->1100->1001->0010->0100->1000->0001*，周期为 15.

5.4 个人收获

通过此次实验，我学习了序列密码的概念与原理；序列密码的核心即如何接近伪随机的生成密钥序列，而实际的加密解密过程只需要通过简单的异或运算即可完成。在代码实现中，注意到了许多看书遗漏的细节，如 zuc 算法中 31 位数据的处理，一开始也没有注意到密钥装入时常量 d 的比特位数，通过代码实现捋清了很多细节，最后也复习了本原多项式，加深了我对序列密码的理解。

参考文献

[1] 祖冲之加密算法详解及代码实现（csdn）：http://t.csdnimg.cn/d1JdW

[2] ZUC 祖冲之序列密码算法（博客园）：https://www.cnblogs.com/mengsuenyan/p/13819504

[3] GM/T 0001-2012 祖冲之序列密码算法

[4]《密码学引论》第三版

教师评语评分

评语:

评　　分:

评 阅 人:

评阅时间: