山东大学
SHANDONG UNIVERSITY

流密码算法的硬件实现

密码工程

2023 年 11 月 29 日

李 昕 202100460065

21 级密码 2 班

摘　要

A5/1 是用于在 GSM 蜂窝电话标准中提供无线通信隐私的流密码。它是为 GSM 使用指定的七种算法之一。A5/1 算法基于线性反馈移位寄存器(LFSR)和非线性布尔函数。它使用一个 64 位的密钥,以及三个不同的 LFSR 序列来生成密钥流。这个密钥流与要传输的数据进行异或运算，从而实现加密和解密。密钥流在通信的每个时间片中都会变化，以增强安全性。

Grain 算法也是面向硬件实现的算法之一，它包括一个 128 比特的 LFSR、一个 128 比特的 NFSR 和一个过滤函数 h，该 LFSR 周期达到最大，即 $2^{128}-1$，而这种 LFSR 到 NFSR 的串联结构保证了输出序列的周期至少是 $2^{128}-1$。同时，NFSR 和非线性过滤函数 h 也增加了密码强度。

在本次实验中，利用 Verilog 语言实现 A5/1 流密码算法和 Grain-128 算法的硬件实现。

关键词：Grain-128 算法　A5/1　Verilog

目录

1 实验要求

问题一 分析 A5/1 的结构和实现细节，利用 Verilog 语言实现 A5/1 流密码算法的硬件实现，并利用软件工具进行仿真测试。

问题二 分析 Grain-128 的结构和实现细节，利用 Verilog 语言实现其硬件实现，并利用软件工具进行仿真测试。

2 符号标记说明

符号	解释
$\oplus$	异或
maj(x, y, z)	多数投票函数

3 实验准备

3.1 A5/1 算法的数学原理 [1]

A5/1 算法的数学原理主要涉及线性反馈移位寄存器（LFSR）和非线性布尔函数的使用。A5/1 算法使用 3 个 LFSR，共计 64 位的密钥作为输入。设定三个寄存器，分别为 X、Y 和 Z。寄存器 X 有 19 位，可以表示为 (x0, x1, ..., x19); 寄存器 Y 有 22 位，可以表示为 (y0, y − 1, ..., y22); 而寄存器 Z 有 23 位，可以表示为 (z0, z1, ..., Z22)，初始密钥流作为 X,Y,Z 寄存器的初始值。

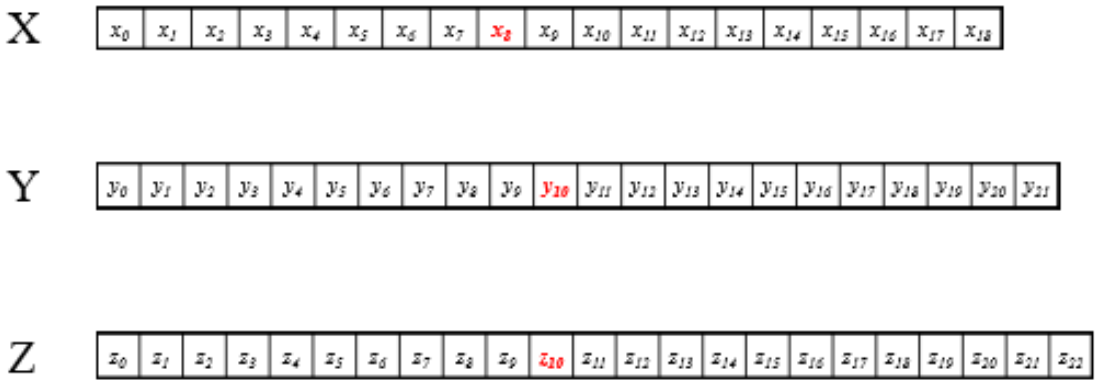

Figure 1: X,Y,Z 寄存器

寄存器 X,Y,Z 分别的操作为:

$$\begin{cases} t = x_{13} \oplus x_{16} \oplus x_{17} \oplus x_{18} \\ x_i = x_{i-1} for i = 18, 17, ..., 1 \\ x_0 = t \end{cases} \tag{1}$$

$$\begin{cases} t = y_{20} \oplus y_{21} \\ y_i = y_{i-1} for i = 21, 20, ..., 1 \\ y_0 = t \end{cases} \tag{2}$$

$$\begin{cases} t = z_7 \oplus z_{21} \oplus z_{22} \\ z_i = z_{i-1} for i = 22, 21, ..., 1 \\ z_0 = t \end{cases} \tag{3}$$

