《密码学》课程设计实验报告

***本次实验源文件和实验报告均已上传至 ZUC 文件夹下：**

sterzhang/Cryptology: Jianshu Zhang's cryptological experiments (github.com)

实验序号：04　　实验项目名称：序列密码

学　　号	2021302181216	姓　　名	张鉴殊	专业、班	21 信安
实验地点	C202	指导教师	余荣威	时间	2022.12.08

一、　实验目的及要求

教学目的：

(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）。

实验要求：

(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。

二、实验设备（环境）及要求

Windows 操作系统，高级语言开发环境

三、实验内容与步骤

1. 序列密码的实现方案

（这一部分的具体实现放在后面“编程实现 ZUC 算法”部分，这里以 ZUC 作为例子讲解理论过程）

（1）种子密钥 K 输入到密钥流发生器；

初始化密钥和向量：ZUC 算法接收一个 128 位的初始密钥 K 和 128 位的初始化向量 IV。这两个作为输入共同作为算法的初始状态。

初始混淆：使用 K 和 IV，算法执行一系列操作来初始化内部状态。这个过程包括初始化线性反馈移位寄存器（LFSR）等。

（2）产生一系列密钥流；

状态更新：ZUC 算法使用一个基于 LFSR 和非线性函数的复杂机制来更新其内部状态。LFSR 负责产生一系列伪随机数，而非线性函数进一步增强了输出的随机性。

生成密钥流：算法在每个时钟周期产生一个或多个密钥流位。这些位是通过对内部状态执行一系列的数学运算生成的。

（3）通过与同一时刻的一个字节或者一位明文流进行异或操作产生密文流。

加密/解密：生成的密钥流与明文（或密文）数据逐位（或逐字节）进行异或操作。这个过程可以用于加密（将明文转换为密文）或解密（将密文还原为明文）。

流处理：由于是流密码，ZUC 算法适用于逐字节或逐位处理数据流，使得它在处理长数据流时特别有效。

2．线性移位寄存器的构造

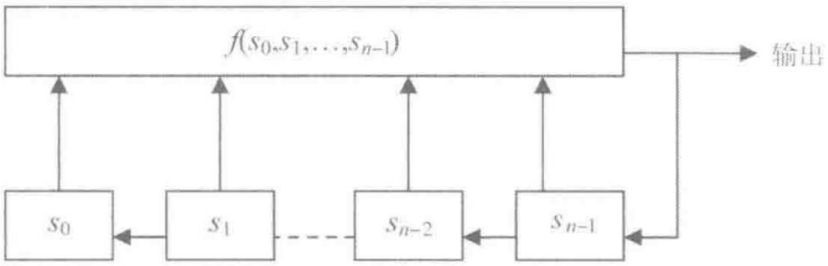


图 5-3　移位寄存器

（1）选择连接多项式（一般选择本原多项式作为连接多项式）

（2）根据连接多项式的反馈系数得出反馈函数

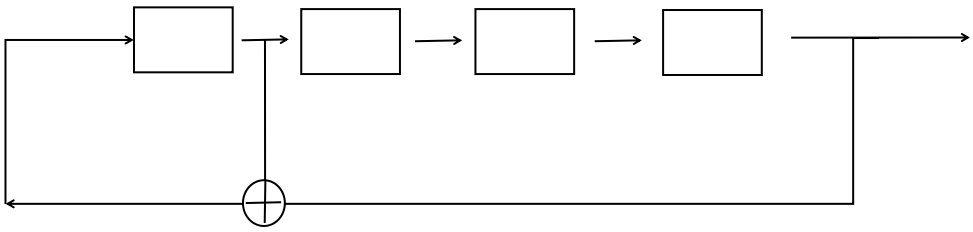
（3）根据反馈函数得出每个节拍的寄存器状态

（必做题：实验 1-2）

这里以 1101 作为初始序列：

```
state = 0b1101
```

实验（1）使用本原多项式 $g_1(x)=x^4+x+1$ 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（在下面图片的终端打印结果中）。

