南京郵電大學
Nanjing University of Posts and Telecommunications

最大化定制RISC-V向量扩展
——对SHA-3加速的潜力

"Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions," 2023 Design, Automation & Test in Europe Conference & Exhibition (DATE), Antwerp, Belgium, 2023, pp. 1-6.

汇报人： 王诗琴

目录

CATALOG

01 研究背景
Study on the background

02 理论支撑
Theoretical support

03 关键技术
Key technologies

04 实验分析
Experimental Analysis

05 论文总结
Summary of papers

01 研究背景

Study on the background

- 研究背景
- 关键技术
- 论文总结
- 理论支撑
- 实验分析

背景与动机

数据完整性的重要性

数据完整性是信息安全的基础之一，确保信息在传输和存储过程中未被未经授权地修改或破坏；
后量子密码学，是密码学的一个研究领域，能够抵抗量子计算机对现有密码算法攻击。

SHA-3标准及其在后量子密码学竞赛中的应用

安全哈希算法（SHA）是由美国国家标准与技术研究所（NIST）发布的一个加密哈希函数家族，在数据完整性验证领域有广泛的应用；
SHA-3，最新的一代，在NIST后量子密码学（PQC）竞赛中被用于许多候选算法；

Keccak-f[1600]置换算法是SHA-3的核心组成

Keccak-f[1600]置换算法是SHA-3的核心组成部分，具有高计算密集度；
Keccak-f[1600]内部状态为5x5x64比特矩阵，当前实现通常以较小的数据块（如64比特）顺序处理，存在显著的并行计算潜力；

RISC-V向量扩展的优势

RISC-V（精简指令集计算机第五代）是一种现代的开源指令集架构，以其开放标准、模块化设计和高度灵活性而著称。它允许开发者根据特定应用需求自由选择并扩展基础指令集，从而定制最适合其硬件平台的功能特性。

■ 特性1

平台	寄存器数量
ARM	16
X86	8
RISCV	32

■ 特性2

指令类型	用途
配置设置指令	定义了VL、LMUL、ELEN等参数信息
向量加载/存储指令	用于存取寄存器
向量算术指令	用于算数和逻辑运算

特性3

许多向量指令都支持掩蔽，并且可以应用于向量寄存器中向量元素的特定位置。向量加载和存储指令和向量算术指令中的vm字段表示相应的指令是否被屏蔽。

特性4

SIMD处理器需要根据LMUL进行向量地址重映射。图1显示了指令{vadd.vv v0、v0、v2}的工作过程。向量v0和v2的第一向量寄存器中的元素被同时读出，并以相同的元素索引号发送到各自的执行模块，用于加法操作。过程完成后，每个执行子模块的结果将根据元素索引号发送到向量v0。之后，v1和v3中的所有元素都将被获取并执行，每个执行子模块的结果将被写回向量v1。

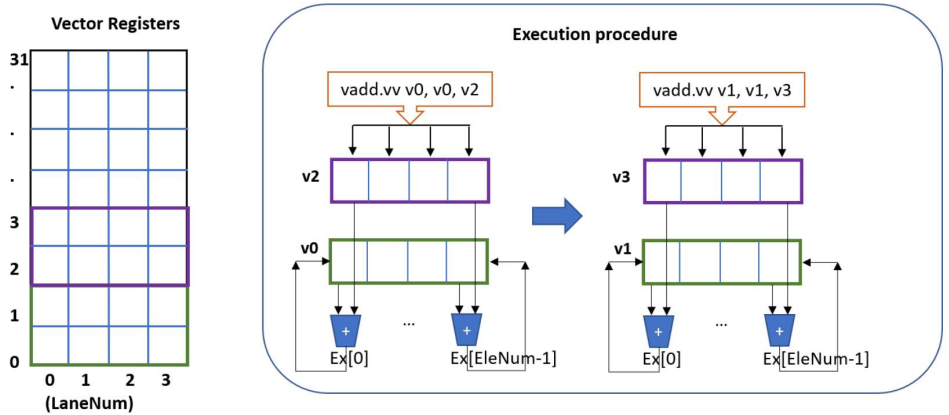


Fig. 1: Vector register file and address allocation [7].