定义 maj 为多数投票函数 (Majority Function), 定义 maj(x,y,z) 的输出为输入值中最多的值，则在 A5/1 中，对于生成的每一个密钥流的位，都将执行如下操作:

算法 1 生成该位值将与明文进行异或运算 (在加密的情况下)，或者与密文进行异或运算 (在解密的情况下)。重复这一算法，可以生成所需数量的密钥流位数。

Algorithm 1 Algorithm for Generating the bits of each key stream

Input: x_8, y_{10}, z_{10}

Output: s

$m = maj(x_8, y_{10}, z_{10})$
if $x_8 = m$ then
    RUN formula(1)
end if
if $y_{10} = m$ then
    RUN formula(2)
end if
if $z_{10} = m$ then
    RUN formula(3)
end if
$s = x_{18} \oplus y_{21} \oplus z_{22}$
return s

3.2 Grain128 算法的数学原理 [2]

和 A5/1 算法相似的是，Grain128 也利用了线性反馈移位寄存器（LFSR），但不同之处在于 Grain128 还是用了非线性滤波器来生成密钥流，而生成的密钥流可以用作对称加密算法中的密钥。

与 A5/1 算法相比，Grain128 的过程更加复杂，下面是其循环流程：

1. 设置密钥和初始化变量：定义密钥：$K = K_0K_1...K_{127}$，其中 K_i 为第 i 位密钥比特。初始化向量：$IV = IV_0IV_1...IV_{95}$，其中 IVi 为第 i 位初始化向量比特。初始内部状态（seed）$Seed = K||IV$，即将密钥和初始化向量进行拼接，得到一个 256 位的 seed。

2. 三个线性反馈移位寄存器的操作：$LFSR_A$ 反馈多项式为 $a(x) = x^{128} + x^{86} + x^{67} + x^5 + 1$, 输出为 $LFSR_{A0}LFSR_{A1}...LFSR_{A127}$。$LFSR_B$ 反馈多项式为 b(x) $= x^{128} + x^{110} + x^{69} + x^{24} + 1$, 输出为 $LFSR_{B0L}FSR_{B1}...LFSR_{B127}$。$LFSR_C$ 反馈多项式为 $c(x) = x^{128} + x^{109} + x^{87} + x^{64}$，输出为 $LFSR_{C0}LFSR_{C1}...LFSR_{C127}$。

每个时钟周期内，对于 LFSR A 和 B，计算反馈值，以 A 为例，其计算式为：$F_A = (LFSR_{A63} + LFSR_{A60} + LFSR_{A52} + LFSR_{A45} + LFSR_{A37} + LFSR_{A33} + LFSR_{A28} + LFSR_{A21} + LFSR_{A15} + LFSR_{A9} + LFSR_{A7} + LFSR_{A0} + LFSR_{A63} * LFSR_{A60} + LFSR_{A37} * LFSR_{A33} + LFSR_{A15} * LFSR_{A9} + LFSR_{A63} * LFSR_{A37} * LFSR_{A15} + LFSR_{A60} * LFSR_{A33} *$

$LFSR_{A9}+LFSR_{A63}*LFSR_{A33}*LFSR_{A9}+LFSR_{A63}*LFSR_{A60}*LFSR_{A9}+LFSR_{A63}*LFSR_{A60}*LFSR_{A33}+LFSR_{A37}*LFSR_{A33}*LFSR_{A15}+LFSR_{A63}*LFSR_{A37}*LFSR_{A33}*LFSR_{A15})mod2$。之后，更新 LFSR A 和 B 的状态，以 A 为例，$LFSR_A = F_A||LFSR_{A127}...LFSR_{A1}$。

设 $LFSR_C$ 的初始状态为 $LFSR_C = LFSR_{A127}...LFSR_{A120}||LFSR_{B127}...LFSR_{B120}$，辅助时钟周期内，对于 $LFSR_C$，计算反馈值 $F_C = (LFSR_{C66}+LFSR_{C65}+LFSR_{C61}+LFSR_{C60})mod2$，并更新 $LFSR_C$ 的状态 $LFSR_C = F_C||LFSR_{C127}...LFSR_{C1}$。

