# Maximizing the Potential of Custom RISC-V Vector Extensions for Speeding up SHA-3 Hash Functions

Huimin Li[1], Nele Mentens[2,3], and Stjepan Picek[4,1]

[1] Delft University of Technology, The Netherlands
[2] Leiden University, The Netherlands
[3] KU Leuven, Belgium
[4] Radboud University, The Netherlands

**Abstract.** SHA-3 is considered to be one of the most secure standardized hash functions. It relies on the Keccak-f[1 600] permutation, which operates on an internal state of 1 600 bits, mostly represented as a $5 \times 5 \times 64-bit$ matrix. While existing implementations process the state sequentially in chunks of typically 32 or 64 bits, the Keccak-f[1 600] permutation can benefit a lot from speedup through parallelization. This paper is the first to explore the full potential of parallelization of Keccak-f[1 600] in RISC-V based processors through custom vector extensions on 32-bit and 64-bit architectures. We analyze the Keccak-f[1 600] permutation, composed of five different step mappings, and propose ten custom vector instructions to speed up the computation. We realize these extensions in a SIMD processor described in SystemVerilog. We compare the performance of our designs to existing architectures based on vectorized application-specific instruction set processors (ASIP). We show that our designs outperform all related work thanks to our carefully selected custom vector instructions.

**Keywords:** Keccak, SHA-3, Vector Extensions, SIMD Processor, RISC-V

## 1 Introduction

Data integrity is a crucial metric to guarantee the accuracy and reliability of transmitted information [6]. The Secure Hash Algorithm (SHA), a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST), has a wide range of applications in the domain of data integrity verification [3]. SHA-3, the newest generation, is used in a number of candidate algorithms in the NIST Post Quantum Cryptography (PQC) contest [8]. Especially in lattice-based schemes, SHA-3 functions are used to calculate hashes and generate random numbers on a large scale. The Keccak permutation in SHA-3 is computationally intensive due to its high number of rounds and a high number of state bits. It is always one of the speed-critical components in lattice-based algorithms [1,**?**,**?**]. In CRYSTALS-Kyber, the same seeds

are usually adopted as input data to generate the polynomial matrix A, the secret key vectors $\mathbf{s}$, and the error data vectors $\mathbf{e}$ using SHA-3 functions. Take the matrix A generation in Kyber1024, for example [1]. The public $4 \times 4$ matrix $\mathbf{A}$ is generated from a two-layer loop structure by SHAKE-128, an extendable output function in SHA-3, whose input data is the seed concatenated with the row order and the column order. Because of the large amount of computation and similar input data, it would be beneficial if one or more Keccak states could work simultaneously to generate A, $\mathbf{s}$, and $\mathbf{e}$. This work explores the feasibility of using vector instructions to make one or more Keccak states work in parallel. To realize this goal, we need a vector instruction set architecture (ISA) supporting a flexible vector length that is large enough to include one or more Keccak states. RISC-V vector extensions meet this requirement. To the best of our knowledge, there are no other papers that use RISC-V vector extensions for speeding up SHA-3.

To investigate how RISC-V vector extensions can improve the performance of SHA-3, we use the same scalable SIMD RISC-V based processor as in [7] to do ASIP designs. We allow different numbers of elements in one vector register to process one or more Keccak states simultaneously. We analyze the algorithm consisting of five different step mappings in the Keccak permutation, propose ten custom vector extensions for 32-bit and 64-bit architectures, and realize all these custom extensions in the SIMD processor described in SystemVerilog. Then, we design the Keccak permutation targeting the 32-bit and 64-bit architectures using our custom vector extensions and existing vector extensions for RISC-V. Our contributions include the following aspects:

- We use RISC-V vector extensions to vectorize the Keccak-f[1 600] permutation of the SHA-3 function. To the best of our knowledge, we are the first to use these extensions to speed up SHA-3.
- We analyze the five step mappings in the Keccak permutation, propose ten custom vector extensions for 32-bit and 64-bit architectures and realize all these extensions in a SIMD processor written in SystemVerilog.
- We optimize the Keccak program for the 32-bit and 64-bit architectures using the custom and existing RISC-V vector extensions. The results show that our ASIP designs significantly outperform all previously proposed implementations.

## 2 Background

All SHA-3 functions use the Keccak-f[1 600] permutation, which works on a 1 600-bit state, which is ordered as a three-dimensional $x \times y \times z$ matrix. where $x$ and $y$ are 5, and $z$ is 64. Therefore, the $5 \times 5 \times 64 - bit$ state can be viewed as 25 lanes, with each lane consisting of 64 bits. They can be partitioned plane-wise as 5 planes, with each plane containing 5 lanes in the same row. Plane-wise partition is preferable to work with vector instructions, where lanes within the same row can be processed simultaneously by the same instructions [2]. We follow this processing approach in this work.

