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 software 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 hardware/software co-design to a software-only implementation on the one hand and to existing architectures based on (vectorized) hardware/software co-design on the other hand. We show that our design outperforms 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 [11]. 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 [5]. These applications include regular hashing, message authentication codes [14], digital signatures [9], pseudo-random number generators [7], key derivation algorithms [7], stream encryption [2], etc. The newest generation, SHA-3, provides better security than former functions. It will gradually replace the applications where SHA-1 and SHA-2 have been used [8,6].

SHA-3 functions are used in a number of candidate algorithms in the NIST Post Quantum Cryptography (PQC) contest [17]. 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 [1,23]. In CRYSTALS-Kyber, the same seeds are usually adopted as input data to generate the polynomial matrix A, the secret key vectors **s**, and the error data vectors **e** using SHA-3 functions. Take the matrix A generation in Kyber1024, for example [1]. The public 4×4 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, **s** and **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 [15] to do hardware/software (HW/SW) co-design. 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 assembly code program for the whole 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 for speeding 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.
- We use a register file with a flexible number of elements to accommodate one or more Keccak states, and we implement the software code in a Xilinx Alveo U250 accelerator card.
- The results show that our HW/SW co-design significantly outperforms all previously proposed software-only and HW/SW co-design implementations.

This paper is organized as follows. In Section 2, we describe the Keccak-f[1 600] permutation and the RISC-V vector Instruction Set Architecture (ISA) and present an overview of the most relevant related works. In Section 3, we demonstrate our methodology to design the 32-bit and 64-bit architecture and elaborate on the custom vector extensions for each step mapping in the two architectures. In Section 4, we illustrate how to use the RISC-V vector extensions and the custom extensions to realize the program for the two architectures. Then we summarize and compare the execution time, throughput, and resource utilization with the C-code implementation and four implementations in related works. In Section 5, we conclude this work and present our plans for future work.

2 Background

This section presents the necessary background information on the Keccak-f[1 600] permutation and the RISC-V vector ISA and gives an overview of four previously proposed implementations.

2.1 Keccak-f[1 600] permutation

All SHA-3 functions use the Keccak-f[1 600] permutation, which NIST selected as the SHA-3 Cryptographic Hash Algorithm Competition winner [4,9]. The Keccak permutation uses the sponge construction structure. As shown in Figure 1, the sponge function construction consists of three main phases: padding, absorbing, and squeezing, and two parameters: rate (r) and capacity (c). This construction can serve as a framework with arbitrary input and output length. That is, after an arbitrary number of input bits have been padded into a 1 600-bit message (padding phase), they are absorbed into the state of the Keccak function (absorbing phase), after which output with an arbitrary number of bits is squeezed out of the state (squeezing phase) [4,9].

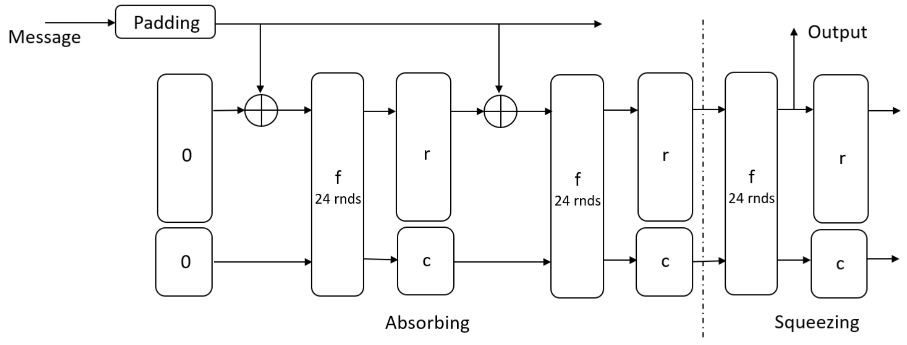

Fig. 1: The sponge construction [9].