研究内容

本文首次通过在32位和64位RISCV架构上的自定义向量扩展，探索了基于RISCV的Keccak-f[1600]并行化的全部潜力。研究内容如下：

1. 使用RISC-V向量扩展来向量化SHA-3函数的Keccak-f[1600]排列。本篇第一个使用这些扩展来加速SHA-3。
2. 分析Keccak排列中的五步映射，为32位和64位架构提出了10个自定义向量扩展，并在一个用系统版本编写的SIMD处理器中实现了所有这些扩展。
3. 使用自定义和现有的RISCV向量扩展，为32位和64位架构优化了Keccak程序。结果表明，ASIP设计结果明显优于所有以前提出的实现。

注：并行化是指在计算过程中，将原本需要按照一定顺序逐个执行的操作分解为多个可以同时进行的子任务，从而利用多核处理器、多处理器系统或者分布式计算环境中的硬件资源，在同一时间段内完成更多的计算工作。这种技术旨在通过增加并发性来提高系统的整体性能和效率。本文使用向量化处理的并行方式，通过SIMD（单指令多数据）架构，允许单个指令同时对多个数据元素执行相同的操作，极大地提高了运算效率，特别适合处理大型矩阵或数组类的数据结构。

02

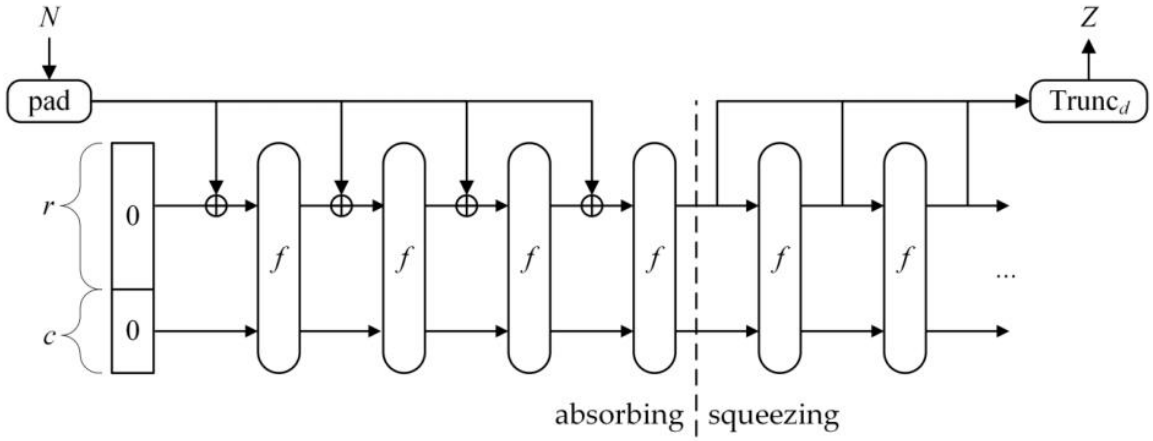
理论支撑

Theoretical support

- 研究背景
- 关键技术
- 论文总结
- 理论支撑
- 实验分析

Keccak-f[1600]算法

哈希函数：任意长度的输入经过变换后产生固定长度的输出（例如，SHA3-512生成512位摘要）
海绵构造法Keccak-f[r+c=b]置换操作，其中 $b \in \{25,50,100,200,400,800,1600\}$



SHA3-224(M) = KECCAK[448] (M || 01, 224);
SHA3-256(M) = KECCAK[512] (M || 01, 256);
SHA3-384(M) = KECCAK[768] (M || 01, 384);
SHA3-512(M) = KECCAK[1024] (M || 01, 512).

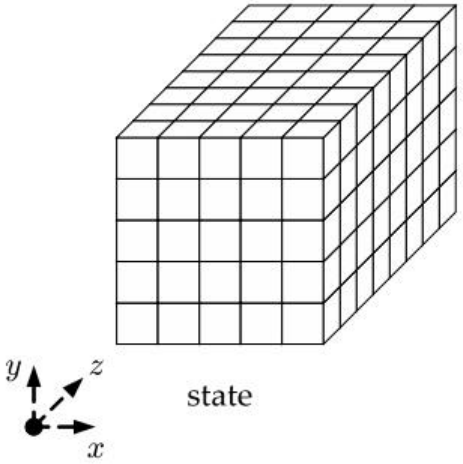
SHAKE128(M, d) = KECCAK[256] (M || 1111, d),
SHAKE256(M, d) = KECCAK[512] (M || 1111, d).