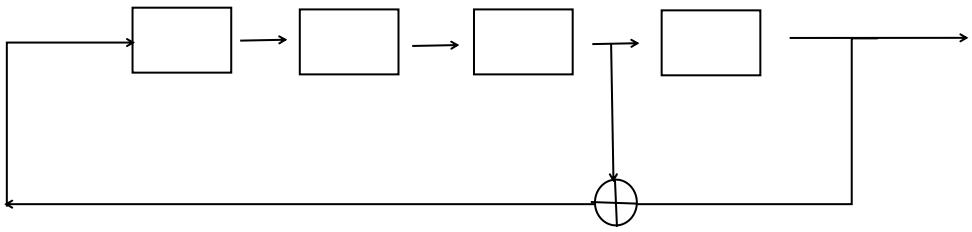


按照如下步骤：

1. 初始化寄存器 state 选择一个非零的初始状态 1101
2. 计算反馈位：将寄存器的第 4 位与第 1 位进行 XOR 操作
3. 得到新的 state：将 state 向右移动一位，将计算出的反馈位放入最左边的位置

```
def lfsr_step_1(state):
    # 计算反馈位
    feedback = ((state >> 3) ^ (state >> 0)) & 1
    # 右移寄存器并把反馈放到高位上
    state = (feedback << 3) | (state >> 1)
    # 确保寄存器仍然是 4 位
    return state & 0b1111
```

实验（2）使用本原多项式 $g_2(x)=x^4+x^3+1$ 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。



按照如下步骤：

1. 初始化寄存器 state 选择一个非零的初始状态 1101
2. 计算反馈位：将寄存器的第 4 位和第 3 位进行 XOR 操作

3. 得到新的 state：将 state 向右移动一位，将计算出的反馈位放入最左边的位置

```
def lfsr_step_2(state):
    # 计算反馈位
    feedback = ((state >> 3) ^ (state >> 2)) & 1
    # 右移寄存器并把反馈放到高位上
    state = (feedback << 3) | (state >> 1)
    # 确保寄存器仍然是 4 位
    return state & 0b1111
```

现在来打印他们分别的输出并计算出他们的周期：

```
print("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: ")
for i in range(100):
    state = lfsr_step_1(state)
    if state == first_state and i > 0:
        print(f"周期为{i}")
        break
    print(f'{state:04b}')

print("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")
for i in range(100):
    state = lfsr_step_2(state)
    if state == first_state and i > 0:
        print(f"周期为{i}")
        break
    print(f'{state:04b}')
```

结果如下：

```
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:
1101
0110
0011
1001
0100
0010
0001
1000
1100
1110
1111
0111
1011
0101
1010
周期为15
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
1101
0110
1011
周期为3
```

试对比以上两组输出序列的关系

提示：（1）与（2）中的多项式是互反多项式，所谓互反多项式是指 f(x)与 $x^n f(\frac{1}{x})$

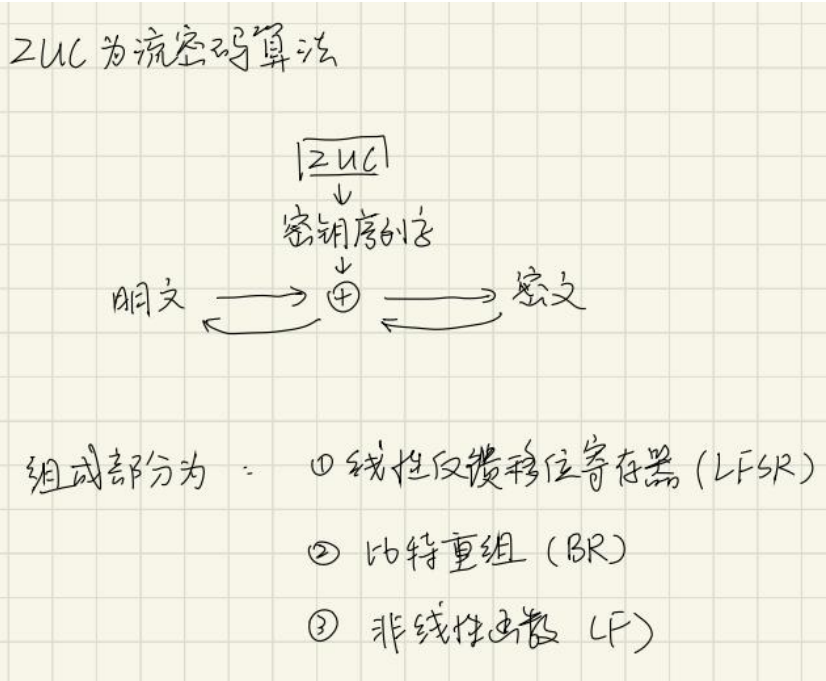
$g_1(x)=x^4+x+1$ 和 $g_2(x)=x^4+x^3+1$ 为互反多项式，其中一个多项式的系数序列是另一个多项式的系数序列的逆序。