The Keccak-f[1 600] permutation comprises 24 rounds. Each round contains five step mappings, denoted as $\theta$, $\rho$, $\pi$, $\chi$, $\iota$. The detailed operations for plane-per-plane processing are shown in Algorithm 1. The $\theta$ step mapping, designed for linear diffusion, changes the lane value through XORing each state bit with parities of adjacent columns. The $\rho$ step mapping, designed for inter-slice dispersion, rotates each lane over a variable number of positions according to its location. The $\pi$ step mapping, designed for disturbing horizontal/vertical alignment, scrambles the location of all lanes. The $\chi$ step mapping, designed for non-linearity, updates the value of each row with AND, NOT, and XOR operations among different lanes. The $\iota$ step mapping, designed for breaking the symmetry, XORs a round constant with lane 0. The round constant (RC) value changes according to the round number.

### 2.1 RISC-V Vector ISA

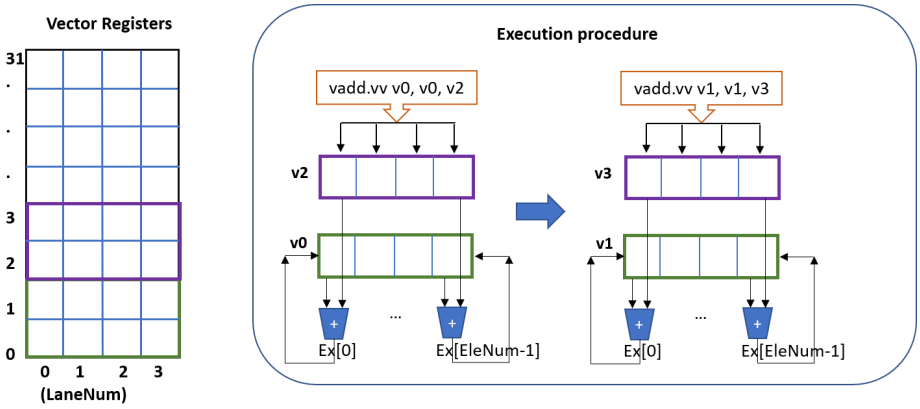


Fig. 1: Vector register file and address allocation [7].

RISC-V is an open and freely accessible ISA with small base instructions (ISA bases) for simplified general-purpose computers and rich optional instruction extensions for more comprehensive applications. RISC-V vector extensions (RISC-V vector ISA) are designed for vector operations. It includes the following main features according to the most recent version 1.0 (RVV1.0) [11]:

1. There are 32 vector registers in total. The vector length, VLEN, defines the number of bits in a single vector register. The element length, ELEN, defines the number of bits in every vector element that any operation can produce or consume. The number of elements, EleNum, defines the number of vector elements in one vector register. EleNum is determined by VLEN/ELEN. The vector length, VL, specifies the number of elements to be operated on in parallel within a vector extension [11]. It can be smaller or greater than

**Algorithm 1** Keccak-f[1 600] step mappings in plane-per-plane processing [4]
**Input:** Keccak state $\mathbf{A}[x, y]$
**Output:** Keccak state $\mathbf{H}[x, y]$
**Note:**
1. $\mathbf{B}, \mathbf{C}, \mathbf{D}, \mathbf{E}, \mathbf{F}, \mathbf{G}$ are all intermediate values.
2. The pairs [x, y] define the lane(x,y), with $0 \leq x<5$ and $0 \leq y<5$.
3. r[x, y] is the rotation value for each lane in the $\rho$ step mapping.
4. RC[i] is the round constant value in the $\iota$ step mapping.

1) $\theta$ 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
2) $\rho$ 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
3) $\pi$ 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
4) $\chi$ 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) $\iota$ step mapping:
$\mathbf{H}[0, 0] = \mathbf{H}[0, 0] \oplus \mathrm{RC}[i]$

EleNum. When VL is smaller than EleNum, all elements are put in the same vector register. When VL is greater than EleNum, several vector registers are grouped to work under the same instruction. The vector length multiplier, LMUL, specifies the maximum number of vector registers grouped under the same instruction. LMUL supports integer values no larger than 8, that is, 1,2,4, or 8.

2. There are three types of instructions: configuration-setting instructions, vector load, store instructions, and vector arithmetic instructions. The configuration-setting instructions define VL, LMUL, ELEN, etc. The vector load and store instructions define how to move values between vector registers and data memory. Vector arithmetic instructions define the operands and the opcode. Their funct3 field specifies whether the two operands are vector-vector (.vv), vector-immediate (.vi), or vector-scalar (.vx).
3. Masking is supported on many vector instructions and can be applied to the specific locations of vector elements in the vector register. The *vm* field in the vector load and store instructions and vector arithmetic instructions denotes whether the corresponding instructions are masked off or not. When *vm* equals 1, the instruction is unmasked. Every element in the operand vectors will participate in the corresponding operation. When *vm* equals 0, the instruction is masked. The corresponding operation only happens to these elements whose mask bit is 1 in the mask vector register, which resides in the vector register file.
4. The SIMD processor needs to do vector address remapping according to LMUL. Figure 1 shows the working procedure for the instruction {*vadd.vv v0,v0,v2*}. The elements in the first vector register of vectors *v0* and *v2* are read out simultaneously and sent to the respective execution module with the same element index number for the addition operation. After the process finishes, the result of every execution sub-module will be sent to vector *v0* according to the element index number. Later, all elements from *v1* and *v3* will be fetched and executed, and the result from every execution sub-module will be written back to vector *v1*.

