说明文档

1. 指令步骤详细说明

主要实现了 Simon64/128 和 Speck64/128。两个密码算法的分组长度都是 64 位，分成左右两个部分，各为 32 位。文中的所有操作对象都是针对 32 位。

两个算法都用到异或（XOR）、按位与（AND）。SIMON 中用到循环左移 1 位（S^1）、循环左移 2 位（S^2）、循环左移 8 位（S^8）；SPECK 中用到循环左移 3 位（S^3）、循环右移 8 位（S^{-8}）、模 2^n 加。SIMON 密钥编排用到循环右移 1 位（S^{-1}）、循环右移 3 位（S^{-3}）。

1.1 Bitwise XOR

1 个 cycle 可以完成 1 个字节的异或操作。完成 32 位异或操作，需要 4 条指令，4 个 cycles，8 字节 flash，如图 1 所示（图中 r0、r1、……、r7 都表示一个寄存器，下文中所有代码示例都是如此，不再重复说明）。

```
eor r0, r4;
eor r1, r5;
eor r2, r6;
eor r3, r7;
```

图1. 32 位异或操作

```
add r0, r4;
add r1, r5;
add r2, r6;
add r3, r7;
```

图2. 32 位与操作

1.2 Bitwise AND

完成 32 位与操作，需要 4 条指令，4 个 cycles，8 字节 flash，如图 2 所示。

1.3 Left circular shift, S^j, by j bits

1) S^1：循环左移 1 位可以通过逻辑左移（LSL）、循环左移（ROL）和带进位加法（ADC）实现。

LSL：寄存器最高位进入到标志寄存器的 C 位，低位向高位移动，最低位补 0。

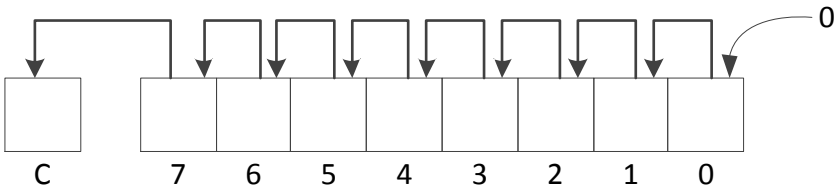


图3. LSL

ROL：最高位进入到标志寄存器的 C 位，低位向高位移动，C 位进入最低位。

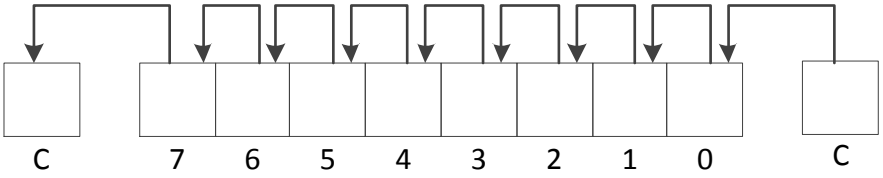


图4. ROL

ADC：标志寄存器 C 位的值加到两个操作数的最低位。如果加 C 后最低位存在进位，则高位继续加 C，循环直到最高位相加结束，最高位的进位保存在 C。下图是 ADC 和 ADD 的详细过程。ADC 与 ADD 最主要的区别在于最低位会加上当前进位 C 的值。

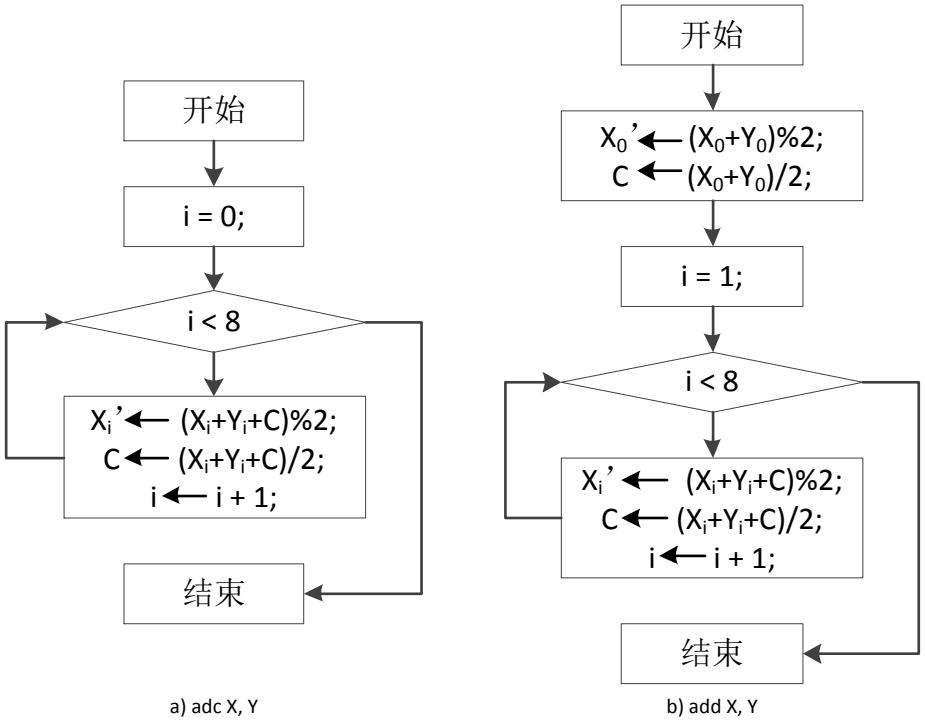


图5. ADD 和 ADC 操作

假设[r3,r2,r1,r0]（r3 对应最高 8 位，依此类推，r0 对应最低 8 位）表示一个 32 bits 的数，图 6 所示是通过一条 LSL、3 条 ROL 和一条 ADC 指令实现[r3,r2,r1,r0]