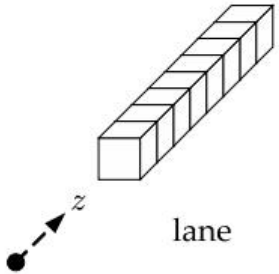
Figure 7: The sponge construction: Z=SPONGE[f, pad, r](N, d) [4]

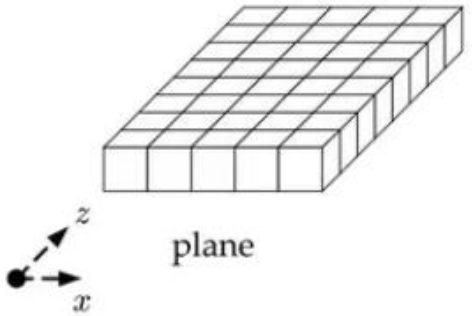
Keccak-f[1600]算法

For Keccak-f[1600], b = 1600, n_r = 24 rounds

|state| = 5*5*64 0<= x <=5, 0<= y <=5, 0<= w <=64
|lane| = 64 bits = a cpu word

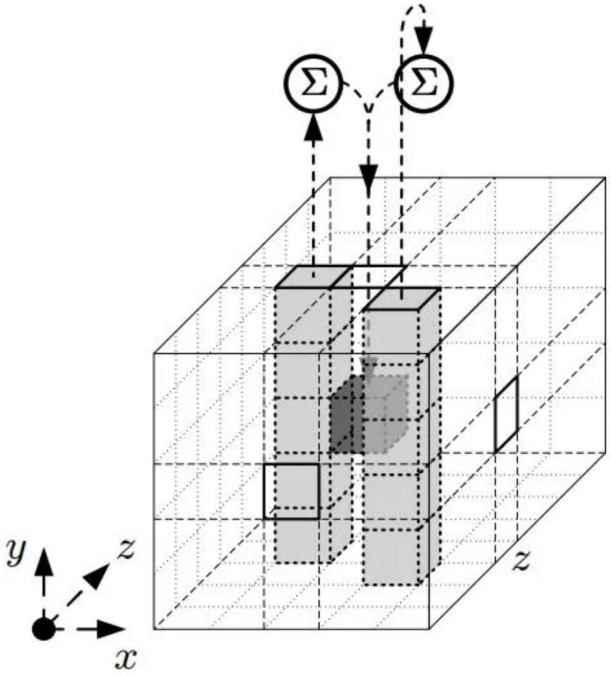






a triple (x, y, z) can describe a state, using formula: A[x, y, z]=S[w(5y+x)+z]

Keccak-f[1600]算法



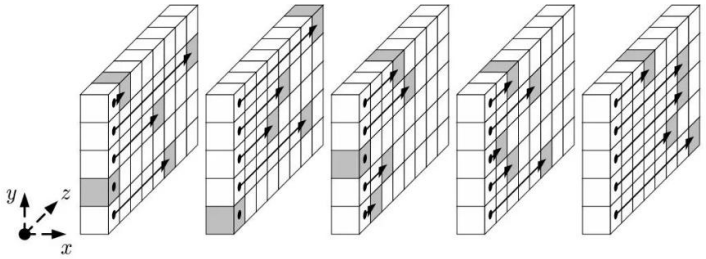
1) θ step mapping:
for $x = 0$ to 4 do
  $\mathbf{B}[x] = \mathbf{A}[x, 0] \oplus \mathbf{A}[x, 1] \oplus \mathbf{A}[x, 2] \oplus \mathbf{A}[x, 3] \oplus \mathbf{A}[x, 4]$
end for
for $x = 0$ to 4 do
  $\mathbf{C}[x] = \mathbf{B}[(x - 1) \bmod 5] \oplus \mathrm{ROT}(\mathbf{B}[(x + 1) \bmod 5], 1)$
end for
for $y = 0$ to 4 do
  for $x = 0$ to 4 do
    $\mathbf{D}[x, y] = \mathbf{A}[x, y] \oplus C[x]$
  end for
end for