### 2.2 Related Work

Instruction Set Extension (ISE) is commonly used in ASIP designs to extend the ISA with customized extensions for specific functions. These custom instructions are usually suited to fine-grained operations that are best integrated into a processor pipeline and still provide software programmability while only needing small hardware changes to processors [12,**?**].

As far as we know, there are three implementations using ISE for SHA-3 implemented in FPGA or ASIC. All are application-specific instruction set processors (ASIP), whose instruction set is tailored to meet the requirements of a specific application. In 2015, Wang et al. [13] were the first to propose custom extensions for SHA-3 implemented in FPGA. In 2016, Elmohr et al. [5] proposed two ASIPs based on a 32-bit processor for SHA-3. The first one (Native ISE) uses four custom instructions, and the second one (Co-processor ISE) adds auxiliary

registers to supply parallel implementations. In the domain of the RISC-V, Rao et al. in 2018 [9] proposed two SHA-3 ASIPs for IoT system. The first ASIP, named OASIP, accelerates operations on the existing datapath with seven instruction extensions. The second ASIP, named DASIP, supports 21 instruction extensions and make data and instructions work in parallel.

In the field of vector instructions, Rawat et al. [10] proposed six vector instruction extensions for 128-bit vector-processing units in some mainstream processors such as ARM (NEON), Intel (SSE, AVX), etc. They designed the assembly code program for Keccak-f[1 600] for a 64-bit architecture and integrated these vector instructions for simulation. As the authors mentioned in the paper, they achieved 66 instructions per keccak-f[1 600] round. Until now, no other published works have used RISC-V vector extensions to design the Keccak functions[5]. In this work, we will use the four designs mentioned above as our reference for performance comparison in Section 4.

## 3 System Design

The authors in [7] realized a scalable SIMD processor that can support RISC-V ISA bases and RISC-V vector extensions written in SystemVerilog. We will use the same SIMD processor to investigate the performance improvement of SHA-3 with the goals of low latency and high throughput. The SIMD processor in [7] contains a scalar core and a vector processing unit. Both parts are 32-bit architectures. However, as the configuration-setting instructions can set the ELEN parameter to different values, the data width in the vector processing unit does not have to be consistent with the scalar core. Following the description from Reference [11], it can be any length that is a power of 2 and no smaller than 8. This mismatch does not impact the load and store operations because the vector load and store instructions can also define the width of the data read from the data memory. We will set the element length (ELEN) to 64 bits and 32 bits separately to realize the 64-bit architecture and the 32-bit architecture, respectively. To show the entire vectorization process for the Keccak permutation, we do not combine operations like many software designs do, for example, by combining the $\rho$ and $\pi$ step mappings [2].

### 3.1 64-bit Architecture

For the 64-bit architecture, we set ELEN to 64 bits for making the SIMD processor's vector processor unit deal with 64-bit operands. Keccak-f[1 600] is easy to map to the 64-bit architecture as its lane width in the Keccak state is compatible with the element length in the vector register.

We set the vector length VLEN, determined by VLEN/ELEN, to fit the $5 \times 5$ lanes inside the vector register file, with 5 planes occupying 5 vector registers.

[5] After we finished this work, the RISC-V Cryptography Extensions Task Group published Vector Crypto Draft 20220920 on 20 September 2022. Until now, there are no vector extensions for Keccak in the draft.

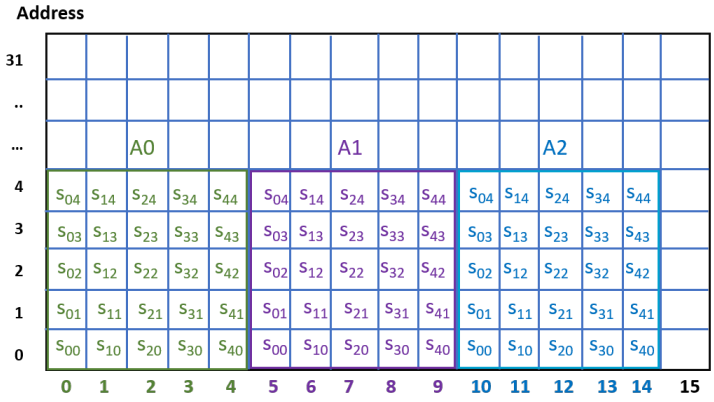


Fig. 2: Memory allocation for Keccak states in the 64-bit architecture.

