RISC-V 密码学指令扩展（K扩展）概述

K扩展，即RISC-V Cryptographic Extension，密码学扩展，因为字母C被别的扩展用了，所以叫“K”。你可以在https://github.com/riscv/riscv-crypto找到完整版的草案。

该扩展提供了一系列密码学相关的指令，他们大多都和其他指令一样使用通用寄存器，保持最多两读一写的原则。相比于纯软件实现，使用这些指令可以提升密码学算法的速度，并降低应用程序的大小。

目前（2021/03/25，spec v0.90）大体为两个部分：

- 标量的NIST系列（AES、SHA2）和国产的商密（ShangMi）系列（SM4、SM3）的加速指令
  - 其中还包括了一些会被各类加密算法使用的，从B扩展（位操作扩展）复用的子集
- 为了生成随机数种子的，墒源（Entropy Source）指令

指令功能子集

由于包含了多种加密算法等功能，K扩展被划分为了多个功能集（Functional Set），用户可以根据实际需要选择要实现哪些子集，来更灵活的定制（图片来自草案手册）：

Functional Set	Description
`K`	The default scalar cryptography extension, short for `ZknZkr`
`Zkg`	Constant time carry-less multiply for Galois/Counter Mode.
`Zkb`	Bitmanip subset included in the scalar cryptography extension, minus those in `Zkg`.
`Zkr`	Entropy source for seeding random number generators.
`Zkn`	NIST algorithm suite. Short for `ZkneZkndZknhZkgZkb`.
`Zkne`	NIST AES Encryption Instructions.
`Zknd`	NIST AES Decryption Instructions.
`Zknh`	NIST SHA2 Hash function instructions.
`Zks`	ShangMi (SM) algorithm suite. Short for `ZksedZkshZkgZkb`.
`Zksed`	SM4 Instructions.
`Zksh`	SM3 Hash function instructions.

Table 1: Explanation of the feature strings used to refer to the functional sets.

有些功能集是其他功能集的简写，比如K就代表了ZkneZkndZknhZkgZkbZkr这一些列功能集，包含了全部的NIST系列算法和墒源指令等。

目前所有的功能集、特性集和具体指令的对应表如下（图片来自草案手册）：

Instructions	Functional Set	Feature Sets				
		Zkn (RV32)	Zkn (RV64)	Zks (RV32)	Zks (RV64)	Zkr
aes32dsi	Zknd	✔				
aes32dsmi	Zknd	✔				
aes32esi	Zkne	✔				
aes32esmi	Zkne	✔				
aes64ds	Zknd		✔			
aes64dsm	Zknd		✔			
aes64es	Zkne		✔			
aes64esm	Zkne		✔			
aes64im	Zknd		✔			
aes64ks1i	Zkne		✔			
aes64ks2	Zkne		✔			
sha256sig0	Zknh	✔	✔			
sha256sig1	Zknh	✔	✔			
sha256sum0	Zknh	✔	✔			
sha256sum1	Zknh	✔	✔			
sha512sig0h	Zknh	✔				
sha512sig0l	Zknh	✔				
sha512sig1h	Zknh	✔				
sha512sig1l	Zknh	✔				
sha512sum0r	Zknh	✔				
sha512sum1r	Zknh	✔				
sha512sig0	Zknh		✔			
sha512sig1	Zknh		✔			
sha512sum0	Zknh		✔			
sha512sum1	Zknh		✔			
sm3p0	Zksh			✔	✔	
sm3p1	Zksh			✔	✔	
sm4ed	Zksed			✔	✔	
sm4ks	Zksed			✔	✔	
pollentropy	Zkr					✔
getnoise	Zkr					✔
clmul, clmulh	Zkg	✔	✔	✔	✔	
xperm.n, xperm.b	Zkb	✔	✔	✔	✔	
ror, rol, rori	Zkb	✔	✔	✔	✔	
roriw	Zkb		✔		✔	
andn, orn, xnor	Zkb	✔	✔	✔	✔	
pack, packu, packh	Zkb	✔	✔	✔	✔	
packw, packuw	Zkb		✔		✔	
rev.b, rev8 (grevi)	Zkb	✔	✔	✔	✔	
rev8.w (grevi)	Zkb		✔		✔	
zip (shfli)	Zkb	✔	✔	✔	✔	
unzip (unshfli)	Zkb	✔	✔	✔	✔	