The Keccak-f[1 600] permutation works on a 1 600-bit state, which is ordered as a three-dimensional $x \times y \times z$ matrix, as shown in Figure 2, 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 as 5 planes with each plane containing 5 lanes in the same row (plane-wise partition), 64 slices with each slice containing 25 bits (slice-wise partition), or 5 sheets with each sheet containing 5 lanes in the same column (sheet-wise partition). Among these different partition options, the plane-wise partition is preferable to work with vector instructions, where lanes within the same row can be processed simultaneously by the same instructions [3]. We follow this plane-per-plane processing approach in this work.

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

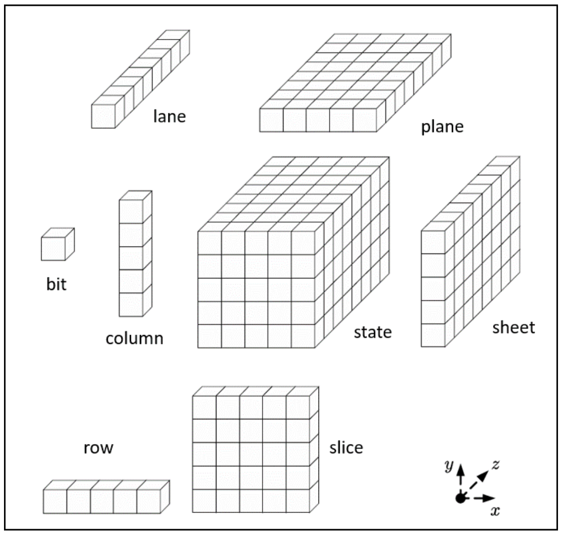

Fig. 2: Keccak state array.

2.2 RISC-V Vector ISA

RISC-V is an open and freely accessible ISA based on reduced instruction set computer (RISC) principles [25]. It is a real ISA suitable for direct native hard-

Algorithm 1 Keccak-f[1 600] step mappings in plane-per-plane processing [9]