Moreover, as illustrated in Figure 2, if VLEN is large enough, more than one Keccak state can be put in the vector register file. In this figure, the EleNum parameter is 16, and $s_{xy}$ denotes the lane index in one Keccak state with row index $x$ and column index $y$. The planes with the same order from different Keccak states reside in the same vector registers. We use the vector register address to denote the y-axis and the element index order modulo 5 to indicate the x-axis of one state. The first Keccak state, A0, occupies element index order 0 to 4, shown in green; the second Keccak state, A1, occupies element index order 5 to 9, shown in purple; and the third Keccak state, A2, occupies element index 10 to 14, shown in blue.

### 3.2 32-bit Architecture

For the 32-bit architecture, we set the ELEN parameter of the SIMD processor to 32 bits. Later, we need to consider cutting the 64-bit lane into two 32-bit lanes to reside inside the vector register file and work on 32-bit operands. The most common way is the bit interleaving technique, where the odd bits are put in one 32-bit word and the even bits in another 32-bit word. This technique is beneficial for the rotation operation, especially in the $\rho$ step mapping, where the rotation length is sometimes larger than 32. However, when SHA-3 algorithms work with other programs, extra efforts are required to separate the lane into odd and even parts and then combine them. In this design, we divide each lane into the most significant and least significant parts, with each part containing 32 bits. We store the two parts separately inside the vector register file, as shown in Figure 3.

### 3.3 Custom Vector Extensions

As the existing RISC-V vector instructions are for general-purpose applications, specific instructions for implementing Keccak in the 64-bit and 32-bit archi-

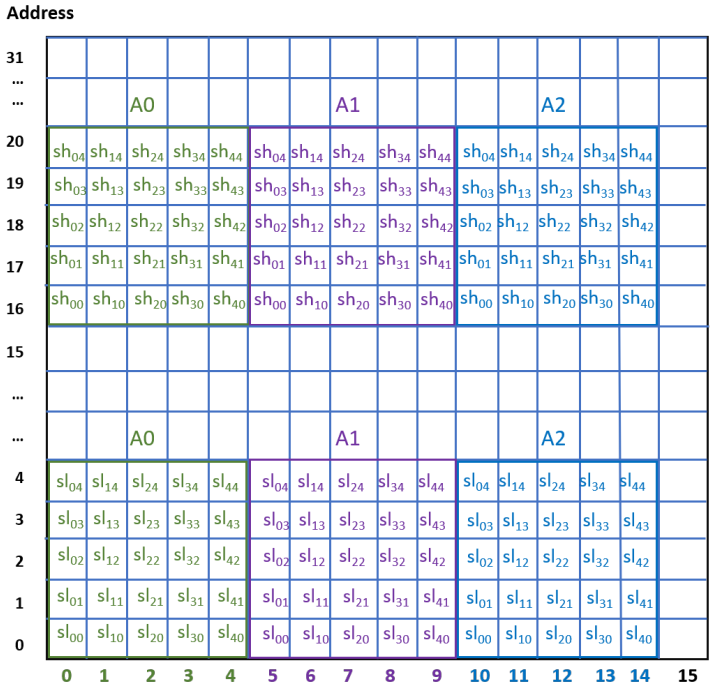


Fig. 3: Memory allocation for Keccak states in the 32-bit architecture

tectures are needed. For example, there are no vector rotation instructions in RISC-V vector ISA, and vector slide instructions define behaviors that are not applicable to our use case, etc. In this part, we propose custom vector extensions for SHA-3 and realize them through SystemVerilog in the SIMD processor. We define the parameter $SN$ to denote the number of Keccak states working in parallel. $5 \times SN$ should not be greater than the number of elements in one vector register. Note that all the following instructions only operate on elements that store the Keccak state values (element index number $\in [0, 5 \times SN - 1]$). Elements with index numbers not smaller than $5 \times SN$ are unchanged. In the following parts, *vd* denotes the destination vector operand. *v1* and *v2* denote the source vector operands. *uimm* defines the unsigned immediate. *simm* specifies the signed immediate. *rs1* specifies the scalar register operand. *vm* denotes whether vector masking is enabled. In this design, we use custom instructions and rewrite unused existing instructions to extend instructions in RISC-V. We do not modify the compiler because it is too time-consuming and not flexible. All instructions and their latency are shown in Table 1.