位操作部分（Zkb）

密码学算法中经常涉及各种特殊的位操作和变换，如果使用一些专用的指令来实现这些操作，将会大幅提升效率。

本部分复用了RISC-V B扩展的一个子集，保证了只要实现了Z或B这两个扩展之一，就可以使用这些指令。具体的指令说明可以参考B扩展的手册，这里仅介绍其在密码学中的用途。

循环移位（ror、rol等）用于SHA256，AES、ChaCha20、SM3、SHA512，SHA3等算法；

位&字节排列组合（rev.b、rev8等）广泛用于各类加密算法；

位插入、反插入指令（zip、unzip）用于在RV32中实现SHA3的64位旋转（rotations），RV64上则不需要；

无进位乘法（clmul，clmulh）用于实现Galois Counter Mode (GCM)，TLS 1.3就使用了这种算法；

带取反逻辑（andn、orn、xnor）可用于避免潜在的侧信道攻击；

装箱（pack、packu、packh）用于一些轻量级块密码器，以及处理算法的字节流输入；

Crossbar 组合（xperm.n、xperm.b）用于实现加密算法中常用的S-Box变换操作。

AES 加速指令（Zkne、Zknd）

这部分指令用于加速AES算法，这些指令也大体上保持了“在通用寄存器上两读一写”的RISC-V指令风格，这部分指令对于RV32和RV64是独立的，分别以aes32和aes64开头。

AES 加密简述

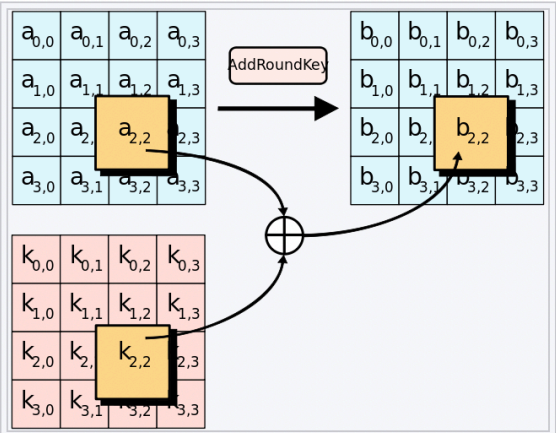
> **高级加密标准**（英语：**A**dvanced **E**ncryption **S**tandard，缩写：AES），又称**Rijndael加密法**（荷兰语发音：[ˈrɛindaːl]，音似英文的“Rhine doll”），是美国联邦政府采用的一种区块加密标准。这个标准用来替代原先的DES，已经被多方分析且广为全世界所使用。经过五年的甄选流程，高级加密标准由美国国家标准与技术研究院（NIST）于2001年11月26日发布于FIPS PUB 197，并在2002年5月26日成为有效的标准。现在，高级加密标准已然成为对称密钥加密中最流行的算法之一。
>
> ——节选自 Wikipedia

简单而言，AES的加密过程在一个4x4字节矩阵上进行，其被称为一个“体（state）”

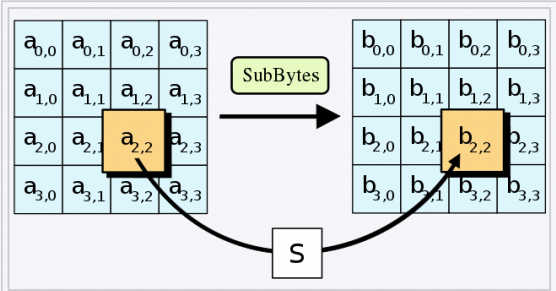
一个完整的加密步骤有10、12或14轮（根据key长度而不同），通常各轮加密分为以下4个步骤（最后一轮除外）：

1. AddRoundKey：矩阵中的每一个字节都与该轮的“回合密钥”做XOR（异或）运算
2. SubBytes：透过一个非线性的替换函数（S-box），替换每个字节
3. ShiftRows：每一列都向左循环位移某个偏移量
4. MixColumns：每行的4个元素通告一种线性变换互相组合