对于非线性滤波器(NLFSR)，设其输入为 $LFSR_{A95}...LFSR_{A66}||LFSR_{B95}...LFSR_{B66}$，其功能和 LFSR 类似，不过 NFSR 的反馈函数使用非线性函数 h。

Grain128 算法通过下图流程完成上述各步：

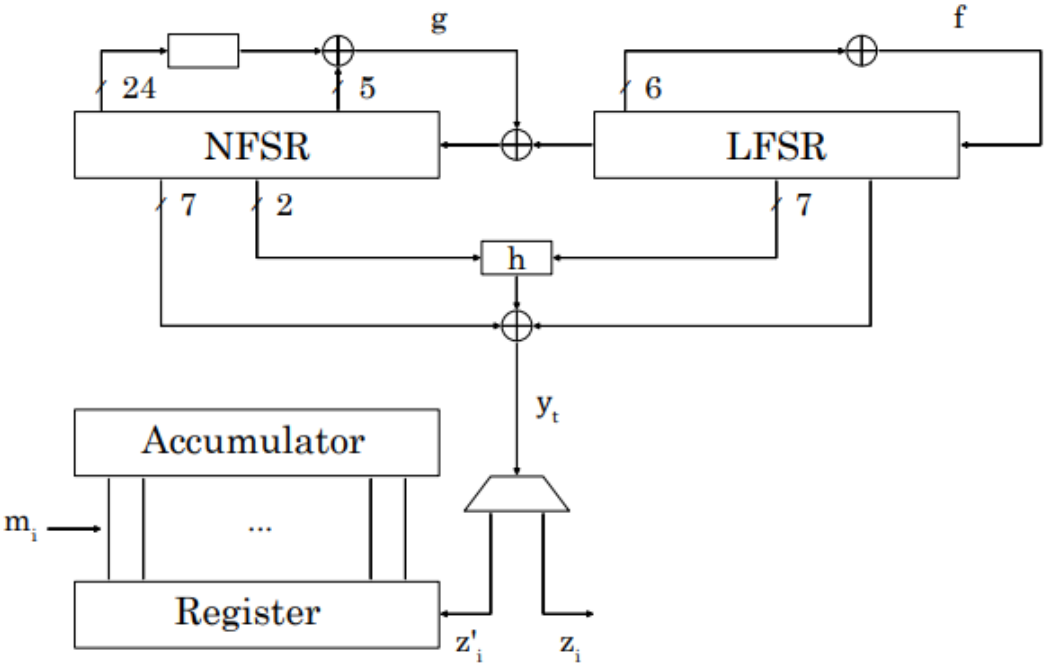

Figure 2: Grain 流程图

4 实验过程

4.1 编写 A5/1 Verilog 代码

4.1.1 编写 LFSR 模块的 Verilog 代码

首先定义输入输出，定义输入 *key* 为 64 位的向量，表示初始密钥。

定义输出 *key_next* 为 64 位的向量，表示下一个时钟周期的密钥。定义 x 表示 LFSR1 的状态，y 表示 LFSR2 的状态，z 表示 LFSR3 的状态。定义 x_next, y_next, z_next

对应 LFSR 的下一个状态。

对寄存器内容进行划分：x 的 19 位来自 key 的低位。y 的 22 位来自 key 的中间部分。z 的 23 位来自 key 的高位。

最后，定义 maj 函数用来计算三个 LFSR 输出的多数位。

以上代码实现如下：

```
input  [63:0] key;         // 输入原始密钥
output [63:0] key_next;    // 输出下一个密钥

wire   [18:0]x;            // lfsr1
wire   [21:0]y;            // lfsr2
wire   [22:0]z;            // lfsr3
wire   [18:0]x_next;
wire   [21:0]y_next;
wire   [22:0]z_next;
wire   majority;

assign x=key[18:0];        // 低位是 x
assign y=key[40:19];
assign z=key[63:41];       // 高位是 z
```

定义多数投票函数 maj：如果 x, y, z 多数为 0，那么函数返回 0，否则返回 1。根据函数功能列出真值表如下：

x	y	z	f(x,y,z)
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

由此得到其计算代码如下：

```
function [0:0] maj(input x,y,z);
  maj = x&y | x&z | y&z;  // 计算多数位
endfunction
```