在理想情况的 state 中，如果一个 LFSR 产生一个序列，另一个会产生第一个序列的反向。这是因为互反多项式在位级别上以相反的顺序执行反馈操作。在这里这种逆序关系可能不会在所有情况下完全显现，因为还受 state 初始状态的影响。

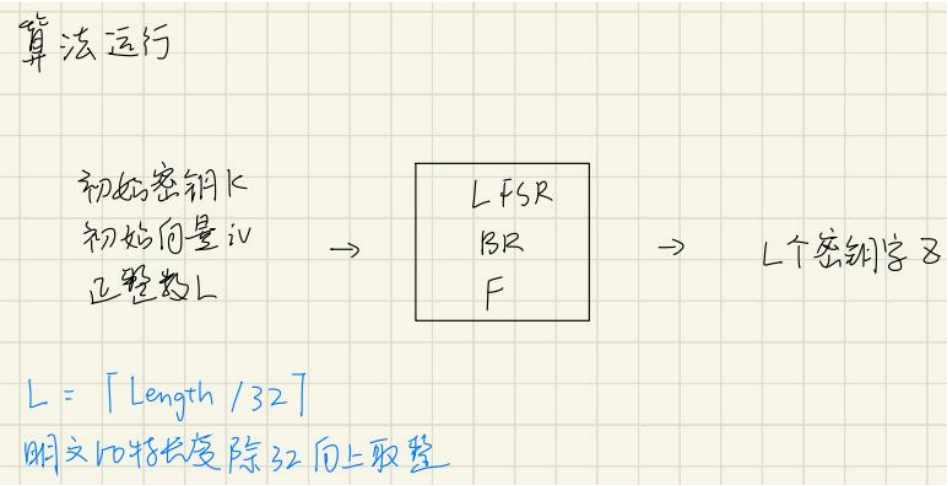
虽然这两个 LFSR 的周期可能不完全相同，但由于互反多项式的特性通常会产生具有相似特征的序列。在理想情况下，如果使用的多项式是本原的，它们都能达到最大长度的周期序列，这里是 15。