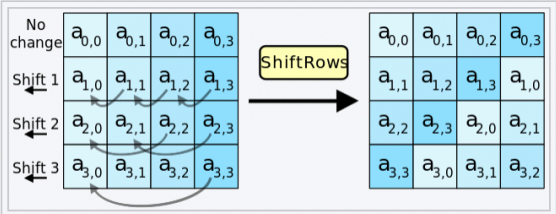
（图片来源：Wikipedia）



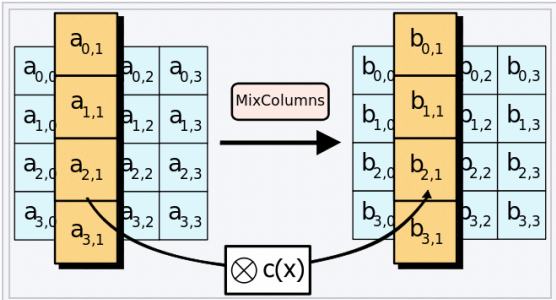
在AddRoundKey步骤中，将每个状态中的字节与该回合密钥做异或（⊕）。



在SubBytes步骤中，矩阵中各字节被固定的8位查找表中对应的特定字节所替换，S; $b_{ij} = S\ (a_{ij})$.



在ShiftRows步骤中，矩阵中每一列的各个字节循环向左方位移。位移量则随着列数递增而递增。



在MixColumns步骤中，每个直行都在modulo $x^4 + 1$之下，和一个固定多项式c(x)作乘法。

RV32 指令

```
aes32esi  rt, rs2, bs // Encrypt: SubBytes
aes32esmi rt, rs2, bs // Encrypt: SubBytes & MixColumns
aes32dsi  rt, rs2, bs // Decrypt: SubBytes
aes32dsmi rt, rs2, bs // Decrypt: SubBytes & MixColumns
```

aes32esmi和aes32dsmi可用一条指令实现SubBytes, ShiftRows和MixColumns操作，bs是一个2-bit的选择器，用于选择对一个word中的哪个byte进行操作。

aes32esi和aes32dsi不包含MixColumns操作，适用于AES加密的最后一轮。

RV64 指令

```
aes64ks1i  rd, rs1, rcon // KeySchedule: SubBytes, Rotate, Round Const
aes64ks2   rd, rs1, rs2  // KeySchedule: XOR summation
aes64im    rd, rs1       // KeySchedule: InvMixColumns for Decrypt
aes64esm   rd, rs1, rs2  // Round:    ShiftRows,    SubBytes,    MixColumns
aes64es    rd, rs1, rs2  // Round:    ShiftRows,    SubBytes
aes64dsm   rd, rs1, rs2  // Round: InvShiftRows, InvSubBytes, InvMixColumns
aes64ds    rd, rs1, rs2  // Round: InvShiftRows, InvSubBytes
```

64位的AES指令功能更为强大一些

首先aes64ks1i和aes64ks2共同实现了AES密钥生成算法，aes64im可用于其逆；

aes64esm、aes64es、aes64dsm、aes64ds的功能与32位的类似，区别在于在64位下可以通过rs1和rs2传入一个完整的128bit的体，然后在rd写入其中一半的结果，调换rs1和rs2的顺序即可得到另一半。