**Vector slide modulo five instructions** In the $\theta$ step mapping, intermediate values move up and down the corresponding vector register after XORing all planes. Moreover, inside the $\chi$ step mapping, all planes must move down their

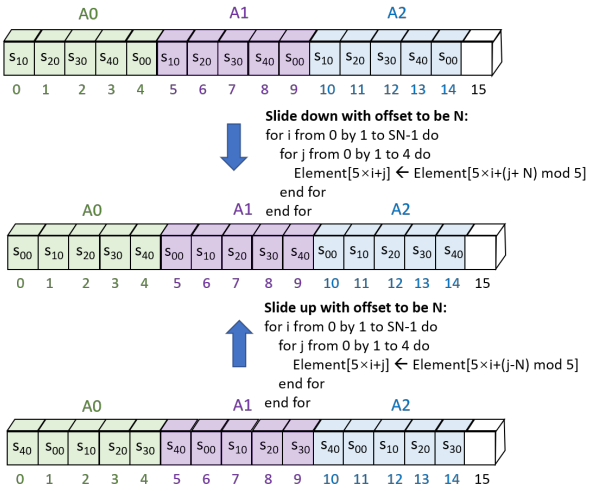


Fig. 4: Vector slide and modulo-five instructions. SN denotes the number of Keccak states. N is the offset. Here, we take the offset of 1 as an example.

corresponding vector registers with offsets one and two, respectively. We propose two extensions for both architectures: *vslidedownm* to do the moving down operation, and *vslideupm* to do the moving up operation. To keep lanes belonging to different Keccak states from interfering, we use modulo-five operations to restrict the element index number, as shown in Figure 4.

**Vector rotation instructions** There are two step mappings using rotation operations: $\theta$ and $\rho$. In the $\theta$ step mapping, the parity of the right column rotates one bit towards the most significant direction. For the 64-bit architecture, we propose the rotation operation *vrotup* with two vector operands and one immediate value, which defines the offset. For the 32-bit architecture, we need to concatenate two 32-bit words into one 64-bit word and then do the rotate operation. As there are two vector operands, we choose the default rotation offset of 1 and create two custom extensions: *v32lrotup* and *v32hrotup*. The results are the low 32 bits and the high 32 bits of the rotated 64-bit data, respectively.

The $\rho$ step rotates each lane over a variable number of positions. For the 64-bit architecture, we do not use the rotation operation *vrotup* here because it makes all lanes in one plane rotate with the same offset under the same immediate value. We store the rotation values in a lookup table and create *v64rho* for the 64-bit architecture, and *v32lrho* and *v32hrho* for the 32-bit architecture. For *v64rho*, the two operands are vector and immediate data. When the immediate is -1, all five planes in the Keccak are executed in sequence. The immediate -1 is used when LMUL is greater than 1. Here, we use a counter in the execution module of the SIMD processor, named *lmul_cnt* to denote the row number for reading the offset from the lookup table. When the immediate equals 0, 1, 2, 3, or

Table 1: 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<br>for $j$ from 0 by 1 to 4 do<br>$vd[5\times i+j] \leftarrow vs2[5\times i+$<br>$(j+uimm) \bmod 5]$<br>end for<br>end for | 1+* |
| *vslideupm.vi vd, vs2, uimm, vm* | for $i$ from 0 by 1 to $SN-1$ do<br>for $j$ from 0 by 1 to 4 do<br>$vd[5\times i+j] \leftarrow vs2[5\times i+$<br>$(j-uimm) \bmod 5]$<br>end for<br>end for | 1+* |
| *vrotup.vi vd, vs2, uimm, vm* | $vd \leftarrow (vs2 \ll uimm) \vee (vs2 \gg (64-uimm))$<br>Note: $\vee$ denotes a bit-wise OR operation. | 1+* |
| *v32lrotup.vi vd, vs2, vs1, vm* | $vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1)$<br>$\gg 63))[31:0]$<br>Note: $vs2 \parallel vs1$ is the concatenation of $vs2$ and $vs1$, to build 64-bit word. | 1+* |
| *v32hrotup.vi vd, vs2, vs1, vm* | $vd \leftarrow (((vs2 \parallel vs1) \ll 1) \vee ((vs2 \parallel vs1)$<br>$\gg 63))[63:32]$ | 1+* |
| *v64rho.vi vd, vs2, simm, vm* | for $i$ from 0 by 1 to $SN-1$ do<br>for $j$ from 0 by 1 to 4 do<br>$vd[5\times i+j] \leftarrow (vs2[5\times i+j] \ll$<br>$rho\_shift[simm][j]) \vee (vs2[5\times i+j] \gg (64-$<br>$rho\_shift[simm][j]))$<br>end for<br>end for<br>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) $vs2 \parallel vs1$;<br>2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table;<br>3) The same process as *v64rho* is executed, and the least significant 32 bits are stored. | 1+* |
| *v32hrho.vi vd, vs2, vs1, vm* | 1) $vs2 \parallel vs1$;<br>2) The counter *lmul_cnt* in hardware indexes the row number automatically for reading the lookup table;<br>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<br>1) Reading elements from *vs2* in the vector register file and re-arranging the elements into columns.<br>2) Storing each column in the vector register with the starting address of the column equals to *vd*.<br>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<br>for $j$ from 0 by 1 to 4 do<br>if($j \equiv 0$)<br>$vd[5\times i+j] \leftarrow vs2[5\times i+j] \oplus RC[rs1]$<br>else<br>$vd[5\times i+j] \leftarrow vs2[5\times i+j]$<br>end for<br>end for<br>Note: $RC$ are round constant data. | 1+* |