循环左移 1 位（其中，zero 对应一个值为 0 的寄存器）。一共需要 5 条指令，5 个 cycles，10 字节 flash。

```
lsl r0;
rol r1;
rol r2;
rol r3;
adc r4, zero
```

图6. S^1

指令分析：根据前面介绍可知，LSL 指令会使得 r0 的最低位位 0，因此通过 ADC 指令加上一个值为 0 的寄存器，实际上就是将当前 C 位的值保存在 r0 的最低位，而最近一次修改 C 位的操作是 ROL，它将 r3 的最高位保存在 C 位，这样上述 5 条指令正好将[r3,r2,r1,r0]循环左移一位。

2) S^2：通过 2 次 S^1 实现，共需要 10 条指令，10 个 cycles，20 字节 flash。
3) S^3：通过 3 次 S^1 实现，共需要 15 条指令，15 个 cycles，30 字节 flash；还可以通过乘法指令实现。

表 1 是实现[Y3,Y2,Y1,Y0]循环左移 3 bits 的过程，移位后的结果保存在[X3,X2,X1,X0]中。

表1. 乘法实现 S^3

operation			cycles
(R1, R0)	←	MUL(Y0, 8)	2
(X1, X0)	←	(R1, R0)	1
(R1, R0)	←	MUL(Y2, 8)	2
(X3, X2)	←	(R1, R0)	1
(R1, R0)	←	MUL(Y1, 8)	2
X1	←	X1 EOR R0	1
X2	←	X2 EOR R1	1
(R1, R0)	←	MUL(Y3, 8)	2
X3	←	X3 EOR R0	1
X0	←	X0 EOR R1	1

图 7 是通过乘法指令实现 S^3 的示意图。

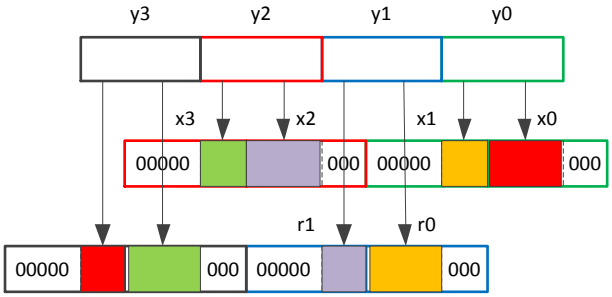


图7. 乘法实现 S^3 示意图

使用乘法指令实现一共需要 10 条指令，14 个 cycles，20 字节 flash。相比直接通过 3 次 S^1 实现，执行时间和执行数目都要少。但由于乘法指令的结果只能保持在 R1:R0，因此这两个寄存器不能用于其它用途。同时，乘法指令执行后的结果并不是在原寄存器中，如果要保持乘法指令带来的优势，则必须通过循环展开抵消这种错位，这样指令数会有所增加。

4) S^8：理论上实现循环左移 8 bits 需要 5 条移位指令，如图 8 所示（temp 也对应一个寄存器）。

```
mov temp, r3;
mov r3, r2;
mov r2, r1;
mov r1, r0;
mov r0, temp;
```

图8. S^8

但由于 Atmega128 的寄存器都是 8 位的，在实际的运算中可以错位选择相应的寄存器参与运算，而不需要通过显式的移位操作。因此，循环左移 8 位不需要执行时间。

1.4 Right circular shift, S^{-j}, by j bits

1) S^{-1}：循环右移 1 位可以通过逻辑右移（LSR）、循环右移（ROR）、位存储（BST）和位加载（BLD）操作实现。

LSR：寄存器最高位补 0，高位向低位移动，最低位进入到标志寄存器的 C 位。

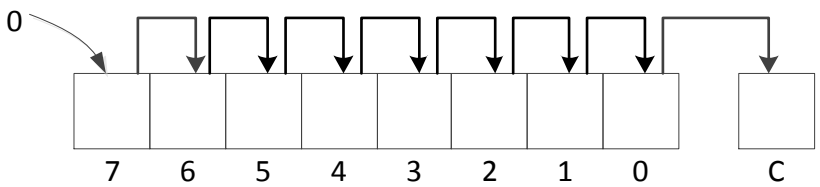


图9. LSR

ROR：C 位进入寄存器最高位，高位向低位移动，最低位进入到标志寄存器的 C 位。

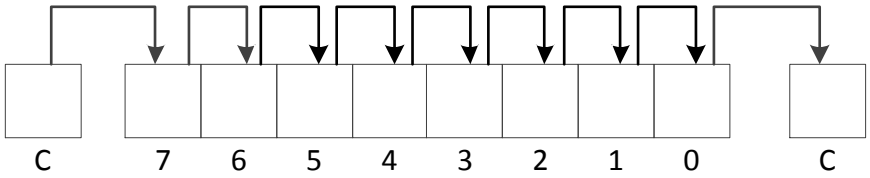


图10.　ROR

BST：将寄存器的指定一个 bit 存放到标志寄存器的 T 位。

BLD：将寄存器的指定一个 bit 置为标志寄存器 T 位的值。

假设[r3,r2,r1,r0]4 个寄存器存放 32 bit 的值，则通过下面的 6 条指令可以实现循环右移 1 位。

```
lsr r3;
ror r2;
ror r1;
bst r0, 0;
ror r0;
bld r3, 7;
```