Keccak-f[1600]算法

ρ : $a[x][y][z] \leftarrow a[x][y][z-(t+1)(t+2)/2]$,

with t satisfying $0 \le t < 24$ and $\begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^t \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix}$ in $\mathrm{GF}(5)^{2\times 2}$,

or $t = -1$ if $x = y = 0$,



2) ρ step mapping:

for $y = 0$ to 4 do

  for $x = 0$ to 4 do

  $\mathbf{E}[x, y] = \mathrm{ROT}(\mathbf{D}[x, y], r[x, y])$

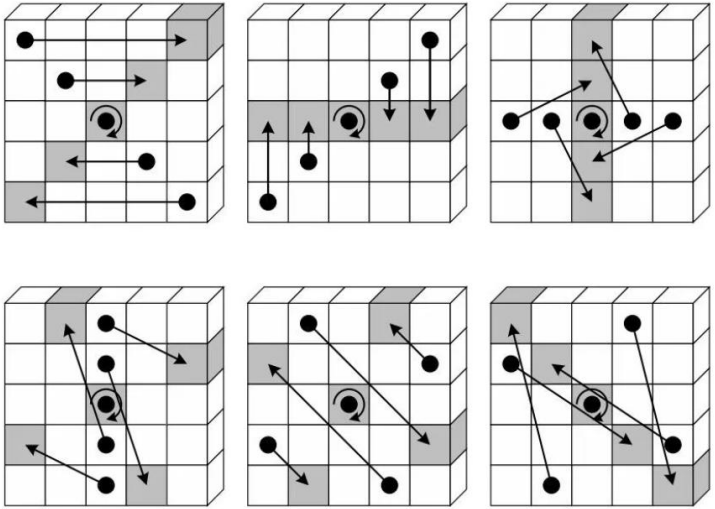
  end for

end for

The constants $r[x, y]$ are the cyclic shift offsets and are specified in the following table.

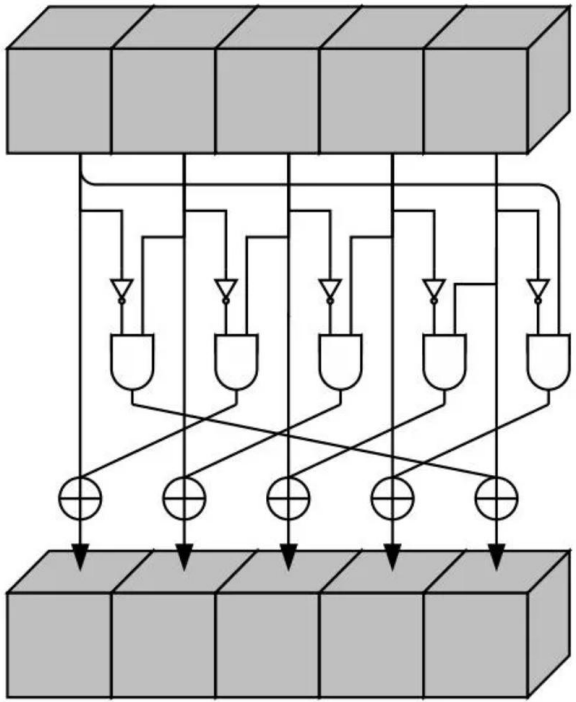
	$x=3$	$x=4$	$x=0$	$x=1$	$x=2$
$y=2$	25	39	3	10	43
$y=1$	55	20	36	44	6
$y=0$	28	27	0	1	62
$y=4$	56	14	18	2	61
$y=3$	21	8	41	45	15

Keccak-f[1600]算法



3) π step mapping:
for $y = 0$ to 4 do
  for $x = 0$ to 4 do
    $\mathbf{F}[x, y] = \mathbf{E}[(x + 3y) \bmod 5, x]$
  end for
end for

Keccak-f[1600]算法



4) χ step mapping:
for $y = 0$ to 4 do
  for $x = 0$ to 4 do
    $\mathbf{G}[x, y] = (\mathbf{F}[(x + 1) \bmod 5, y] \oplus 1) \cdot \mathbf{F}[(x + 2) \bmod 5, y]$
    $\mathbf{H}[x, y] = \mathbf{F}[x, y] \oplus \mathbf{G}[x, y]$
  end for
end for
5) ι step mapping:
$\mathbf{H}[0, 0] = \mathbf{H}[0, 0] \oplus \mathrm{RC}[i]$