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 ρ step mapping.
4. RC[i] is the round constant value in the ι 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:
$\mathbf{H}[0, 0] = \mathbf{H}[0, 0] \oplus \mathrm{RC}[i]$
```

ware implementation with small base instructions (ISA bases) for simplified general-purpose computers and rich optional instruction extensions for more comprehensive applications. These optional extensions are designed to work with all ISA bases without conflicts [25]. Besides, RISC-V allows users to customize their instructions by three different methods to accelerate specification applications [15].

RISC-V vector extensions (RISC-V vector ISA) are designed for vector operations. These extensions make multiple data execute the same highly-parallel process under one instruction and improve the whole system's performance. The RISC-V vector ISA includes the following main features according to the most recent version 1.0 (RVV1.0) [20]:

1. There are 32 vector registers in total.
2. The vector length, VLEN, defines the number of bits in a single vector register.
3. The element length, ELEN, defines the number of bits in every vector element that any operation can produce or consume.
4. The number of elements, EleNum, defines the number of vector elements in one vector register. EleNum is determined by VLEN/ELEN.
5. The vector length, VL, specifies the number of elements to be operated on in parallel within a vector extension [20]. It can be smaller or greater than 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.
6. 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.
7. There are three types of instructions: configuration-setting instructions, vector load and store instructions, and vector arithmetic instructions.
8. The configuration-setting instructions define VL, LMUL, ELEN, etc.
9. The vector load and store instructions define how to move values between vector registers and data memory. They support unit-stride, strided, and indexed addressing modes. There are fields in these instructions to define the width of memory elements, which can be different from the ELEN value described in the configuration-setting instructions.
10. 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). It also defines whether the corresponding operations are integer operations, multiply/division (MULT/DIV) operations, or fixed-point operations.
11. 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.

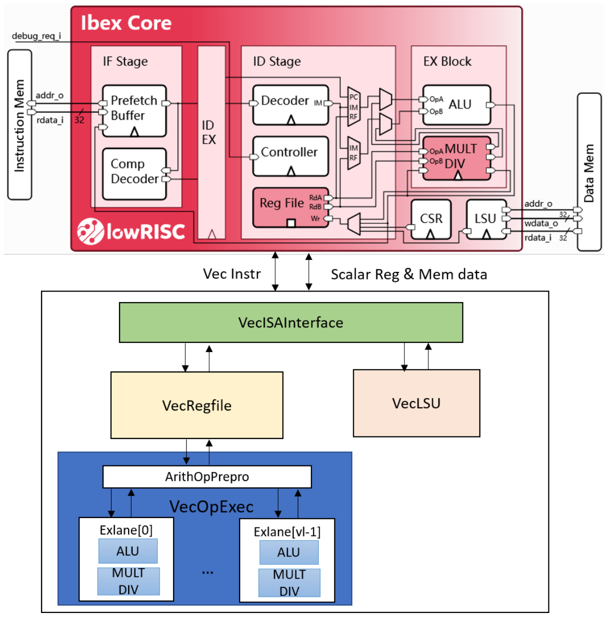

Fig. 3: The architecture of the SIMD RISC-V based Processor [15].

The authors in [15] realized a scalable SIMD processor that can support RISC-V ISA bases and RISC-V vector extensions written in SystemVerilog. The SIMD processor contains a scalar core (top) and a vector processing unit (bottom). The scalar core is an existing RISC-V core, Ibex [16], which decodes all RISC-V instructions and sends vector instructions to the vector processing unit. Inside the vector processing unit, the **VecISAInterface** module decouples the vector instructions into configuration-setting instructions, memory instructions, and vector arithmetic instructions, which are sent to **VecISAInterface** module, **VecLSU** module, and **VecOpExec** module, respectively. The **VecOpExec** module decode the vector arithmetic instructions further into different operations by **ArithOpPrepro** submodule. And the arithmetic instructions are sent to Execution Lane (**ExLane**) sub-modules.

Figure 4 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*.