图11.　S^{-1}

S^{-1} 需要 6 条指令，共 6 个 cycles，12 字节 flash。

2) S^{-3}：通过 3 次 S^{-1} 实现，共需要 18 条指令，18 个 cycles，36 字节 flash。

3) S^{-8}：和 S^{8} 的原理一样，不需要执行时间。

1.5 Modular addition

模 2^n（n 为一个分组所包含比特位的数目，本文中都是 32）的加法和一般的加法基本一致，不同之处在于它会舍弃最高位的进位。因此，对于 32 位的模 2^n 加法可以通过 4 条加法指令实现。

下面 4 条指令是实现[X3,X2,X1,X0]和[Y3,Y2,Y1,Y0]模 2^n 加法的指令。一共需要 4 个 cycles、8 字节 flash。

```
add X0, Y0;
adc X1, Y1;
adc X2, Y2;
adc X3, Y3;
```

图12.　模 2^n 加

1.6小结

表 2 是各个操作消耗的时间、RAM 和 Flash 汇总。

表2. 基本操作时间空间消耗

Operation		T(cycles)	RAM(bytes)	Flash（bytes）
XOR		4	0	8
AND		4	0	8
S^1		5	0	10
S^2		10	0	20
S^3	No Mul	15	0	30
	Mul	14	0	20
S^8		0	0	0
S^{-1}		6	0	12
S^{-3}		18	0	36
S^{-8}		0	0	0
+		4	0	8

2. 算法比较

主要包括低 RAM（Minimal RAM）和高吞吐率（High-Throughput）两类。

1) **Minimal RAM**

Minimal RAM 侧重于消耗更少的 RAM。算法将轮密钥写在 Flash 中，加密时直接从 Flash 加载。

Minimal RAM 适用需要加密数据比较少的场景。从 Flash 加载比从 RAM 加载，每一轮只多 4 个 cycles，由于数据量不是很大，整个加密时间不会多很多；但轮密钥不消耗 RAM，可以为其它程序节省很多 RAM。

2) High-Throughput

当数据量比较大时，从 Flash 加载将会消耗比较多的时间，因此将轮密钥放在 RAM 中。有两种方式可以实现：a)初始密钥仍存放于 Flash 中，经过密钥编排计算后的轮密钥保存在 RAM 中供程序使用；b)直接将轮密钥全部放在 Flash 中，开始加密前先将轮密钥从 Flash 加载到 RAM，加密过程中则直接从 RAM 加载轮密钥（Minimal RAM 在加密时轮密钥仍然是从 Flash 加载的）。理论上，a)需要计算轮密钥，时间比 b)要多；但 Flash 的消耗比 b)要少（但目前通过 a)方式实现的

Speck 比 b)方式消耗 Flash 要多）。同时，将循环进行展开，减少循环控制语句的执行时间。

High-Throughput 适用于需要加密的数据量比较大的场景，通过空间来换取时间。

2.1 Simon Minimal RAM

包括从 RAM 加载明文，从 Flash 加载轮密钥，使用轮密钥进行加密以及将最终的密文放回 RAM 的所有时间；不包括明文的初始化。

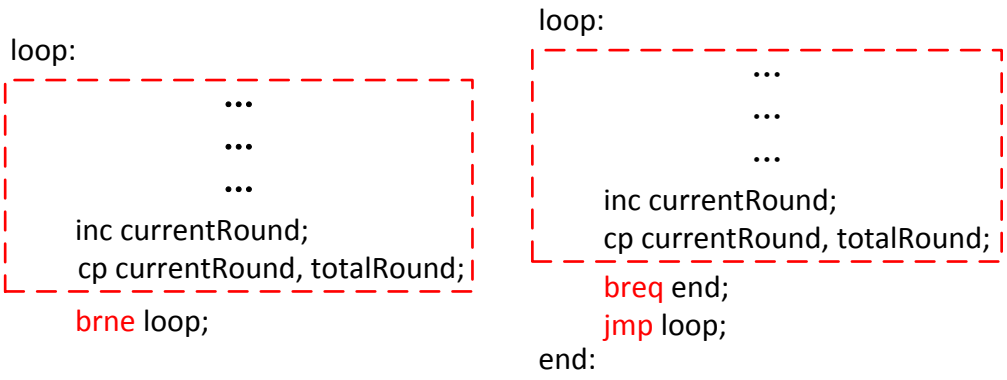
1) 轮函数：一轮所需要的 cycles 是 42，指令条数是 34。表 3 是 Simon 一轮的执行时间及指令条数。

表3. 一轮 Simon 指令

Mnemonic	Operation	Register Contents	Cycles	Instructions
Load	K ← k	K = k	12	4
Eor	K ← K⊕Y	K = k⊕y	4	4
Load	Y ← X	Y = x	2	2
Rotate	X ← S^1(X)	X = S^1(x)	5	5
Load	T ← X	T = S^1(x)	2	2
And	T ← T & S^8(Y)	T = S^1(x) &S^8(x)	4	4
Rotate	X ← S^1(X)	X = S^2(x)	5	5
Eor	X ← X⊕T	X = S^2(x) ⊕(S^1(x) &S^8(x))	4	4
Eor	X ← X⊕K	X = S^2(x) ⊕(S^1(x) &S^8(x)) ⊕y⊕k	4	4