SHA-256、SHA-512 加速指令（Zknh）

这部分指令加速了SHA-256和SHA-512算法中的σ函数和Σ函数，这两个函数都有着一个32位或64位的输入和一个同样大小的输出。

SHA-2 简述

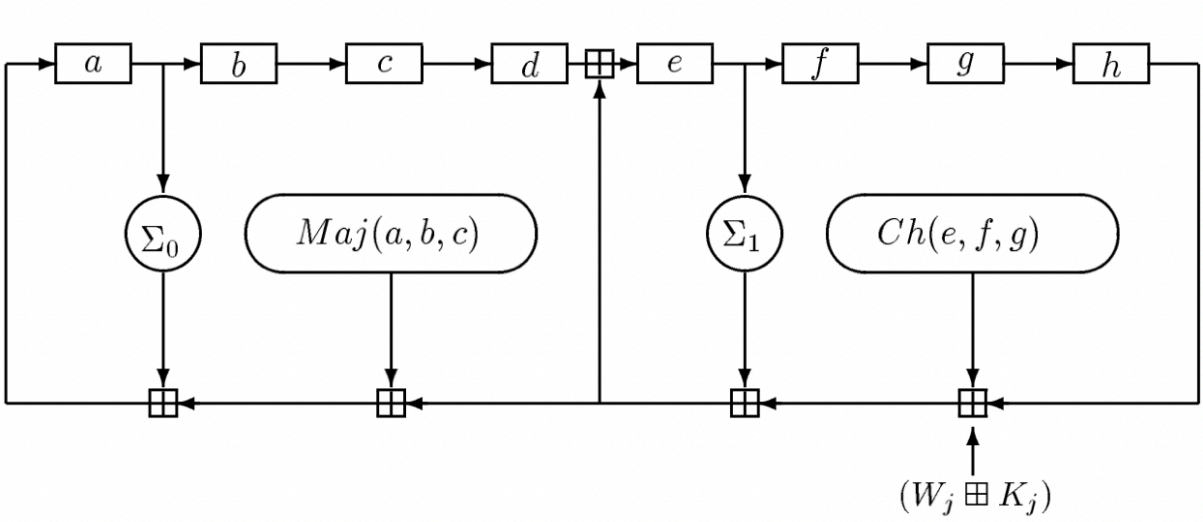
SHA-256和SHA-512都属于SHA-2（安全散列算法2），同样也由NIST发布，取代已经不再安全的SHA-1，是目前主流的散列算法。

以下以SHA-256为例简要的介绍其算法，SHA-512是类似的，详细资料可阅读：http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf，下文中部分图片也来自其中。

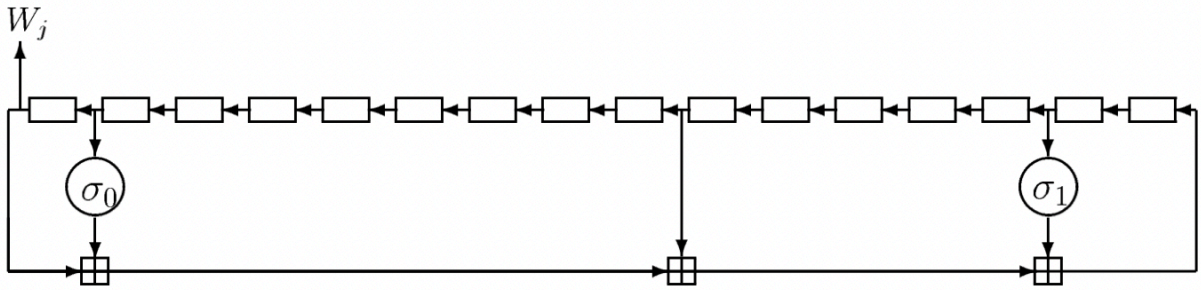
在SHA-256中，消息会被划分为512bit大小的块，以块为单位进行运算，每轮运算间会传递8个32bit的中间哈希值，初值为算法提供的固定常数，其后每轮运算用上一轮的8个中间哈希值和该块内容作为输入，计算得到新的8个中间哈希值，对最后一块运算完成后得到的8个中间哈希值即为我们要求出的256位哈希值。

每轮计算中有a, b, c, ... ,h这8个中间变量，初值为上一轮的8个中间哈希值，然后会运行64次**SHA-256压缩函数**，每次都会更新a～g这8个中间变量，最后用a～g这8个中间变量的值分别与上一轮的8个中间哈希值相加（模2^32），得到本轮新的8个中间哈希值。

SHA-256压缩函数的运算过程如下，角标j为压缩函数运行的代数，取值0～63：



其中Wj由以下运算得到，下图中16个长方形每个代表一个32bit的寄存器，将该轮的消息块线性分割为16个32bit部分，即为这16个寄存器的初值：



两张图中所用函数的定义如下：

$$
\begin{aligned}
Ch(x,y,z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x,y,z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) \\
\Sigma_0(x) &= S^2(x) \oplus S^{13}(x) \oplus S^{22}(x) \\
\Sigma_1(x) &= S^6(x) \oplus S^{11}(x) \oplus S^{25}(x) \\
\sigma_0(x) &= S^7(x) \oplus S^{18}(x) \oplus R^3(x) \\
\sigma_1(x) &= S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)
\end{aligned}
$$

我们在本部分中要通过专用指令加速的便正是上图中的σ函数和Σ函数，而对于Ch和Maj函数，由于他们都是3输入的函数，在RISC-V上实现相应的指令的编码代价较大，因此不被包含。