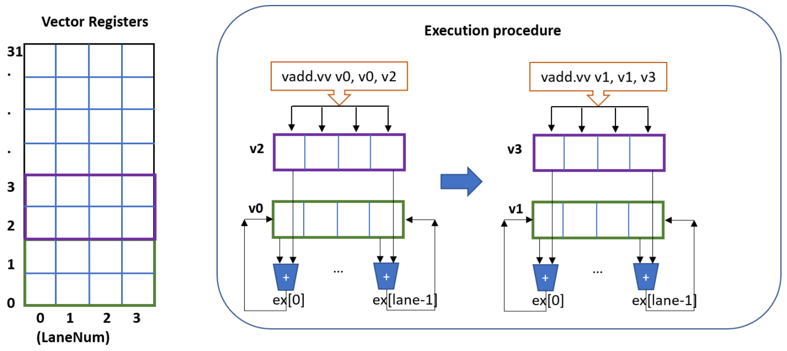

Fig. 4: Vector register file and address allocation [15].

2.3 Related Work

HW/SW co-design partitions the whole SoC (system-on-chip) system into hardware and software parts, with the hardware parts implemented on FPGA or ASIC and the software parts embedded in the memory of the FPGA or ASIC [15], to be executed on a processor. The advantage of going from a software-only design to HW/SW co-design is a trade-off between performance, flexibility, and resource utilization. The approach of using an Instruction Set Extension (ISE) is commonly used in HW/SW co-design 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 [22,18]. These instructions can be integrated into general-purpose processors and can also be integrated into application-specific instruction set processors (ASIP), whose instruction set is tailored to meet the requirements of a specific application.

As far as we know, there are three implementations using ISE for SHA-3 implemented in FPGA or ASIC. All are ASIPs. In 2015, Wang et al. [24] were the first to propose custom extensions for SHA-3 implemented in FPGA. In 2016, Elmohr et al. [10] 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 [18] 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, Rawa et al. [19] 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 reached a performance of 66 instructions per keccak-f[1 600] round. Until now, no other published works have used RISC-V vector extensions to design the SHA-3 functions. The RISC-V Cryptographic Extension Proposals [26] from University of Bristol design vector extensions for AES, SHA-2, etc, but not SHA-3 functions. And The authors design coarse-grained operations for per-round and all-round of AES and SHA-2, not fine-grained operations. In this work, we will design customized vector extensions for fine-grained operations in Keccak, and will use the four designs mentioned above [12,21,10,18] as our references for performance comparison in Section 4.2.

3 System Design

In this section, we will firstly give a preliminary introduction to our design methods for the 64-bit and 32-bit architectures in Section 3.1 and Section 3.2, respectively. Then we will elaborate on the custom vector extensions for each step mapping in the permutation for the two architectures in Section 3.3.

We will use the same SIMD processor in [15] to investigate the performance improvement of SHA-3 with the goals of low latency and high throughput. The SIMD processor 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 [20], 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. There is also nothing to consider for the vector arithmetic instructions when the two operands are vector-vector (.vv) and vector-immediate (.vi). We need to adjust the length of the scalar integer register after reading it from the scalar core if the two operands are vector-scalar (.vx). 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 ρ and π step mappings [3].

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

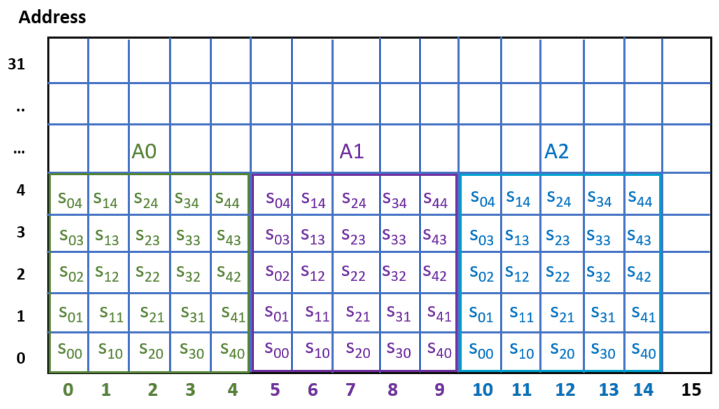

Fig. 5: Memory allocation for Keccak states in the 64-bit architecture.

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 to be large enough to make the EleNum parameter, determined by VLEN/ELEN, large enough for five lanes in one plane. We can fit the 5×5 lanes inside the vector register file, with 5 planes occupying 5 vector registers. Moreover, as illustrated in Figure 5, we can even put more than one Keccak state 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, and then also set the EleNum parameter large enough to accommodate one or more Keccak states. 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 ρ 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.