2) 循环控制

循环控制需要通过跳转指令实现。AVR 跳转指令分为条件跳转和无条件跳转两种类型。条件跳转指令（比如 BRNE、BREQ 等）需要在一定的情况下跳转，一般根据标志寄存器的值判断是否跳转。条件跳转的范围较小，为[-64,63]（数值大小表示跳转目的地址与当前地址的差值，负数意味着向前跳转，正数则意味着向后跳转）；无条件跳转指令（比如 JMP）不需要依据任何寄存器的值，跳转范围比较大，为[0,4*2^{20}]。图 13 给出了两种情况的指令示意图。



a) 条件跳转示意图　　b) 条件跳转示意图

说明：

1. a)和b)实现同样的功能；
2. a) 中红色区域指令数量小于64；b) 中红色区域指令数量不小于64；
3. loop和end表示标签，用于条件跳转；
4. currentRound为一个寄存器，记录当前循环的次数；
5. totalRound为一个寄存器，记录总共应该循环的次数。

图13.　条件跳转和无条件跳转

由于轮函数指令只有 34 条，小于 64，可以直接使用条件跳转语句进行跳转，故循环控制只需要 3 条指令即可，共 3 cycles。

3)　明文加载密文写入

明文和密文都是 64 位，一条指令只能完成 8 位的转移，因此从 RAM 加载明文需要 8 条指令，将密文写入到 RAM 也需要 8 条指令，一共 16 条指令。RAM 和寄存器之间交换 1 字节（即 8 位）数据需要 2 个 cycles。故总时间为 32 cycles。图 14 a)表示从 RAM 加载明文到寄存器，其中 x 初始指向的是明文空间的首地址，一条指令表示将 x 所对应的内容从 RAM 中加载到寄存器后，x 的值加 1（即指向下一个数据）；图 14 b)表示将寄存器中的内容存放到 RAM，x 初始指向的是密文空间的首地址。

```
ld r0, x+ ;
ld r1, x+ ;
ld r2, x+ ;
ld r3, x+ ;
ld r4, x+ ;
ld r5, x+ ;
ld r6, x+ ;
ld r7, x+ ;
```

a)　加载明文

```
st x+, r0;
st x+, r1;
st x+, r2;
st x+, r3;
st x+, r4;
st x+, r5;
st x+, r6;
st x+, r7;
```

b)　写入密文

图14.　明文加载与密文写入

4)　初始化

初始化包括当前轮数置 0，总轮数设置，带进位加法源寄存器置 0，获取明文、密钥的地址。如图 15，一共是 7 条指令。

```
clr currentRound ;
ldi totalRound, 44;
clr zero;
ldi r26, low(plainText) ;
ldi r27, high(plainText);
ldi r30, low(keys<<1) ;
ldi r31, high(keys<<1) ;
```

图15.　Simon Minimal RAM 初始化

5)　程序返回

程序返回需要一条 ret 指令，共 4 个 cycles。

- 总时间 T(cycles)：

$$T = (42+3)\times 44+32+7+4 = 2023 cycles$$

- 加密效率 V(cycles/byte)：

$$V = T \div 8 \approx 253 cycles / byte$$

- Flash(bytes)：

$$S_{flash} = (34+3+16+7+1)\times 2+44\times 4 = 298 bytes$$

- RAM(bytes)：不考虑明文和密文占用的 RAM，RAM 消耗为 0。

表4. Simon Minimal RAM 时间空间消耗

类型	T	V	Flash	RAM
轮函数	42*44	253	34*2	-
循环控制	3*44		3*2	-
明文密文转移	32		16*2	-
寄存器初始化	7		7*2	-
程序返回	4		1*2	-
轮密钥	-		44*4	-
总量	2023		298	-

2.2 Simon High-Throughput

和 Minimal RAM 不同，High-Throughput 的实现中，轮密钥从 RAM 中加载。轮密钥可以直接放在 Flash 中，在加密前先加载到 RAM；或者通过密钥编排产生后直接放在 RAM 中。为了进一步减少时间，可以通过循环展开来减少循环控制

语句的执行次数，文中进行了 4 轮循环展开。

2.2.1 Without Key Schedule

1) 一次循环

每一轮的密钥从 RAM 中加载，加载 32 位轮密钥需要 8 cycles，相比从 Flash 加载减少了 4 个 cycles，因此一轮的时间是(42-4=)38 cycles，指令条数仍为 34。由于进行了 4 轮展开，因此一次循环的总时间是(38*4=)152 cycles，指令条数是(34*4=)136。