上图中函数的变量x的大小为一个SHA的word，SHA-256定义一个word为32bit，因此其的指令在RV32和RV64上是通用的，只是在RV64上相关寄存器只有低32位有效；而SHA-512的word为64bit，故其在RV32和RV64上是两组不同的指令，因为RV32上需要进行拆分。

SHA-256 加速指令

```
sha256sum0 rd, rs1
sha256sum1 rd, rs1
sha256sig0 rd, rs1
sha256sig1 rd, rs1
```

这些指令在RV32和RV64种通用，对应了上文提到的SHA-256算法中的这些函数：

$$\Sigma_0(x) = S^2(x) \oplus S^{13}(x) \oplus S^{22}(x)$$
$$\Sigma_1(x) = S^6(x) \oplus S^{11}(x) \oplus S^{25}(x)$$
$$\sigma_0(x) = S^7(x) \oplus S^{18}(x) \oplus R^3(x)$$
$$\sigma_1(x) = S^{17}(x) \oplus S^{19}(x) \oplus R^{10}(x)$$

其中Sn表示右移位n位，Rn表示右循环移位n位

SHA-512 加速指令

与SHA-256下类似，区别在于SHA-512下σ函数和Σ函数的输入和输出都为64bit，需要对于RV32和RV64需要分开考虑。

$$\Sigma_0(x) = S^{28}(x) \oplus S^{34}(x) \oplus S^{39}(x)$$
$$\Sigma_1(x) = S^{14}(x) \oplus S^{18}(x) \oplus S^{41}(x)$$
$$\sigma_0(x) = S^1(x) \oplus S^8(x) \oplus R^7(x)$$
$$\sigma_1(x) = S^{19}(x) \oplus S^{61}(x) \oplus R^6(x)$$

RV32

```
sha512sum0r rd, rs1, rs2
sha512sum1r rd, rs1, rs2
sha512sig0l rd, rs1, rs2
sha512sig0h rd, rs1, rs2
sha512sig1l rd, rs1, rs2
sha512sig1h rd, rs1, rs2
```

由于RV32的字长限制，需要对64bit的输出进行拆分，而64bit的输入由rs1和rs2共同提供。

对于`sig*[l|h]`，l和h分别表示输出低32bit和高32bit；

对于`sum*r`，由于其数学特性，可通过交换rs1和rs2的顺序来选择输出高32bit还是低32bit。

RV64

```
sha512sig0 rd, rs1
sha512sig1 rd, rs1
sha512sum0 rd, rs1
sha512sum1 rd, rs1
```

由于本身就是64bit字长，所以与前述的SHA-256类似，不需要拆分处理。

SM3 加速指令（Zksh）

SM3是和SHA-256同类的算法，其指令的功能也是类似的。

SM3属于国密算法，由国家密码管理局发布，是我国自主设计的一种散列算法，算法整体流程和前述的SHA-256类似，当然具体的算法和公式上和SHA-256是不同的，详细资料可以参考：https://sca.gov.cn/sca/xwdt/2010-12/17/1002389/files/302a3ada057c4a73830536d03e683110.pdf

在指令中包含的是SM3的两个置换函数，作用类似SHA-256的Σ函数和σ函数，其定义如下：

$$P_0(X) = X \oplus (X \lll 9) \oplus (X \lll 17)$$

$$P_1(X) = X \oplus (X \lll 15) \oplus (X \lll 23)$$

式中X为字。

对应了如下两个指令：

```
sm3p1 rd, rs1
sm3p0 rd, rs1
```

由于字长是32bit，这两条指令对于RV32和RV64是通用的。

SM4 加速指令（Zksed）

SM4是一种分组密码算法，也由国家密码管理局发布。

网上的公开资料不是很多，所以这里就不再介绍其算法了。

这部分指令也是RV32和RV64通用的：

```
sm4ed     rt, rs2, bs
sm4ks     rt, rs2, bs
```

sm4ed是加密/解密的指令，为轮函数中SBox和L变换的部分；

sm4ks是密码扩展的指令，为密码扩展中SBox和L'变换的部分

墒源扩展 (Zkr)