之后，实现算法 1 所述部分：

```
function [18:0] lfsr1(input [19:0]in);
begin
  case(in[19])
    0:lfsr1={in[17:0], in[18]^in[17]^in[16]^in[13]};  // 如果最高位为0，进行位运算
    1:lfsr1=in[18:0];      // 如果最高位为1，保持不变
  endcase
end
endfunction

function [21:0] lfsr2(input [22:0]in);
begin
  case(in[22])
    0:lfsr2={in[20:0], in[21]^in[20]};  // 如果最高位为0，进行位运算
    1:lfsr2=in[21:0];      // 如果最高位为1，保持不变
  endcase
end
endfunction

function [22:0] lfsr3(input [23:0]in);
begin
  case(in[23])
    0:lfsr3={in[21:0], in[22]^in[21]^in[20]^in[7]};  // 如果最高位为0，进行位运算
    1:lfsr3=in[22:0];      // 如果最高位为1，保持不变
  endcase
end
endfunction

assign majority = maj(x[8],y[10],z[10]);  // 计算多数位
assign x_next = lfsr1({majority^x[8], x[18:0]}); // 计算下一个 x
assign y_next = lfsr2({majority^y[10], y[21:0]}); // 计算下一个 y
assign z_next = lfsr3({majority^z[10], z[22:0]}); // 计算下一个 z
```

```
assign key_next = {z_next,y_next,x_next};        // 组合下一个密钥，低位是 x，高位是 z
```

4.1.2 编写 A5/1 主体模块的 Verilog 代码

定义 A51 模块，输入包括原始密钥 key、明文 plain、时钟信号 clk 和密钥准备信号 krdy，输出为密文 cipher。

定义三个寄存器类型的变量：kpre、knext 和 k，分别用于存储上一个时钟周期的 key、下一个时钟周期的 key 和最后生成的流密码。然后实例化 LFSR 模块，并将 kpre 和 knext 与对应的信号连接起来。使用 assign 语句将密文 cipher 赋值为 k 与明文 plain 的异或结果。

最后，在 always 块中，根据 krdy 信号的状态，判断密钥是否就绪。如果就绪，则将输入的 key 赋值给 kpre；否则，将 knext 的第 18、40 和 63 位进行异或运算，结果赋值给 k，并将 knext 赋值给 kpre。

代码实现如下图所示：

```
module A51(key,plain,cipher,clk,krdy);
input  [63:0]key;          // 原始密钥
input  plain;              // 明文流
output cipher;             // 密文流
input  clk;                // 系统时钟
input  krdy;               // 原始密钥是否就绪

reg [63:0] kpre;           // 密钥寄存器
wire[63:0] knext;          // 下一个密钥
reg k;                     // 密文流的密钥

LFSR lfsr(kpre,knext);     // 调用 LFSR 模块

assign cipher=k^plain;     // 计算密文流的密钥

always @(posedge clk) begin
  if(krdy==1) begin        // 如果原始密钥已经就绪，则刷新密钥寄存器
    kpre <=key;
  end
  else begin               // 如果原始密钥未就绪，则使用下一个密钥
    k = knext[18]^knext[40]^knext[63];  // 计算下一个密文流的密钥
```

```
    kpre <= knext;
  end
end

endmodule
```

4.2 编写 A5/1 仿真测试代码

接下来，编写对应的仿真测试代码：

```
module A51_tb;
  reg  Plain;
  reg  [63:0] Key;
  wire Cipher;
  reg  Krdy;
  reg clk;
  initial begin
    clk = 0;
    #10 clk = ~clk;
    #10 clk = ~clk;
    ...//此处为压缩篇幅，省略部分时钟控制
    #10 clk = ~clk;
    #10 clk = ~clk;
    #10 clk = ~clk;
  end
  A51 A51_in_tb(Key,Plain,Cipher,clk,Krdy);
  initial begin
    Key  ={32'b00001111000101010111000111001001,
          32'b10101111011111110110011110011000};
    Krdy =1;
    #20 Krdy = 0;
  end
  initial begin
    Plain=1;
    #30 Plain=~Plain;
    #20 Plain=~Plain;
    #20 Plain=~Plain;
```

```
    #20 Plain=~Plain;
    #20 Plain=~Plain;
    #20 Plain=~Plain;
  end
  initial begin
    $dumpfile("test.vcd");
    $dumpvars;
  end
endmodule
```