第二种情况序列进入了一个短循环。这表明在这个特定初始状态下，它没有达到最大周期，说明所选的初始状态与该多项式的特定组合会导致较短的循环。

3. 编程实现 ZUC 算法

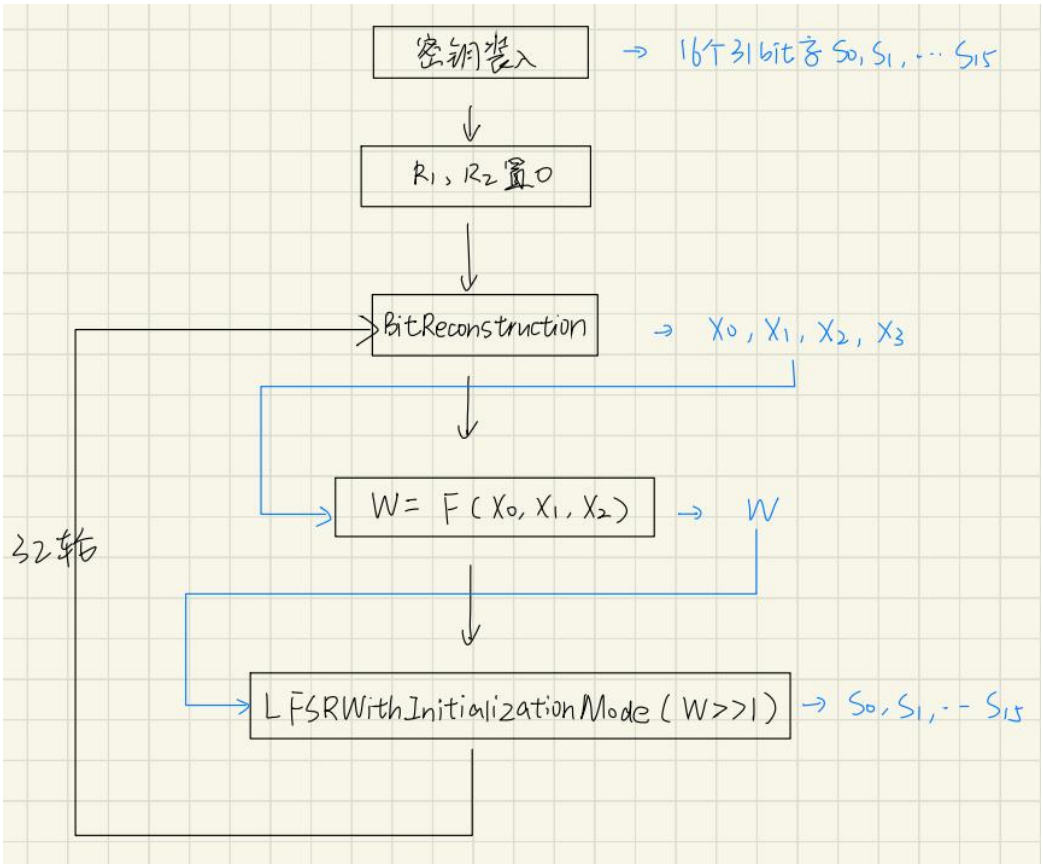


整个算法的运行过程可以抽象为：



ZUC 分为两个阶段，第一个阶段是初始化阶段，代码部分及流程图如下：

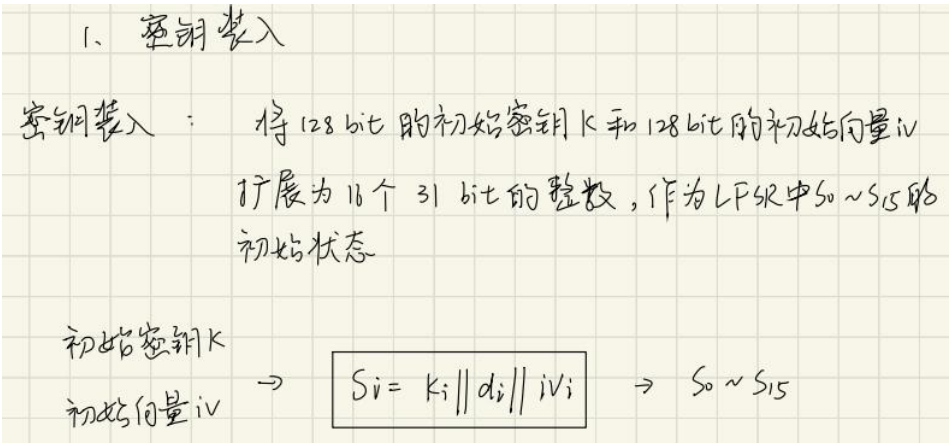
```
class ZUC:
    #整体流程，经历两个阶段，初始化阶段和工作阶段
    def __init__(self, key, iv):
        self.initialize(key, iv)
```