本部分是一个墒源接口，可用于为真随机数生成器（TRNG）提供种子，来提供密码学等级的随机数。

pollentropy 指令

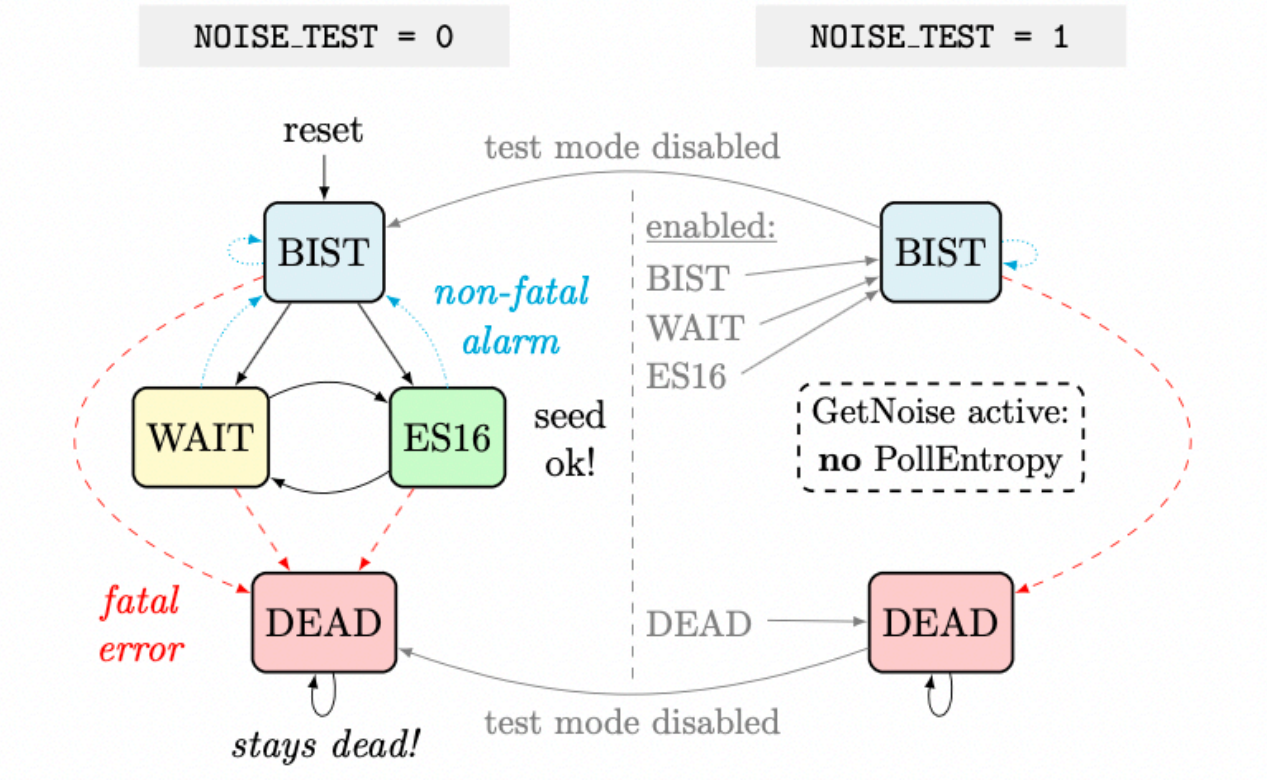
```
pollentropy    rd   // Poll randomness. Encoding: csrrs rd, mentropy, x0
```

这个指令是RV32和RV64通用的，用于"拉取墒"，其只有一个目标寄存器，返回的值含义如下：

Bits	Name	Description
63:32	*Set to 0*	Upper bits are set to zero in RV64.
31:30	OPST	Status: BIST (00), ES16 (01), WAIT (10), DEAD (11).
29:24	*reserved*	For future use by the RISC-V specification.
23:16	*custom*	Reserved for custom and experimental use.
15: 0	seed	16 bits of randomness, only when OPST=ES16.

该指令是非阻塞的，调用后会立即返回结果，其中OPST表示状态，如果为ES16，那么低16位就是一个有效的随机值。

几种状态的转移图如下：



getnoise指令

如果NOISE_TEST为1，则表示是测试模式，可以通过getnoise指令得到原始的“噪音”（pollentropy的随机值就是从中生成的），而pollentropy指令的功能会被禁用。

```
getnoise    rd   // Noise source test. Encoding: csrrs rd, mnoise, x0
```

这条指令也是RV32和RV64通用的。