2) 循环控制

展开 4 轮后，指令条数超过 64，需要借助无条件跳转语句（JMP）来控制循环，循环控制指令由原先的 3 条增加到 4 条，因此需要 4 cycles。

3) 其它几项和 Minimal RAM 相同

- 总时间 T(cycles)：

$$T = (152+4)\times 11+32+7+4 = 1759 cycles$$

- 加密效率 V(cycles/byte)：

$$V = T \div 8 \approx 220 cycles / byte$$

- Flash(bytes)：只包括加密指令和轮密钥，不包括初始化、明文加载和密文写入指令

$$S_{flash} = (136+4)\times 2+44\times 4 = 456 bytes$$

- RAM(bytes)：由于不考虑明文和密文占用的 RAM，因此 RAM 消耗为轮密钥占用的空间。

$$S_{RAM} = 44\times 4 = 176 bytes$$

表5. Simon High-Throughput Without Key Schedule 时间空间消耗

类型	T	V	Flash	RAM
轮函数	152*11	220	136*2	-
循环控制	4*11		4*2	-
明文密文转移	32		-	-
初始化	7		-	-
程序返回	4		-	-
轮密钥	-		44*4	44*4
总量	1759		456	176

2.2.2 With Key Schedule

加密部分和 Without Key Schedule 的情况一样，只是增加了密钥编排指令。

相比加密大量数据所需要的时间，密钥编排的时间比较小。因此时间计算不包括密钥编排的时间，仍为 220 cycles/byte。加密指令条数不变，需要的 Flash 仍为(136*2+4*2=)280 bytes。只是不需要 176 bytes 来保存全部轮密钥，而是通过密钥编排实现。

Simon 密钥编排算法如式(1)。

$$k_{i+m} = c \oplus (z_j)_i \oplus k_i \oplus (I \oplus S^{-1}) \oplus (S^{-3}k_{i+3} \oplus k_{i+1}) \\ where: i \in [0, 40), m = 4, j = 3 \tag{1}$$

1) 一轮密钥编排，表 6 对应 1 轮的指令描述，一共需要 63 条指令

表6. Simon 一轮密钥编排

Mnemonic	Operation	Register Contents	Cycles	Instructions
Load	$K_i \leftarrow k_i$	$K_i = k_i$	4	4
Load	$K_{i+1} \leftarrow k_{i+1}$	$K_{i+1} = k_{i+1}$	4	4
Add, Adc	X ← X + 4	X=start address of k_{i+3}	2	2
Load	$K_{i+3} \leftarrow k_{i+3}$	$K_{i+3} = k_{i+3}$	4	4
Rotate	$K_{i+3} \leftarrow S^{-3}K_{i+3}$	$K_{i+3} = S^{-3} k_{i+3}$	18	18
Eor	$K_{i+3} \leftarrow K_{i+3}$ EOR K_{i+1}	$K_{i+3} = S^{-3} k_{i+3} \oplus k_{i+1}$	4	4
And	$K_i \leftarrow K_i$ EOR K_{i+3}	$K_i = k_i \oplus (S^{-3} k_{i+3} \oplus k_{i+1})$	4	4
Rotate	$K_{i+3} \leftarrow S^{-1}K_{i+3}$	$K_{i+3} = S^{-1}(S^{-3} k_{i+3} \oplus k_{i+1})$	6	6
Eor	$K_i \leftarrow K_i$ EOR K_{i+3}	$K_i = k_i \oplus (I \oplus S^{-1})(S^{-3} k_{i+3} \oplus k_{i+1})$	4	4
Eor	$C \leftarrow C$ EOR $(Z_3)_i$	$C = c \oplus (zj)i$	3	3
Eor	$K_i \leftarrow K_i$ EOR C	$K_i = c \oplus (zj)i \oplus k_i \oplus (I \oplus S^{-1})(S^{-3} k_{i+3} \oplus k_{i+1})$	4	4
Store	RAM ← K_i	$K_{i+4} = c \oplus (zj)i \oplus k_i \oplus (I \oplus S^{-1})(S^{-3} k_{i+3} \oplus k_{i+1})$	4	4
Sub, Sbc	X ← X - 16	X= start address of k_{i+1}	2	2

指令说明：在加载 k_{i+3} 前地址寄存器指向的是 k_{i+2} 的首地址，因此需要两条指令（add r26, four; adc r27, zero）将地址寄存器的值指向 k_{i+3}。由于常量 C 的值是 0xfffffffc，最低位为 0，C 和$(Z_3)_i$ 的异或可以直接将这个$(Z_3)_i$ 比特位移到 C 的最低位，需要 3 条指令。一轮结束后，地址寄存器指向 k_{i+5} 的首地址，下一轮循环

要加载轮密钥 k_{i+1}，因此需要两条指令（sub r26, sixteen; sbc r27, zero）将地址寄存器的值减 16。

2) 循环控制：常量 Z 的周期是 62，分为 8 个字节存储在内存中，每循环 8 次后需要重新加载常量 Z，同时要控制循环次数不超过 44。整个控制共需要 11 条指令。

```
        inc currentRound;
        inc remain8;
        cp remain8, eight;
        breq continue;;if the remain8 = 8
        jmp keysExtend;
    continue:
        clr remain8; start with 0 again
        ld currentZ, y+;
        cp currentRound, totalRound;
        breq encryption;
        jmp keysExtend;
```

图16. Simon 密钥编排循环控制

3) 常量：16 字节初始密钥、8 字节常量 Z 保存在 Flash 中；

- Flash(bytes)：**只包括密钥编排指令和加密指令，不包括寄存器初始化、常量初始化、初始密钥转移的指令**

$$S_{flash} = 280 + (63+11) \times 2 + 16 + 8 = 452 bytes$$

- RAM(bytes)：由于不考虑明文和密文占用的 RAM，因此 RAM 消耗为轮密钥占用的空间。

$$S_{RAM} = 44 \times 4 = 176 bytes$$

表7. Simon High-Throughput With Key Schedule 空间消耗

类型		Flash	RAM
加密		456-176=280	176
密钥编排	轮函数	63*2	-
	循环控制	11*2	-
	初始轮密钥	16	-
	常量	8	
总量		452	176