03 关键技术

Key technologies

- 研究背景
- 关键技术
- 论文总结
- 理论支撑
- 实验分析

Key Technologies

Key Technologies for Maximizing the Potential of Customized RISC-V Vector Expansion for Sha3 Acceleration

Step 1

64位架构优化方案

Step 2

32位架构优化方案

Step 3

定制RISC-V向量指令集设计

64位架构优化方案

我们将使用的SIMD处理器（包含一个标量核心和一个向量处理单元）来研究SHA- 3的性能改进，其目标是低延迟和高吞吐量。

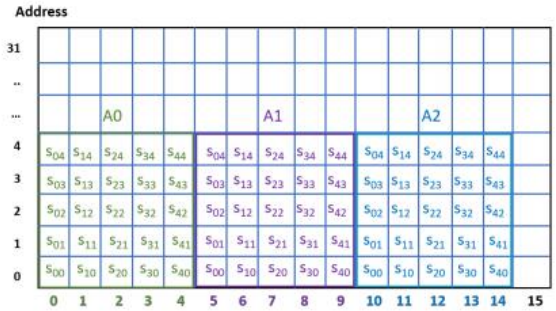


Fig. 2: Memory allocation for Keccak states in the 64-bit architecture.

对于64位的体系结构，我们将ELEN设置为64位，以便使SIMD处理器的向量处理器单元处理64位操作数。

Keccak-f[1600]很容易映射到64位体系结构，因为它在Keccak状态下的车道宽度与向量寄存器中的元素长度**兼容**。

32位架构优化方案

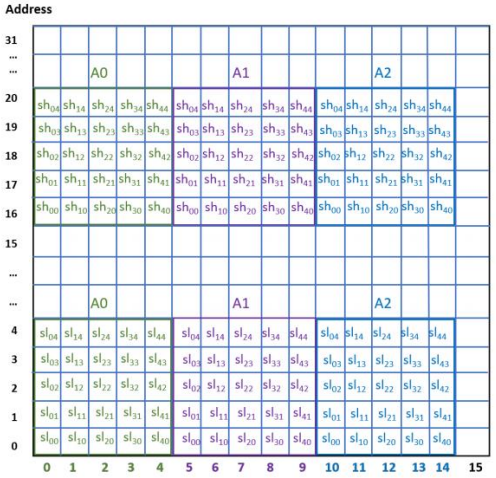


Fig. 3: Memory allocation for Keccak states in the 32-bit architecture

对于32位的体系结构，我们将SIMD处理器的ELEN参数设置为32位。使用位交错技术，即将一个64位通道被编码为两个32位，其中一个包含偶数z坐标的通道位，另一个包含奇数z坐标的通道位。

$$L[z] = a[x][y][z] \begin{cases} U_0[j] = L[2j] \\ U_1[j] = L[2j+1] \end{cases}$$

将这两个部分分别存储在向量寄存器文件中，如图3所示。

定制RISC-V向量指令集设计

文章提出了对SHA-3的自定义向量扩展，并通过在SIMD处理器中的SystemVerilog来实现它们。

定义了参数SN来表示并行工作的Keccak状态的数量。5×SN不应该大于一个向量寄存器中的元素数。在下面的几个部分中：

- vd表示目标向量操作数；
- vs1和vs2表示源向量操作数；
- uimm定义了无符号立即数；
- simm指定了有符号立即数；
- rs1指定标量寄存器操作数；
- vm表示是否启用了向量掩蔽。

在这个设计中，我们使用自定义指令和重写未使用的现有指令来扩展RISC-V中的指令。

TABLE I: Vector instructions and latency. * denotes ⌈VL/EleNum⌉.