4.3 编写 Grain-128 硬件代码

4.3.1 LFSR 和 NLFSR 的编写

根据 3.2 节数学原理，编写硬件代码实现 LFSR，NLFSR(包括 NFSR 和 h 函数) 代码，在本部分中，实现三个模块：

1. hx 模块：算法的非线性反馈函数，该模块根据输入信号对一系列状态寄存器进行逻辑门操作，并返回 ho 作为 h 函数的函数值。

2. linear 模块实现了线性反馈函数。通过对输入信号进行异或操作，生成了 lfb 作为线性反馈的一部分。

3. nonlinear 模块实现非线性反馈函数。根据输入信号进行一系列异或和逻辑门操作，生成了 nfb 作为非线性反馈的一部分。

这三个模块组合在一起，构成了 Grain-128 算法中的反馈函数（Feedback Function）部分，用于生成密钥流。

```
module h_x
(
    input s8,
    input s13,
    input s20,
    input s42,
    input s60,
    input s79,
    input s95,
    input b12,
    input b95,
```

```
    output ho
);

// 通过一系列逻辑操作计算输出ho
assign ho = (b12 & s8) ^ (s13 & s20) ^ (b95 & s42) ^ (s60 & s79) ^ (b12 & s95 & b95);

endmodule

module feedback_linear
(
    input s0,
    input s7,
    input s38,
    input s70,
    input s81,
    input s96,
    output lfb
);

// 通过异或逻辑操作计算输出lfb
assign lfb = s0 ^ s7 ^ s38 ^ s70 ^ s81 ^ s96;

endmodule

module feedback_non_linear
(
    input b0,
    input b26,
    input b56,
    input b91,
    input b96,
    input b3,
    input b67,
    input b11,
    input b13,
    input b17,
    input b18,
    input b27,
```

```
    input b59,
    input b40,
    input b48,
    input b61,
    input b65,
    input b68,
    input b84,
    output nfb
);

// 通过一系列异或和与逻辑操作计算输出nfb
assign nfb = b0 ^ b26 ^ b56 ^ b91 ^ b96 ^ (b3 & b67) ^ (b11 & b13) ^ (b17 & b18) ^ (b27 & b59) ^ (
    b40 & b48) ^ (b61 & b65) ^ (b68 & b84);

endmodule
```

4.3.2 Grain-128 主体的编写

```
module grain128
(
    input clk,
    input rst,
    input [127:0] key,
    input [95:0] iv,
    input gen,
    output reg rdy,
    output z,
    output [31:0] tag
);

localparam INIT = 0;
localparam RDY  = 1;

reg [127:0] s;          //LFSR
reg [127:0] b;          //NFSR
reg curr_state = INIT;
reg next_state = INIT;
```

```
20 reg [7:0] cnt  = 0;       //Counter
21 reg m          = 1;       //Mode
22
23 wire f,g,h;
24 wire nfb, lfb;
25 wire mo;                  //MUX out
26 wire y;                   //y
27 wire [127:0] key_in;      //Flipped key
28 wire [95:0] iv_in;        //Flipped iv
29
30 feedback_linear FB_L (s[0], s[7], s[38], s[70], s[81], s[96], f);
31 feedback_non_linear FB_N (b[0], b[26], b[56], b[91], b[96], b[3], b[67], b[11], b[13], b[17], b[18], b[27], b
       [59], b[40], b[48], b[61], b[65], b[68], b[84], g);
32 h_x HX (s[8], s[13], s[20], s[42], s[60], s[79], s[95], b[12], b[95], h);
33
34 assign mo  = m & y;
35 assign lfb = f ^ mo;
36 assign nfb = g ^ mo ^ s[0];
37 assign y   = h ^ s[93] ^ b[2] ^ b[15] ^ b[36] ^ b[45] ^ b[64] ^ b[73] ^ b[89];
38 assign z   = y;
39
40 assign tag = 0;
41
42 generate
43     genvar i;
44     for (i = 0; i < 96; i = i + 1) begin
45         assign iv_in[i] = iv[95-i];
46     end
47 endgenerate
48
49 generate
50     genvar j;
51     for(j = 0; j < 128; j = j + 1) begin
52         assign key_in[j] = key[127-j];
53     end
54 endgenerate
55
56 always @(posedge clk) begin
```

```
57      if (rst) begin
58          curr_state <= INIT;
59          s          <= {{32{1'b1}},iv_in};
60          b          <= key_in;
61          cnt        <= 0;
62      end
63      else begin
64          curr_state <= next_state;
65      end
66
67      case (curr_state)
68          INIT : begin
69              m   <= 1;
70              rdy <= 0;
71              if (cnt == 255) begin
72                  next_state <= RDY;
73              end
74              else begin
75                  next_state <= INIT;
76                  cnt        <= cnt + 1;
77              end
78              s = {lfb, s[127:1]};
79              b = {nfb, b[127:1]};
80          end
81          RDY : begin
82              rdy <= 1;
83              m   <= 0;
84
85              if (gen && rdy) begin
86                  s = {lfb, s[127:1]};
87                  b = {nfb, b[127:1]};
88              end
89              next_state <= RDY;
90          end
91      endcase
92  end
93  endmodule
```