4, only one plane is operated with the row index defined by the immediate, and LMUL should equal 1. For *v32lrho* and *v32hrho*, we combine two 32-bit words into one 64-bit word and then do the rotate operation. As there are only two operands, i.e., two vectors, there is no value defining the row number. Thus, they also use the counter *lmul_cnt* to index the row number for reading the lookup table. The results of *v32lrho* and *v32hrho*, are the least-significant 32 bits and the most-significant 32 bits of the rotated 64-bit data, respectively.

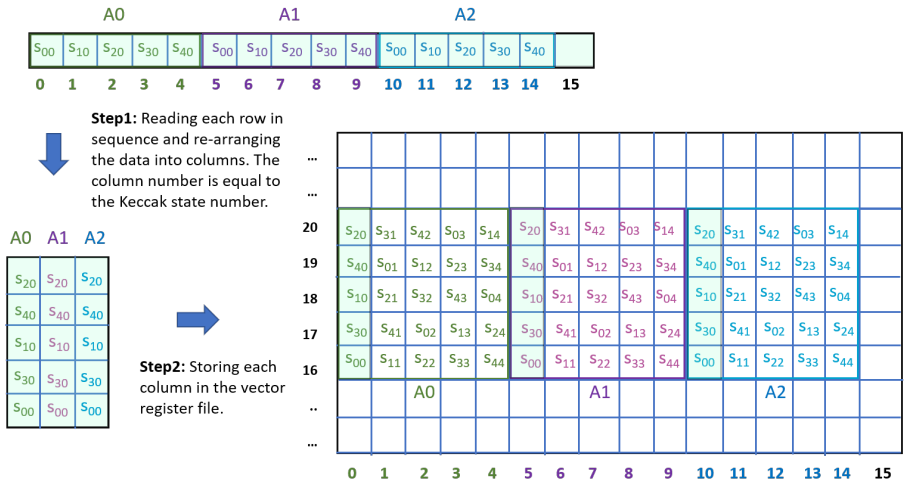


Fig. 5: $\pi$ operation in the design.

**Vector $\pi$ instruction** The $\pi$ step mapping contains two steps: 1) reading every row from the vector register file in sequence and re-arranging the elements into columns; 2) storing each column in the vector register. The column number is equal to the Keccak state number, $SN$. The operation is illustrated in Figure 5. We add interfaces between the execution module and the vector register file in the SIMD processor to make data writing in column mode available. We propose a new custom extension *vpi*. This instruction can work in both architectures. The two operands are vector and immediate data. When the immediate value is -1, all five planes in the Keccak are executed in sequence. This is used for LMUL greater than 1. When the immediate equals 0, 1, 2, 3, or 4, only one plane is processed, where the order is defined by the immediate, and LMUL should equal 1.

**Vector $\iota$ instruction** We propose the instruction *viota* to XOR a round constant with lane 0 in the first row of every Keccak state for the $\iota$ step mapping. The two operands in the instruction are a vector register and a scalar register.

The latter is used to index the round constant data. The data width of the round constant for the 64-bit architecture is 64 bits. For the 32-bit architecture, every round constant is divided into a high 32-bit value and a low 32-bit value, and the *viota* instruction runs twice for each Keccak round.

## 4 Implementations and Results

This design uses one RISC-V GNU Compiler Toolchain[6] to compile all our software programs. The Xilinx Alveo U250 Data Center accelerator card is selected as the hardware platform. We use Vivado 2020.1 tools to synthesize and implement the SIMD processor at 100 MHz. We compare our designs with the existing ASIP designs mentioned in Section 2.2, which adopt tailored processors with a subset of instructions to meet design requirements. In our implementations, we also use a smaller set of instructions together with the custom extensions for Keccak. We keep all instructions in the scalar core of the SIMD processor, where the base RISC-V ISA and multiplication and division extensions are supported. The vector processing unit reserves configuration-setting instructions, vector load and store instructions, vector logical instructions in vector arithmetic, and all custom extensions for different architectures.