初始化部分整体代码如下：

```
# ZUC 算法第一个阶段是初始化：分为 1.密钥装入 2.将 F 中的 R1，R2 置为 0 3.执行 32 次循环
    def initialize(self, key, iv):
        # 1.密钥装入：将初始密钥 key 和初始向量 iv 作为输入，与 d 进行拼接，得到 s0~s15
        self.s = []
        for i in range(16):
            self.s.append(key[i*8:i*8+8] + int2ba(zuc.d[i], length=15) + iv[i*8:i*8+8])
        # 2.将 F 中的 R1，R2 置为 0
        self.r1 = self.r2 = int2ba(0, length=32)
        # 3.执行 32 次循环（1.比特重组 2.非线性函数输出 W 3.LSFR 初始化）
        for i in range(32):
            # 1.比特重组（输出 X0，X1，X2，X3）
            self.BitReconstruction()
            # 2.将上一步得到的 X0，X1，X2，X3 作为输入传给 F 函数，得到 W
            self.F()
            # 3.将上一步得到的 W 送进 LSFR 初始化模式得到 S0~S15
            self.LFSRWithInitializationMode()
```

1）密钥装入：将初始密钥 key 和初始向量 iv 作为输入，与 d 进行拼接，得到 s0~s15：



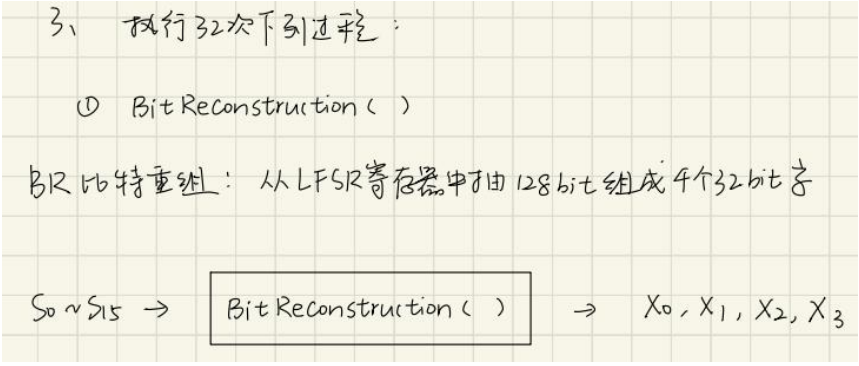
代码如下：

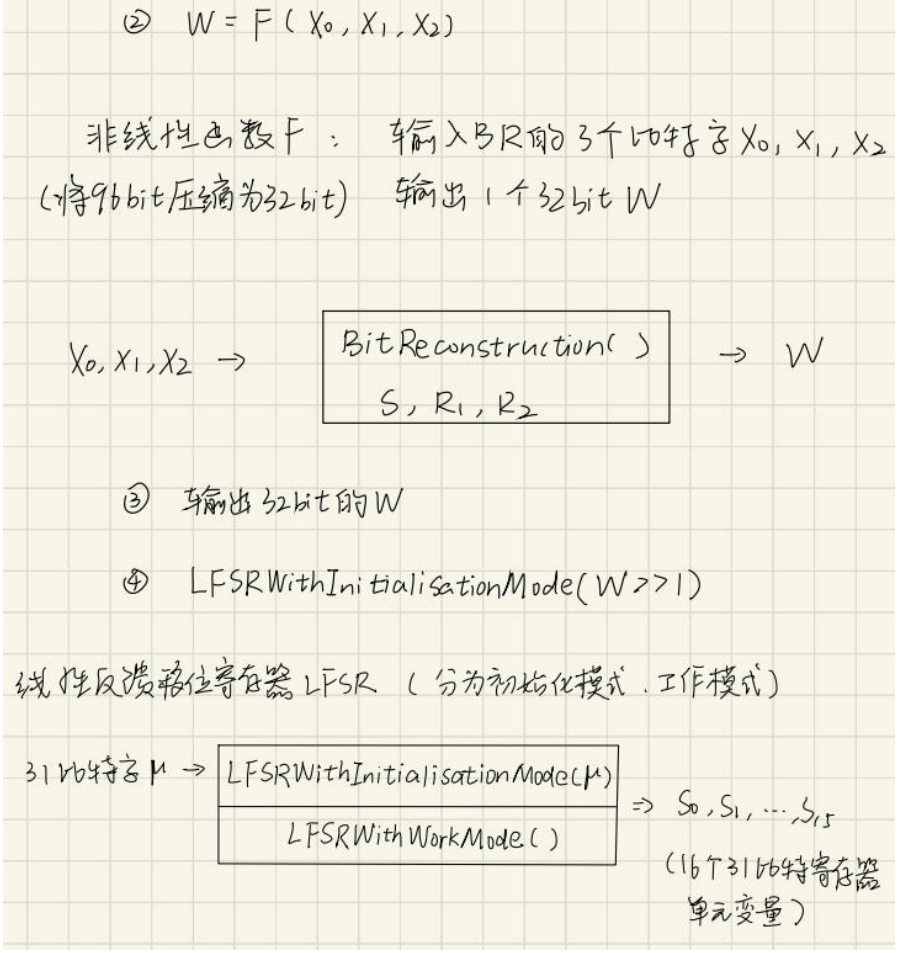
```
        # 1.密钥装入:将初始密钥 key 和初始向量 iv 作为输入,与 d 进行拼接,得到 s0~s15
        self.s = []
        for i in range(16):
            self.s.append(key[i*8:i*8+8] + int2ba(zuc.d[i], length=15) +
iv[i*8:i*8+8])
```