2.3 Speck Low-RAM

轮密钥放在 Flash 中，每轮从 Flash 加载轮密钥然后加密，最终将密文保存在 RAM。

1) 轮函数：一轮需要指令 33 条，共 41 cycles。

表8. 一轮 Speck 指令

Mnemonic	Operation	Register Contents	Cycles	Instructions
Load	K ← k	K = k	12	4
Add	X ← $S^8(S^{-8}(X) + Y)$	X = $S^8(S^{-8}(x) + y)$	4	4
Eor	K ← K⊕$S^{-8}(X)$	K = k⊕$(S^{-8}(x) + y)$	4	4
Rotate	Y ← $S^3(Y)$	Y = $S^3(y)$	15	15
Eor	Y ← Y⊕K	Y = $S^3(y)$ ⊕ $(S^{-8}(x) + y)$ ⊕k	4	4
Load	X ← K	X = k⊕$(S^{-8}(x) + y)$	2	2

2) 循环控制：一轮指令条数小于 64，只需要 3 条指令即可控制循环。
3) 明文加载密文写入：明文加载、密文写入个 8 条指令，共 16 条指令，32 cycles。
4) 初始化：7 条指令，7 个 cycles。
5) 程序返回：一条 ret 指令，4 个 cycles。

- 总时间 T(cycles)： $T = (41+3)\times 27+32+7+4 = 1231 cycles$
- 加密效率 V(cycles/byte)： $V = T \div 8 \approx 154 cycles/byte$
- Flash(bytes)： $S_{flash} = (33+3+16+7+1)\times 2+27\times 4 = 228 bytes$
- RAM(bytes)：为 0。

表9. Speck Low-RAM 时间空间消耗

类型	T	V	Flash	RAM
轮函数	41*27	154	33*2	-
循环控制	3*27		3*2	-
明文密文转移	32		16*2	-
初始化	7		7*2	-
程序返回	4		1*2	-

轮密钥	-		27*4	-
总量	1231		228	0

2.4 Speck Faster Low-RAM

1) 循环一次：在一次循环中执行两次轮函数，可以消除一轮中最后的转移指令。因此，一次循环需要(39*2=)78cycles、指令(31*2=)62 条。整个加密需要 27 轮，每次循环执行 2 轮，最后 1 轮单独做。
2) 循环控制：一次循环共(32*2=)62 条指令，加上循环控制的 2 条指令，一共 64 条，而条件跳转语句的范围不能超过 63，因此需要借助无条件跳转语句。整个循环控制指令共 4 条，需要 4 cycles。
3) 其它：16 条明文加载密文写入指令，共 32cycles；7 条初始化指令，共 7cycles；1 条程序返回指令，共 4cycles；最后一轮单独执行，需要 31 条指令（可以直接将密文写入 RAM，而不用先转移，所以不是 33 条指令），共 39cycles。

- 总时间 T(cycles)：

$$T=(39\times 2+4)\times 13+39+32+7+4=1148cycles$$

- 加密效率 V(cycles/byte)：

$$V=T\div 8\approx 144cycles/byte$$

- Flash(bytes)：

$$S_{flash}=(62+4+16+7+1+33)\times 2+27*4=354bytes$$

- RAM(bytes)：为 0。

表10. Speck Faster Low-RAM 时间空间消耗

类型	T	V	Flash	RAM
轮函数	39*27	144	(62+33)*2	-
循环控制	4*13		4*2	-
明文密文转移	32		16*2	-
初始化	7		7*2	-
程序返回	4		1*2	-
轮密钥	-		27*4	-
总量	1148		354	0

2.5 Speck High-Throughput

高吞吐率适用于数据量较大的情况，相比整个加密过程，密钥编排所需要的时间非常小。因此，时间计算不包括密钥编排的部分。

2.5.1 Without Key Schedule

循环左移 3 位通过乘法指令实现。

1) 轮函数

由 1.3 节可知，乘法指令实现循环左移 3 位，需要 10 条指令，14cycles。乘法指令实现 Speck 一轮的情况如表 6：

表11. 乘法指令实现 Speck 一轮

Mnemonic	Operation	Register Contents	Cycles	Instructions
Load	K ← k	K = k	8	4
Add	X ← $S^8(S^{-8}(X) + Y)$	X = $S^8(S^{-8}(x) + y)$	4	4
Eor	K ← K⊕$S^{-8}(X)$	K = k⊕$(S^{-8}(x) + y)$	4	4
Mul	X ← $S^3(Y)$	X = $S^3(y)$	14	10
Eor	X ← X⊕K	X = $S^3(y)$ ⊕ $(S^{-8}(x) + y)$ ⊕k	4	4

则乘法指令实现一轮需要 26 条指令，34cycles。由 3.1 节可知，乘法指令的结果并不是保存在原寄存器中。需要进行展开才能消除，通过 3 轮展开即可完全恢复由于乘法指令和减少移位指令带来的错位。这样实际上循环一轮需要执行轮函数 3 轮，指令(26*3=)78 条，时间(34*3=)102 cycles。