Instruction	Description	Latency

vslidedownm.vi vd, vs2, uimm, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	$vd[5\times i+j] \leftarrow vs2[5\times i+(j+uimm) \bmod 5]$
	end for
	end for
		1+*

vslideupm.vi vd, vs2, uimm, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	$vd[5\times i+j] \leftarrow vs2[5\times i+(j-uimm) \bmod 5]$
	end for
	end for
		1+*

vrotup.vi vd, vs2, uimm, vm	$vd \leftarrow (vs2 \ll uimm) \vee (vs2 \gg (64-uimm))$ Note: $\vee$ denotes a bit-wise OR operation.	1+*
v32lrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1) \gg 63))[31:0]$ Note: $vs2 \parallel vs1$ is the concatenation of $vs2$ and $vs1$, to build 64-bit word.	1+*
v32hrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1) \gg 63))[63:32]$	1+*

v64rho.vi vd, vs2, simm, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	$vd[5\times i+j] \leftarrow (vs2[5\times i+j] \ll rho\_shift[simm][j]) \vee (vs2[5\times i+j] \gg (64-rho\_shift[simm][j]))$
	end for
	end for
	Note: if $simm$ is -1, the five rows process in sequence. The counter *lmul_cnt* in hardware indexes the row.
		1+*

v32lrho.vi vd, vs2, vs1, vm	1) $vs2 \parallel vs1$; 2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table; 3) The same process as *v64rho* is executed, and the least significant 32 bits are stored.	1+*
v32hrho.vi vd, vs2, vs1, vm	1) $vs2 \parallel vs1$; 2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table; 3) The same process as *v64rho* is executed, and the most-significant 32 bits are stored.	1+*
vpi.vi vd, vs2, simm, vm	The process is illustrated in Figure 5 1) Reading elements from *vs2* in the vector register file and re-arranging the elements into columns. 2) Storing each column in the vector register with the starting address of the column equals to *vd*. 3) If $simm$ equals 0, 1, 2, 3, or 4, only one row is processed. If $simm$ is -1, the five rows process in sequence. *lmul_cnt* in hardware indexes the row.	2+*

viota.vx vd, vs2, rs1, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	if($j \equiv 0$)
	$vd[5\times i+j] \leftarrow vs2[5\times i+j] \oplus RC[rs1]$
	else
	$vd[5\times i+j] \leftarrow vs2[5\times i+j]$
	end for
	end for
	Note: RC are round constant data.
		1+*

定制RISC-V向量指令集设计

向量滑动模五指令

在θ步映射中，中间值在XORing所有平面后对应的向量寄存器上上下移动。此外，在χ步映射中，所有平面必须分别以offset 1和2向下移动相应的向量寄存器。我们为这两种架构提出了两种扩展：*vslidedownm*滑动向下移动操作，和 *vslideupm*滑动向上移动操作。为了防止属于不同Keccak状态的lane受干扰，我们使用模五操作来限制元素索引数，如图4所示。

vslidedownm.vi vd, vs2, uimm, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	$vd[5\times i+j] \leftarrow vs2[5\times i+(j+uimm) \bmod 5]$
	end for
	end for

vslideupm.vi vd, vs2, uimm, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	$vd[5\times i+j] \leftarrow vs2[5\times i+(j-uimm) \bmod 5]$
	end for
	end for

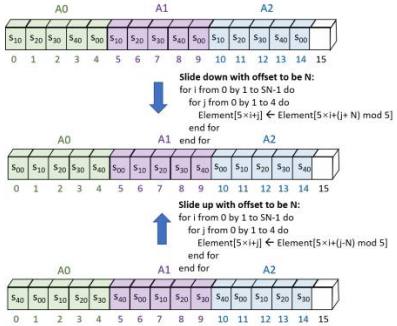


Fig. 4: Vector slide and modulo-five instructions. SN denotes the number of Keccak states. N is the offset. Here, we take the offset of 1 as an example.

定制RISC-V向量指令集设计

vrotup.vi vd, vs2, uimm, vm	$vd \leftarrow (vs2 \ll uimm) \vee (vs2 \gg (64 - uimm))$ Note: $\vee$ denotes a bit-wise OR operation.
v32lrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1) \gg 63))[31:0]$ Note: $vs2 \parallel vs1$ is the concatenation of $vs2$ and $vs1$, to build 64-bit word.
v32hrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1) \gg 63))[63:32]$

v64rho.vi vd, vs2, simm, vm	for i from 0 by 1 to $SN - 1$ do
	for j from 0 by 1 to 4 do
	$vd[5 \times i + j] \leftarrow (vs2[5 \times i + j] \ll rho\_shift[simm][j]) \vee (vs2[5 \times i + j] \gg (64 - rho\_shift[simm][j]))$
	end for
	end for
	Note: if $simm$ is -1, the five rows process in sequence. The counter *lmul_cnt* in hardware indexes the row.