2）将 F 中的 R1，R2 置为 0：

```
        # 2.将 F 中的 R1，R2 置为 0
        self.r1 = self.r2 = int2ba(0, length=32)
        # 3.执行 32 次循环（1.比特重组 2.非线性函数输出 W 3.LSFR 初始化）
        for i in range(32):
            # 1.比特重组（输出 X0，X1，X2，X3）
            self.BitReconstruction()
            # 2.将上一步得到的 X0，X1，X2，X3 作为输入传给 F 函数，得到 W
            self.F()
```

3）执行 32 次循环（1.比特重组 2.非线性函数输出 W 3.LSFR 初始化）



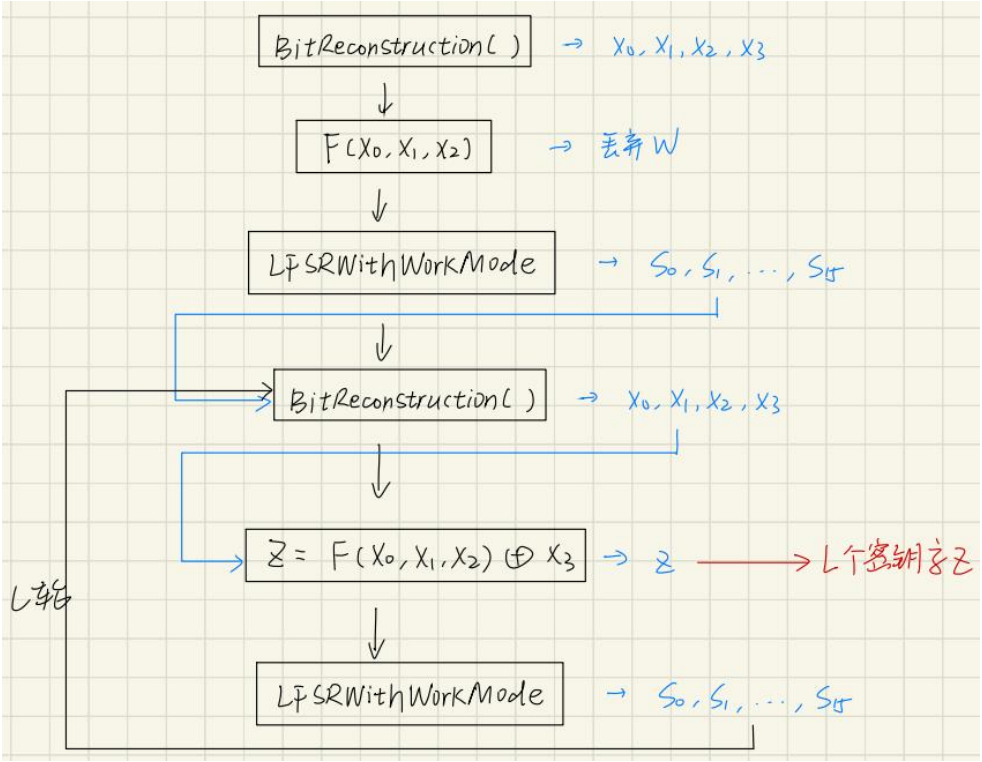


代码如下：

```
# 3.执行 32 次循环（1.比特重组 2.非线性函数输出 W 3.LSFR 初始化）
        for i in range(32):
            # 1.比特重组（输出 X0，X1，X2，X3）
            self.BitReconstruction()
            # 2.将上一步得到的 X0，X1，X2，X3 作为输入传给 F 函数，得到 W
            self.F()
            # 3.将上一步得到的 W 送进 LSFR 初始化模式得到 S0~S15
            self.LFSRWithInitializationMode()
```

接下来就是第二阶段，工作状态，流程图及代码如下：

```
self.generate_32bit()
```



1）进行比特重组,将初始化后的 S0~S15 进行比特重组，得到 X0，X1，X2，X3：

```
    # ZUC 的第二个阶段 工作模式
    def generate_32bit(self):
        # 1.比特重组：将初始化后的 S0~S15 进行比特重组，得到 X0，X1，X2，X3
        self.BitReconstruction()
```

2）F 函数,工作模式下不需要保留 F 函数的输出 W（将 W 丢弃），仅仅是对 F 函数的 R0，R1 进行更新：

```
        # 2.F 函数：工作模式下不需要保留 F 函数的输出 W（将 W 丢弃），仅仅是对 F 函数的 R0，R1 进行更新
        z = self.F() ^ self.x3
```

3）进入 LFSR 工作模式：

```
        # 3.进入 LFSR 工作模式
        self.LFSRWithWorkMode()
        return z
```

主体部分已经介绍完毕，现在来看具体的实现细节。

1）移位函数及线性变换函数：

```
# 不丢失位的移位函数
def rol(bits, shift_num):
    return bits[shift_num:] + bits[:shift_num]

# 线性变换 L1，L2 都是将 32 位的字转换成另一个 32 位的字
def L1(bits):
    return bits ^ rol(bits, 2) ^ rol(bits, 10) ^ rol(bits, 18) ^ rol(bits,