4.4 编写 Grain-128 仿真测试代码

```
‘timescale 1ns / 1ps
module tb_grain128;

reg clk = 0;
reg rst = 0;
reg gen = 0;

reg [127:0] key = 128’h0123456789abcdef123456789abcdef0;
reg [95:0] iv  =  96’h0123456789abcdef12345678;


reg [127:0] keystream = 0;
reg [7:0] len = 0;
wire rdy,z;
wire [31:0] tag;

grain128 DUT (
    .clk(clk),
    .rst(rst),
    .key(key),
    .iv(iv),
    .gen(gen),
    .rdy(rdy),
    .z(z),
    .tag(tag)
);

initial begin
    #10 rst = 1;
    #10 rst = 0;
 #10 rst = 1;
    #10 rst = 0;
  ...//此处为压缩篇幅，省略部分时钟控制
#10 rst = 1;
    #10 rst = 0;

end
```

```
38
39 initial begin
40     clk = 0;
41     #10 clk = ~clk;
42     #10 clk = ~clk;
43      ...//此处为压缩篇幅，省略部分时钟控制
44     #10 clk = ~clk;
45     #10 clk = ~clk;
46   end
47
48 always @(posedge clk ) begin
49     if (rdy == 1) begin
50         if (len > 128) begin
51             gen = 0;
52         end
53         else begin
54             gen = 1;
55             // keystream = {z,keystream[127:1]};
56             keystream = {keystream[126:0],z};
57             len = len + 1;
58         end
59     end
60 end
61
62  initial begin
63     $dumpfile("test_tb.vcd");
64     $dumpvars;
65   end
66
67 endmodule
```