v32lrho.vi vd, vs2, vs1, vm	1) $vs2 \parallel vs1$; 2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table; 3) The same process as *v64rho* is executed, and the least significant 32 bits are stored.
v32hrho.vi vd, vs2, vs1, vm	1) $vs2 \parallel vs1$; 2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table; 3) The same process as *v64rho* is executed, and the

向量旋转

使用旋转操作有两个步骤的映射：θ和ρ。

在θ步映射中，右列的奇偶性向最重要的方向旋转1位。**对于64位架构**，提出了具有两个向量操作数和一个直接值的旋转操作*vrotup*，它定义了偏移量。

在ρ步骤中，每个通道旋转一个可变数量的位置。**对于64位架构**，创建了*v64rho*。 当立即值为-1时，意味着所有五个平面都将按顺序执行；对于立即值为0到4的情况，只有一个平面会被操作，而操作的行索引由立即值决定。

定制RISC-V向量指令集设计

π步骤映射包括两个步骤：
1)按顺序从向量寄存器文件中读取每一行，并将元素重新排列成列；
2)将每一个列存储在向量寄存器中。列号等于Keccak状态号SN。该操作如图5所示。
我们在SIMD处理器中添加了执行模块和向量寄存器文件之间的接口，以使列模式下的数据写入可用。我们提出了一种新的自定义扩展vpi。

向量π指令

vpi.vi vd, vs2, simm, vm	The process is illustrated in Figure 5 1) Reading elements from *vs2* in the vector register file and re-arranging the elements into columns. 2) Storing each column in the vector register with the starting address of the column equals to *vd*. 3) If $simm$ equals 0, 1, 2, 3, or 4, only one row is processed. If $simm$ is -1, the five rows process in sequence. *lmul_cnt* in hardware indexes the row.

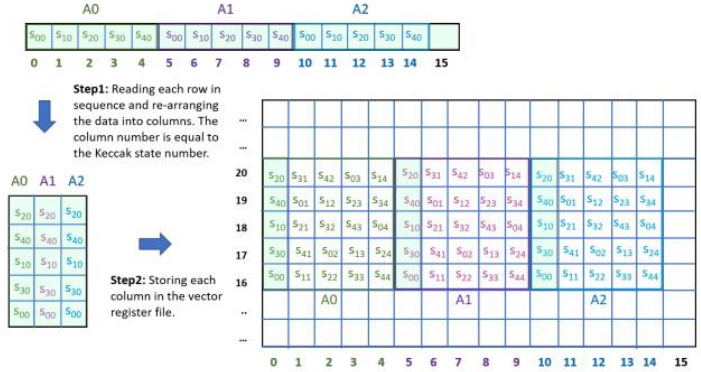


Fig. 5: π operation in the design.

定制RISC-V向量指令集设计

5) ι step mapping:
$\mathbf{H}[0,0] = \mathbf{H}[0,0] \oplus \mathrm{RC}[i]$

viota.vx vd, vs2, rs1, vm	for i from 0 by 1 to $SN-1$ do
	for j from 0 by 1 to 4 do
	if($j \equiv 0$)
	$vd[5\times i+j] \leftarrow vs2[5\times i+j] \oplus RC[rs1]$
	else
	$vd[5\times i+j] \leftarrow vs2[5\times i+j]$
	end for
	end for
	Note: RC are round constant data.

ι步骤，将轮常量与状态的特定部分进行XOR操作。
viota指令是为了实现这一操作而设计的。它取两个操作数：一个向量寄存器和一个标量寄存器。向量寄存器包含Keccak状态的数据，而标量寄存器用于索引轮常量的数据。
对于64位架构，轮常量的数据宽度是64位，这意味着整个轮常量可以与Keccak状态的相应部分进行XOR操作。

向量ι指令

04

实验分析

Experimental Analysis

- 研究背景
- 关键技术
- 论文总结
- 理论支撑
- 实验分析

性能比较与实验结果

TABLE II: Results of our 64-bit architectures and comparison with a 64-bit reference architecture. The execution time for one round is reported as the number of cycles to complete one round (cyc/rnd). The execution time to complete the entire permutation is reported as the number of cycles per byte (cyc/byte).