24)
def L2(bits):
    return bits ^ rol(bits, 8) ^ rol(bits, 14) ^ rol(bits, 22) ^ rol(bits,
30)
```

2）生成 S-box，32*32 大小的 S-Box（S）本质上是四个大小为 8*8 的 S-boxes 并置而成：

```
# 生成 S-box
def S(bits):
    result = bitarray('')
    result += int2ba(zuc.s0[ba2int(bits[0:8])], length=8)
    result += int2ba(zuc.s1[ba2int(bits[8:16])], length=8)
    result += int2ba(zuc.s0[ba2int(bits[16:24])], length=8)
    result += int2ba(zuc.s1[ba2int(bits[24:32])], length=8)
    return result
```

3）比特重组 BR，从 LFSR 中提取出来 128bit 组成 4 个 32bit 的字，前三个字提供给底层非线性 F 函数使用的，最后一个字用来生成密钥流：

```
    # 比特重组 BR：从 LFSR 中提取出来 128bit 组成 4 个 32bit 的字，前三个字提供给
底层非线性 F 函数使用的，最后一个字用来生成密钥流
    def BitReconstruction(self):
        self.x0 = self.s[15][:16] + self.s[14][-16:]
        self.x1 = self.s[11][-16:] + self.s[9][:16]
        self.x2 = self.s[7][-16:] + self.s[5][:16]
        self.x3 = self.s[2][-16:] + self.s[0][:16]
```

4）非线性函数 F 含有 2 个 32bit 的记忆单元 R1，R2；输入位 3 个 32bit 的字 X0，X1，X2，输出 32bit 的 W：

```
    # 非线性函数 F：含有 2 个 32bit 的记忆单元 R1，R2；输入位 3 个 32bit 的字 X0，
X1，X2，输出 32bit 的 W
    def F(self):
```

```
        # W 是 X0 和 R1 异或后，跟 R2 进行模 2^32 加法运算，作为输出 W
        self.w = int2ba((ba2int(self.x0 ^ self.r1) + ba2int(self.r2)) %
(2**32), length=32)
        # 更新 W1，W2
        w1 = int2ba((ba2int(self.r1) + ba2int(self.x1)) % (2**32),
length=32)
        w2 = self.r2 ^ self.x2
        # 更新 R1，R2
        self.r1 = S(L1(w1[-16:] + w2[:16]))
        self.r2 = S(L2(w2[-16:] + w1[:16]))
        return self.w
```