Table 2: 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** | **Area** | **Throughput/Area** |
|---|---|---|---|---|---|
| | **cyc/rnd** | **cyc/byte** | **(bits /cycle)** | **(slices)** | **(bits /(cycle $\times$ slices))** |
| Vector Extensions [10] | 66 | - | $1\,010.1\times10^{-3}$ | (only simulation) | |
| 64-bit with LMUL =1 (EleNum=5, 1 state) | 103 | 12.8 | $624.02\times10^{-3}$ | 7 323 | $85.21\times10^{-6}$ |
| 64-bit with LMUL =1 (EleNum=15, 3 states) | 103 | 12.8 | $1\,872.07\times10^{-3}$ | 24 785 | $75.52\times10^{-6}$ |
| 64-bit with LMUL= 1 (EleNum=30, 6 states) | 103 | 12.8 | $3\,744.15\times10^{-3}$ | 48 180 | $77.71\times10^{-6}$ |
| **64-bit with LMUL =8 (EleNum=5, 1 state)** | **75** | **9.5** | $\mathbf{845.67}\times10^{-3}$ | **7 323** | $\mathbf{115.48}\times10^{-6}$ |
| 64-bit with LMUL =8 (EleNum=15, 3 states) | 75 | 9.5 | $2\,537.00\times10^{-3}$ | 24 789 | $102.34\times10^{-6}$ |
| **64-bit with LMUL=8 (EleNum=30, 6 states)** | **75** | **9.5** | $\mathbf{5\,073.00}\times10^{-3}$ | **48 180** | $\mathbf{105.29}\times10^{-6}$ |

We compile all the optimized programs using vector extensions for three different structures: (1) 64-bit architecture with LMUL equal to 1, (2) 64-bit architecture with LMUL equal to 8, and (3) 32-bit architecture with LMUL equal

[6] `https://github.com/riscv-collab/riscv-gnu-toolchain/`

Table 3: Results of our 32-bit architectures and comparison with 32-bit reference architectures.

| **Implementation** | **Execution time** | | **Throughput** | **Area** | **Throughput/Area** |
|---|---|---|---|---|---|
| | **cyc/rnd** | **cyc/byte** | **(bits /cycle)** | **(slices)** | **(bits /(cycle × slices))** |
| LEON3 [13] | - | 369 | $21.68 \times 10^{-3}$ | 8 648 | $2.51 \times 10^{-6}$ |
| MIPS Native [5] | - | 178.1 | $44.92 \times 10^{-3}$ | 6 595 | $6.81 \times 10^{-6}$ |
| MIPS Coprocessor [5] | - | 137.9 | $58.01 \times 10^{-3}$ | 7 643 | $7.59 \times 10^{-6}$ |
| OASIP [9] | - | 291.5 | $27.44 \times 10^{-3}$ | 981 | $27.97 \times 10^{-6}$ |
| DASIP [9] | - | 130.4 | $61.36 \times 10^{-3}$ | 1 522 | $40.31 \times 10^{-6}$ |
| **32-bit with LMUL=8 (EleNum=5, 1 state)** | **147** | **18.1** | $\mathbf{441.98} \times 10^{-3}$ | **6 359** | $\mathbf{69.5} \times 10^{-6}$ |
| 32-bit LMUL=8 (EleNum=15, 3 states) | 147 | 18.1 | $1\,325.97 \times 10^{-3}$ | 23 408 | $56.65 \times 10^{-6}$ |
| **32-bit LMUL=8 (EleNum=30, 6 states)** | **147** | **18.1** | $\mathbf{2\,651.93} \times 10^{-3}$ | **48 036** | $\mathbf{55.2} \times 10^{-6}$ |

to 8. Every generated binary machine code is stored inside the program memory of the SIMD processor. The former two structures use the same SystemVerilog code because the instructions can support different LMUL settings. As we increase the EleNum value, the vector register file can hold more than one Keccak state, and the architecture can perform multiple Keccak operations in parallel. The Keccak state number ($SN$) determines the number of states processed in parallel. The latency is the same no matter how many Keccak states there are in the system simultaneously.

We compare our results to the four reference designs introduced in Section 2.2. All results and comparisons are shown in Table 2 and Table 3. All references [13,5,9] use the number of slices as the unit to represent the resource utilization (area). In our work, we derive the number of slices from the post-implementation results in Vivado. We define two types of execution time: cycles per Keccak round (cycles/round) and cycles per message byte in one Keccak state (cycles/byte). Cycles/round is the latency to finish one Keccak round, while cycles/byte is the latency measured in clock cycles for hashing one byte of the message in the entire 24-round Keccak permutation. Either is justified to present the execution time. The reason to use the two is that different references use different measures. For example, reference [5] uses cycles/byte to denote the execution time; reference [9] adopts bytes/cycle to compare the performance. In addition, reference [10] uses cycles/round to represent its running time. Besides, we do not include the clock frequency to compare the performance because the reference designs either use different clock frequencies or do not mention their frequency.

**LMUL = 1 vs. LMUL = 8** In Table 2, we can see that in the 64-bit architecture, when LMUL is equal to 8, the performance improves. The throughput increases with a factor of 1.35 compared to when LMUL equals 1.