Implementation	Execution time		throughput (bits /cycle)	Area (slices)	Throughput/Area (bits /(cycle × slices))
	cyc/rnd	cyc/byte			
Vector Extensions [15]	66	-	1010.1×10^{-3}	(only simulation)	
64-bit with LMUL =1 (EleNum=5, SN=1)	103	12.8	624.02×10^{-3}	7323	85.21×10^{-6}
64-bit with LMUL =1 (EleNum=15, SN=3)	103	12.8	1872.07×10^{-3}	24785	75.52×10^{-6}
64-bit with LMUL= 1 (EleNum=30, SN=6)	103	12.8	3744.15×10^{-3}	48180	77.71×10^{-6}
64-bit with LMUL =8 (EleNum=5, SN=1)	**75**	**9.5**	$\mathbf{845.67}\times10^{-3}$	**7323**	$\mathbf{115.48}\times10^{-6}$
64-bit with LMUL =8 (EleNum=15, SN=3)	75	9.5	2537.00×10^{-3}	24789	102.34×10^{-6}
64-bit with LMUL=8 (EleNum=30, SN=6)	**75**	**9.5**	$\mathbf{5073.00}\times10^{-3}$	**48180**	$\mathbf{105.29}\times10^{-6}$

TABLE III: Results of our 32-bit architectures and comparison with 32-bit reference architectures.

Implementation	Execution time		Throughput (bits /cycle)	Area (slices)	Throughput/Area (bits /(cycle × slices))
	cyc/rnd	cyc/byte			
LEON3 [13]	-	369	21.68×10^{-3}	8648	2.51×10^{-6}
MIPS Native [14]	-	178.1	44.92×10^{-3}	6595	6.81×10^{-6}
MIPS Coprocessor [14]	-	137.9	58.01×10^{-3}	7643	7.59×10^{-6}
OASIP [12]	-	291.5	27.44×10^{-3}	981	27.97×10^{-6}
DASIP [12]	-	130.4	61.36×10^{-3}	1522	40.31×10^{-6}
32-bit with LMUL=8 (EleNum=5, SN=1)	**147**	**18.1**	$\mathbf{441.98}\times10^{-3}$	**6359**	$\mathbf{69.5}\times10^{-6}$
32-bit LMUL=8 (EleNum=15, SN=3)	147	18.1	1325.97×10^{-3}	23408	56.65×10^{-6}
32-bit LMUL=8 (EleNum=30, SN=6)	**147**	**18.1**	$\mathbf{2651.93}\times10^{-3}$	**48036**	$\mathbf{55.2}\times10^{-6}$

使用三种不同结构的向量扩展编译所有优化的程序：

a. LMUL等于1的64位体系结构；
b. LMUL等于8的64位体系结构；
c. LMUL等于8的32位体系结构。

每个生成的二进制机器代码都存储在SIMD处理器的程序存储器中。前两种结构使用相同的系统验证器代码，因为指令可以支持不同的LMUL设置。当我们增加EleNum值时，向量寄存器文件可以保持多个Keccak状态，并且该体系结构可以并行地执行多个Keccak操作。Keccak状态数（SN）决定了并行处理的状态数。无论系统中同时存在多少个Keccak状态，延迟都是相同的。

05

论文总结

Summary of papers

- 研究背景
- 关键技术
- 论文总结
- 理论支撑
- 实验分析

论文总结

论文成果总结

本文探讨了使用自定义向量指令集扩展来实现SHA-3哈希函数中的Keccak-f[1600]排列。

- 分析了五步映射，为64位和32位架构提出了10个自定义向量扩展，并在系统verilog中的SIMD处理器中实现了这些自定义指令。
- 使用自定义向量指令和现有的RISC-V向量扩展，为64位和32位架构设计了Keccak-f[1600]排列。我们对32位的结果与现有的两种并行设计[12]，[14]相比，该体系结构的吞吐量分别提高了45.7倍和43.2倍。与支持[15]进行向量扩展的现有设计相比，64位体系结构提供了5.3倍的吞吐量优化。

未来展望

- 应用于实际：把这项工作整合到PQC算法的实现中，如Kyber和Dilithium，通过Keccak-f[1600]排列的向量化来提高性能。
- 拓展研究：将研究优化完整的后量子密码学方案与其他技术，如多项式乘法优化。

非常感谢您的阅览

Thank you very much for your reading.