2) 循环控制

循环一轮的指令条数超过了 64，循环控制需要 4 条指令，共 4cycles。

3) 初始化

16 条明文加载密文写入指令，共 32cycles；8 条初始化指令（乘法指令的被乘数 8 需要通过一条指令放到寄存器汇总），共 8cycles；1 条程序返回指令，共 4cycles。

- 总时间 T(cycles)：

$$T = (34 \times 3 + 4) \times 9 + 32 + 8 + 4 = 998 cycles$$

- 加密效率 V(cycles/byte)：

$$V = T \div 8 \approx 125 cycles / byte$$

- Flash(bytes)：不包括明文初始化和全部轮密钥由 Flash 加载到 RAM 的指

令。

$$S_{flash} = (78+4+16+8+1)\times 2 + 27*4 = 322bytes$$

- RAM(bytes)：

$$S_{RAM} = 27*4 = 108bytes$$

表12. Speck High-Throughput Without Key Schedule 时间空间消耗

类型	T	V	Flash	RAM
轮函数	34*27	125	78*2	-
循环控制	4*9		4*2	-
明文密文转移	32		16*2	-
初始化寄存器	8		8*2	-
程序返回	4		1*2	-
轮密钥	-		27*4	27*4
总量	998		322	108

2.5.2 With Key Schedule

Speck 密钥编排函数如式(2)。

$$\begin{aligned} &l_{i+m-1} = (k_i + S^{-8} l_i) \oplus i; \\ &k_{i+1} = S^3 k_i \oplus l_{i+m-1}; \\ &where: K = (l_{m-2}, ..., l_1, l_0, k_0), i \in [0, 27), m = 4 \end{aligned} \tag{2}$$

密钥编排开始之前 16 字节的初始密钥 K 已分别加载到常量和轮密钥中。

1) 轮函数：l_i 只在计算 l_{i+3} 时用到了，为了让常量占用较少的空间，每一轮计算出 l_{i+3} 后，l_i 会被覆盖掉。表 13 为一轮密钥编排的流程。

表13. Speck 一轮密钥编排

Mnemonic	Operation	Register Contents	Cycles	Instructions
Ldi	Ldi r28,low(l) Ldi r29,high(l)	Y = the start address of l_i	2	2
Load	$K_i \leftarrow k_i$	$K_i = k_i$	4	4
Load	$L_i \leftarrow l_i$	$L_i = l_i$	4	4
Add, Adc	$L_i \leftarrow S^8(K_i + S^{-8}L_i)$	$L_i = S^8(k_i + S^{-8}l_i)$	4	4
Eor	$L_i \leftarrow L_i \oplus i$	$L_i = S^8((k_i + S^{-8}l_i) \oplus i)$	1	1

Rotate	$K_i \leftarrow S^3 K_i$	$K_i = S^3 k_i$	15	15
Eor	$K_i \leftarrow K_i \oplus L_i$	$K_i = S^3 k_i \oplus ((k_i + S^{-8} l_i) \oplus i)$	4	4
Store	$K_i \rightarrow k_{i+1}$	$k_{i+1} = S^3 k_i \oplus ((k_i + S^{-8} l_i) \oplus i)$	4	4
Sub, Sbc	$X \leftarrow X - 4$	X = the start address of k_{i+1}	2	2
Load, Store	$l_{i+1} \rightarrow l_i$ $l_{i+2} \rightarrow l_{i+1}$ $l_{i+3} \rightarrow l_{i+2}$	-	14	14

指令说明：i 用一个寄存器即可表示，第 5 步中的异或只需要将 L_i 的最低字节与 i 异或即可，因此只需要 1 条指令。第 8 步将 k_{i+1} 存放到 RAM 中，需要 4 条指令；存放后地址寄存器指向 k_{i+2} 的首地址，而下一次循环需要加载 k_{i+1}，因此需要通过两条指令(sub r26, four; sbc r27, zero)将地址寄存器指向 k_{i+1}。

2) 循环控制：54 小于 64，因此循环控制只需要 3 条指令。

3) 寄存器初始化：共 6 条指令。

```
ldi four, 4;
ldi twelve, 12;
clr currentRound;
ldi totalRound, 27;
ldi r26, low(keysRAM);
ldi r27, high(keysRAM);
```

图17. Speck 密钥编排寄存器初始化

4) 加密：和 Without Key Schedule 一样，需要(322-27*4=)214 bytes。

5) 程序返回：1 条指令。

6) 初始密钥：16 bytes。

- Flash(bytes)：不包括明文初始化、第一轮密钥加载和常量 L 前 3 个值的加载；

$$S_{flash} = (54+3+49+1)\times 2+16+214 = 444 bytes$$

- RAM(bytes)：

$$S_{RAM} = 27*4 = 108 bytes$$

表14. Speck High-Throughput With Key Schedule 空间消耗

类型		Flash	RAM
加密		322-108=214	108
密钥编排	轮函数	54*2	-

	循环控制	3*2	-
	初始化寄存器	6*2	-
	程序返回	1*2	-
	初始轮密钥	16	-
总量		358	108

2.5.3 Unroll 6 rounds

上面介绍的 Without Key Schedule 和 With Key Schedule 两种方法都是循环展开了 3 轮，通过展开更多的轮可以进一步提高加密速度，但同时指令数会增加很多。这里计算 Without Key Schedule 情况下循环展开 6 轮的时间和空间消耗。