5 结果分析

5.1 A5/1 仿真结果

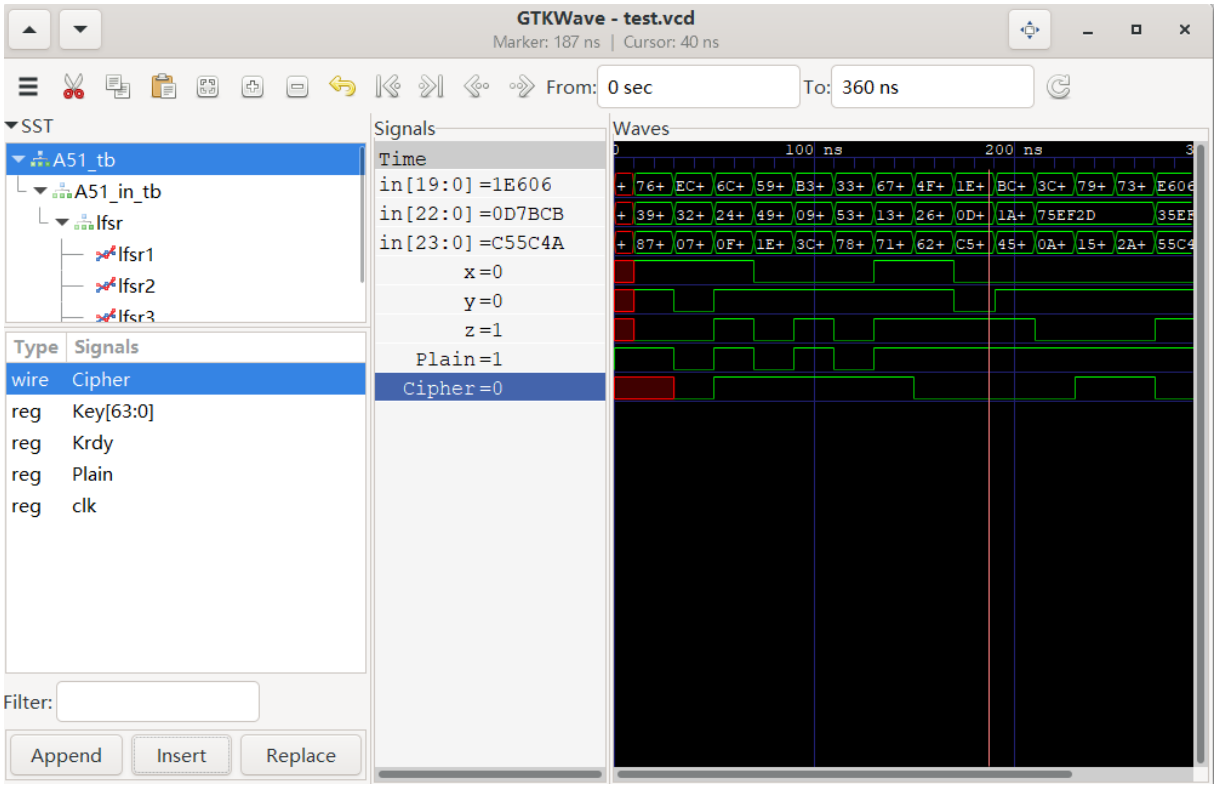

Figure 3: 仿真结果

分析波形图可以看到，当前选择时间点，$x = 0, y = 0, z = 1, k = x \oplus y \oplus z = 1$，当前明文 Plain 为 1，密文 Cipher $= Plain \oplus k = 0$，验证正确。

5.2 Grain-128 仿真结果

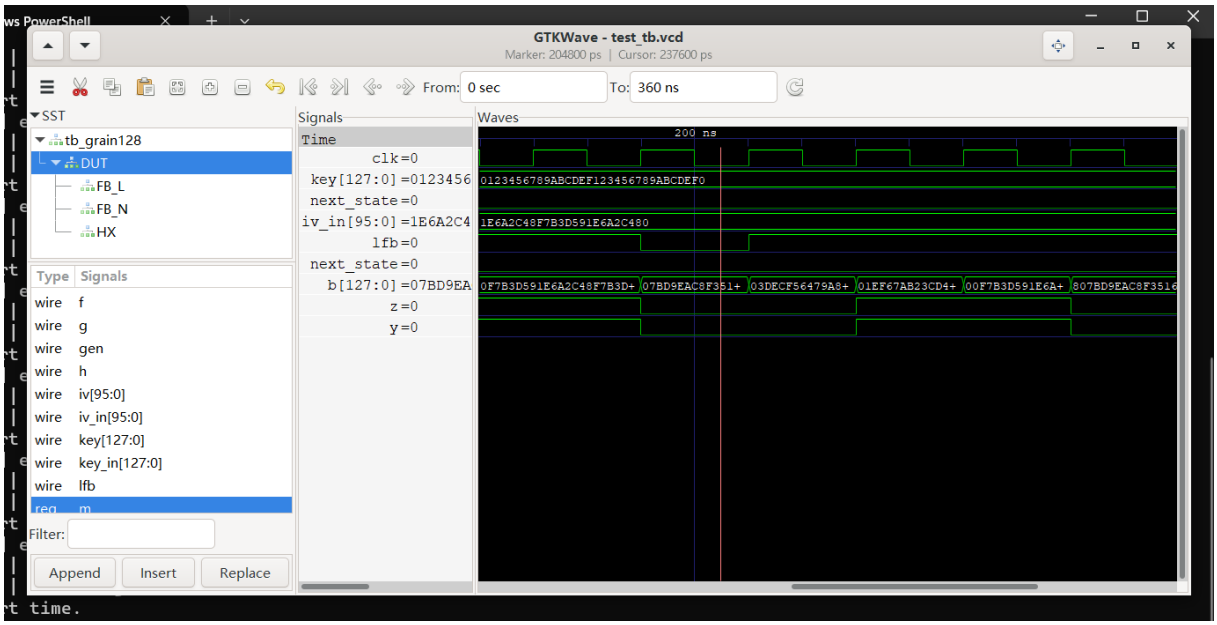

Figure 4: 仿真结果

可以看到得到了流密码 Grain-128 的曲线。

参考文献

[1] Biham E, Dunkelman O (2000) Cryptanalysis of the A5/1 GSM stream cipher. In: Indocrypt 2000. Lecture notes in computer science, vol 1977. Springer, Heidelberg, pp 43–51.

[2] Martin Hell,Thomas Johansson, Alexander Maximov; Willi Meier.A Stream Cipher Proposal: Grain-128.