**64-bit architecture vs. 32-bit architecture** When comparing the 64-bit and 32-bit architectures with LMUL 8, we can see that the 64-bit architecture

runs almost twice as fast as the 32-bit architecture, and both use similar resources. The reason that the resources are similar is that the 32-bit architecture uses more resources for the rotation instructions, while the 64-bit architecture uses more resources for the datapath and the register file.

**32-bit architecture vs. MIPS Co-processor ISE [5]** When compared with the Co-processor ISE in [5], where parallel operations are supported, the throughput of our 32-bit architecture (LMUL = 8 and EleNum = 30) is improved by a factor of 45.7. The area is increased by a factor of 6.3.

**32-bit architecture vs. DASIP [9]** Our 32-bit architecture (LMUL = 8 and EleNum = 30) is 43.2 times faster but 31.5 larger than DASIP [9], which supports data-level and instruction-level parallelism.

**64-bit architecture vs. Vector Extensions [10]** For the 64-bit architecture (LMUL = 8 and EleNum = 30), the performance is increased by a factor of 5.3 compared to the vector extensions design for Keccak in [10] because more Keccak states can be processed simultaneously.

## 5 Conclusion and Future work

In this paper, we explore the use of custom vector instruction set extensions for the implementation of the Keccak-f[1 600] permutation in SHA-3 hash functions. We analyze the five step mappings, propose ten custom vector extensions for 64-bit and 32-bit architectures, and realize these custom instructions in the SIMD processor in SystemVerilog. Then, we design the Keccak-f[1 600] permutation for both the 64-bit and the 32-bit architectures using the custom vector instructions and the existing RISC-V vector extensions. Our results for the 32-bit architecture show an improvement of 45.7 and 43.2 times in throughput compared to two existing parallelized designs [5,9]. The 64-bit architecture offers optimization of 5.3 times compared to an existing design where vector extensions are supported [10]. Our work uses instruction-set customization and does not fuse adjacent operations for the purpose of showing the whole vectorization process using RISC-V vector extension. Predictably, the two architectures' performance will improve more if we increase the granularity or combine some adjacent operations.

In future work, we will integrate this work into the implementation of PQC algorithms, such as CRYSTALS-Kyber and CRYSTALS-Dilithium to see how the performance can be improved by the vectorization of Keccak-f[1 600] permutation. Moreover, we will investigate the optimization of the complete post-quantum cryptography schemes with other techniques, such as polynomial multiplication optimizations.

## References

1. Avanzi, R., Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: Crystals-kyber algorithm specifications and supporting documentation. NIST PQC Round **2**(4) (2017)

2. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V., Keer, R.V.: Keccak implementation overview. https://keccak.team/files/Keccak-implementation-3.2.pdf (2012)
3. Bider, D., Baushke, M.: Sha-2 data integrity verification for the secure shell (ssh) transport layer protocol. Request for Comments **6668** (2012)
4. Dworkin, M.J.: Sha-3 standard: Permutation-based hash and extendable-output functions (2015)
5. Elmohr, M.A., Saleh, M.A., Eissa, A.S., Ahmed, K.E., Farag, M.M.: Hardware implementation of a sha-3 application-specific instruction set processor. In: 2016 28th International Conference on Microelectronics (ICM). pp. 109–112. IEEE (2016)
6. Giani, A., Bent, R., Hinrichs, M., McQueen, M., Poolla, K.: Metrics for assessment of smart grid data integrity attacks. In: 2012 IEEE Power and Energy Society General Meeting. pp. 1–8. IEEE (2012)
7. Li, H., Mentens, N., Picek, S.: A scalable simd risc-v based processor with customized vector extensions for crystals-kyber. Cryptology ePrint Archive, Report 2021/1648 (2021), https://ia.cr/2021/1648
8. NIST: Pqc standardization process round4. https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4 (2022)
9. Rao, J., Ao, T., Xu, S., Dai, K., Zou, X.: Design exploration of sha-3 asip for iot on a 32-bit risc-v processor. IEICE TRANSACTIONS on Information and Systems **101**(11), 2698–2705 (2018)
10. Rawat, H., Schaumont, P.: Vector instruction set extensions for efficient computation of keccak. IEEE Transactions on Computers **66**(10), 1778–1789 (2017)
11. RISCVTeam: Risc-v vector specification. https://github.com/riscv/riscv-v-spec/releases/download/v1.0-rc1/riscv-v-spec-1.0-rc1.pdf (2021)
12. Sun, F., Ravi, S., Raghunathan, A., Jha, N.K.: A synthesis methodology for hybrid custom instruction and coprocessor generation for extensible processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **26**(11), 2035–2045 (2007)
13. Wang, Y., Shi, Y., Wang, C., Ha, Y.: Fpga-based sha-3 acceleration on a 32-bit processor via instruction set extension. In: 2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC). pp. 305–308. IEEE (2015)