5）接下来是 LFSR 的实现，初始化模式读入一个 31 位的输入字 u，通过从非线性函数 F 的 32 位输出 W 中删除最右边的位来获得；工作模式没有输入：

```
  # LFSR 的初始化模式：读入一个 31 位的输入字 u，通过从非线性函数 F 的 32 位输出 W
中删除最右边的位来获得
    def LFSRWithInitializationMode(self):
        v = (2**15*ba2int(self.s[15]) + 2**17*ba2int(self.s[13]) +
2**21*ba2int(self.s[10]) \
             + 2**20*ba2int(self.s[4]) + (1+2**8)*ba2int(self.s[0])) %
(2**31 - 1)
        s16 = int2ba((v + ba2int(self.w[:-1])) % (2**31 - 1), length=31)
        for i in range(15):
            self.s[i] = self.s[i + 1]
        self.s[15] = s16

    # LSFR 的工作模式：没有输入
    def LFSRWithWorkMode(self):
        s16 = (2**15*ba2int(self.s[15]) + 2**17*ba2int(self.s[13]) +
2**21*ba2int(self.s[10]) \
               + 2**20*ba2int(self.s[4]) + (1+2**8)*ba2int(self.s[0])) %
(2**31 - 1)
        if s16 == 0:
            s16 = 2**31 - 1
        s16 = int2ba(s16, length=31)
        # 将(s1,s2...s16)->(s0,s1...s15)
        for i in range(15):
            self.s[i] = self.s[i + 1]
        self.s[15] = s16
```

四、实验结果与数据处理

执行 main 函数，这里用书上 171 的例子作为输入：

```
if __name__ == "__main__":
    # ZUC 的输入如下（这里采用书上 171 页的例子）：
    # 1.初始密钥 key
    key = int2ba(0x3d4c4be96a82fdaeb58f641db17b455b, length=128)
    # 2.初始向量 iv
    iv = int2ba(0x84319aa8de6915ca1f6bda6bfbd8c766, length=128)
    # 3.正整数 L（由明文比特长度/32 向上取整得到）
    L = 2

    # 现在进入 ZUC 算法
    zucBlock = ZUC(key, iv)
    for i in range(L):
        print(bit2str(zucBlock.generate_32bit()))
```

得到结果如下，和书本上的结果一致：

```
(py38) root@6dceeba50b41:~/project/test/ZUC# python ZUC-main.py
14f1c272
3279c419
```

五、分析与讨论

1. 做到这一步，我才发现我在实现中的初始向量 IV 还挺重要的，但上面并没有解释 IV 要如何选择，那么 ZUC 算法使用 128 位的初始向量 IV 如果重复了会怎么样呢？

查阅资料发现 IV 的作用是即使在相同的密钥下，也会生成不同的密钥流。但是如果 IV 被重用，特别是在与相同的密钥结合时，它会降低系统的安全性。所以这里 IV 应该是唯一的，最好是随机生成的。在实际应用中，应该确保每次加密使用不同的 IV。一种可能的策略是结合某些会话特定的参数，如时间戳、随机数或序列号等信息来生成 IV，这样可以减少重复的可能性。在协议设计中，应当明确规定如何安全地生成和管理 IV，以避免重用。

2. 我发现 ZUC 算法的一个关键环节是其复杂的内部状态更新机制，这种状态更新机制怎么能来确保呢？

ZUC 在保持其输出序列的随机性和不可预测性至关重要。每一个环节都要进行彻底的测试和验证，不仅是功能性测试，还应包括安全性测试，如抗差分攻击和抗线性攻击的能力。另外需要定期审查和更新。	
六、教师评语 签名： 日期：	成绩