类比循环展开 3 轮结果，循环展开 6 轮每轮的指令条数会增加一倍，有原先的(26*3=)78 条扩展到(26*6=)156 条。循环控制语句仍为 4 条。Speck 一共有 27 轮，每次展开 6 轮，一共循环 4 次，且最终会有 3 轮剩余，剩余 3 轮全部展开的话又需要指令数(26*3=)78 条，由于是最后 3 轮，因此不再需要另外的循环控制语句。其它初始化语句和展开 3 轮一样。循环展开 3 轮和循环展开 6 轮的时间空间消耗如表 15。

表15.　循环展开 3 轮和 6 轮时间空间对比图

类型	Unroll 3 rounds				Unroll 6 rounds			
	T	V	Flash	RAM	T	V	Flash	RAM
轮函数	34*27	125	78*2	-	34*27	123	156*2	-
循环控制	4*9		4*2	-	4*4		4*2	-
明文密文转移	32		16*2	-	32		16*2	-
初始化寄存器	8		8*2	-	8		8*2	-
程序返回	4		1*2	-	4		1*2	-
轮密钥	-		27*4	27*4	-		27*4	27*4
剩余 3 轮	-		-	-			78*2	-
总量	998		322	108	978		634	128

时间上：展开 6 轮比展开 3 轮循环控制语句少执行了 5 次，一共少 20 个 cycles；

Flash：展开 6 轮比展开 3 轮每次循环指令数增加了 1 倍（即 78*2=156 bytes），剩余的 3 轮全展开又需要 156 bytes，一共多了 312 bytes。

RAM：没有变化。

2.6 小结

2.6.1 算法结果

各种算法的执行时间，RAM 和 Flash 消耗情况如表 16。

表16. 算法时间空间对比

Algorithm		V(cycles/byte)	RAM(bytes)	Flash(bytes)
Simon Minimal RAM		253/264/253	0	298/300/290
Simon High-Throughput	Without Key Schedule	220/223/220	176	456/460/436
	With Key Schedule	220/223/220	176	452/460/436
Speck Low-RAM		154/158/154	0	228/228/218
Speck Faster Low-RAM		144/147/-	0	354/356/-
Speck High-Throughput	Without Key Schedule	125/127/125	108	322/324/316
	With Key Schedule	125/127/125	108	358/360/316
	Unroll 6 rounds	123/123/122	108	634/636/628

表 15 中对于加密速度 V、Flash，第一个数字表示分析计算的值；第二个数字表示实际在 Atmel 平台测得的值；第三个数字表示论文中给出的值。对于 RAM，三种情况都是一样的，因此只给出了一个数值。”-”表示论文中没有给出相应的值。

2.6.2 结果分析

1) Minimal RAM
   - Simon Minimal RAM 和 Speck Low-RAM 算法的数据与论文基本完全吻合；
   - Speck Low-RAM 比 Simon Minimal RAM 在加密速度上优 39.13%，Flash 消耗要优 23.49%；
   - Speck Faster Low-RAM 在加密速度上比 Simon Minimal RAM 优 43.08%，但 Flash 却要多 18.79%；
2) High-Throughput
   - Simon High-Throughput
     - With Key Schedule 和 Without Key Schedule 两种方法的时间和 RAM 与论文吻合。但 Flash 消耗都没有考虑寄存器初始化、明文加载密文写入、明文初始化的指令，这样得到的值才勉强和论文吻合。拿 Without

Key Schedule 的情况来说，理论上全部轮密钥需要(44*4=)176 bytes，循环展开 4 轮需要(34*4=)136 条指令（即 272 bytes），这两部分最少也需要 448 bytes，这已经超过了论文给出的 436 bytes，所以我认为论文中没有包括寄存器初始化、明文加载密文写入、明文初始化以及全部轮密钥由 Flash 加载到 RAM 的指令，因此我在统计 Simon High-Throughput 这两种方法的时候也没有考虑这四部分的指令。

- Simon High-Throughput 上次的结果中，Without Key Schedule 比 With Key Schedule 的 Flash 要小很多，后来发现是因为在计算 Without Key Schedule 的时候没有加上轮密钥占用 176 bytes 的 Flash，所以结果才会很意外。

- Speck High-Throughtpu
  - Without Key Schedule 没有包括明文的初始化、全部轮密钥的由 Flash 加载到 RAM 需要的指令；With Key Schedule 没有包括明文的初始化、第一轮子密钥和常量 L 前 3 个值的初始化需要的指令。
  - Speck High-Throughtpu Without Key Schedule 的结果和论文基本完全吻合，With Key Schedule 的 Flash 消耗比论文多 13.29%，可能是由于密钥编排的指令组织不合理。
  - Speck 在加密速度上比 Simon 优 43.18%；RAM 要优 38.64%；Flash 消耗要优 20.80%~29.39%。

3) 分析值和实际值比较

- 加密速度上，实际测得的值都比分析的值稍微大一点，一方面是因为程序返回指令实际测得的值是 5，但在分析计算时认为的是 4；另外，程序在运行时可能会有一些额外的计算（比如，计算跳转标签所表示的实际地址值等），而在分析时没有考虑这些，所以实际的值比分析的要稍微高一点。