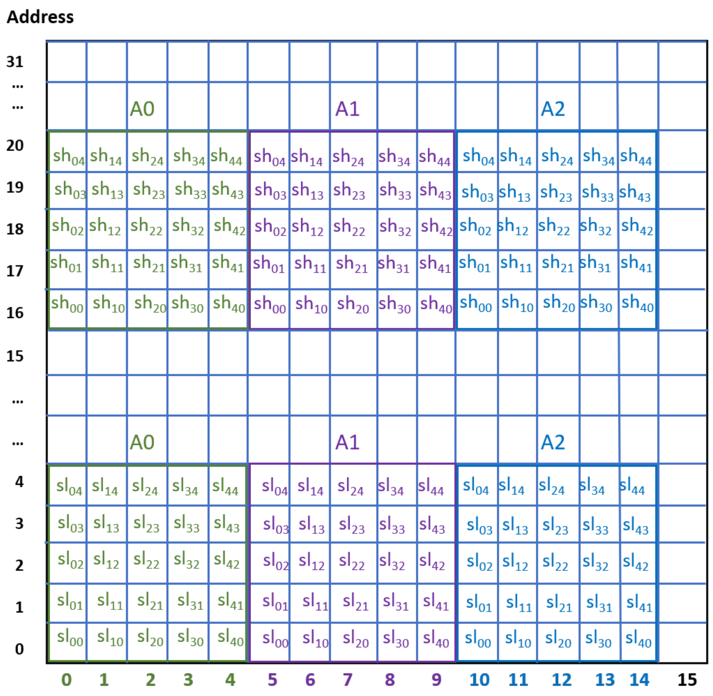

Fig. 6: Memory allocation for Keccak states in the 32-bit architecture

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 6. As a result, we do not need to partition each lane before and after the Keccak permutation because we can use the vector load and store instructions with indexed addressing modes to exchange data between the vector register file and data memory.

3.3 Custom Vector Extensions

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

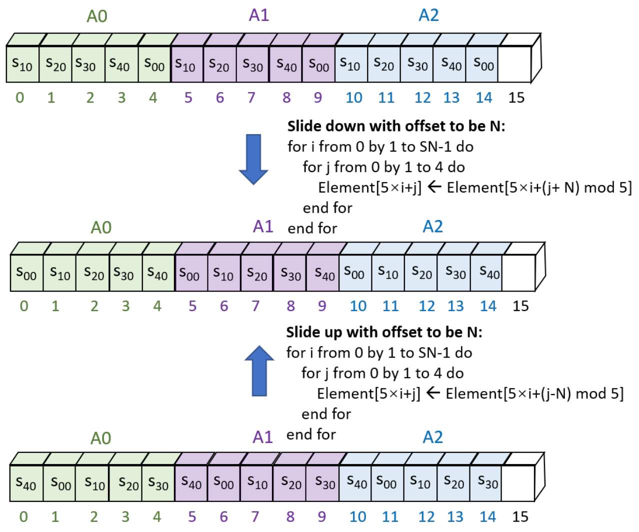
A0
A1
A2
Slide down with offset to be N:
for i from 0 by 1 to SN-1 do
for j from 0 by 1 to 4 do
Element[5×i+j] ← Element[5×i+(j+ N) mod 5]
end for
end for
Slide up with offset to be N:
for i from 0 by 1 to SN-1 do
for j from 0 by 1 to 4 do
Element[5×i+j] ← Element[5×i+(j-N) mod 5]
end for
end for

Fig. 7: 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.

Vector slide modulo five instructions In the θ step mapping, intermediate values move up and down the corresponding vector register after XORing all planes. Moreover, inside the χ step mapping, all planes must move down their 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 7. The two instructions are also explained in Table 1.

Vector rotation instructions There are two step mappings using rotation operations: θ and ρ. In the θ 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

Table 1: Vector slide modulo five instructions.

Instruction	**Description**	**64-bit**	**32-bit**
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	Yes	Yes
vslideupm.vi vd, vs2, uimm, vm	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	Yes	Yes

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.

Table 2: Lookup table for the ρ step mapping.

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

The ρ 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 (see Table 2) 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 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. All rotation instructions are illustrated in Table 3.

Vector π instruction The π 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

Table 3: Vector rotation instructions.

Instruction	**Description**	**64-bit**	**32-bit**
vrotup.vi vd, vs2, uimm, vm	$vd \leftarrow (vs2 \lll uimm) \vee (vs2 \ggg (64 - uimm))$ Note: $\vee$ denotes a bit-wise OR operation.	Yes	No
v32lrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \lll 1) \vee ((vs2 \parallel vs1) \ggg 63))[31:0]$ Note: $vs2 \parallel vs1$ is the concatenation of $vs2$ and $vs1$, to build 64-bit word.	No	Yes
v32hrotup.vi vd, vs2, vs1, vm	$vd \leftarrow (((vs2 \parallel vs1) \lll 1) \vee ((vs2 \parallel vs1) \ggg 63))[63:32]$	No	Yes

v64rho.vi vd, vs2, simm, vm	from 0 by 1 to $SN-1$ do
	  for j from 0 by 1 to 4 do
	    $vd[5 \times i + j] \leftarrow (vs2[5 \times i + j] \lll rho\_shift[simm][j])$
	      $\vee(vs2[5 \times i + j] \ggg (64 - rho\_shift[simm][j]))$
	  end for
	end for
	Note: if *simm* is -1, the five rows process in sequence. The counter *lmul_cnt* in hardware indexes the row.
		Yes	No

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.	No	Yes
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.	No	Yes

Figure 8 and Table 4. 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.

Table 4: Vector π instruction.

Instruction	**Description**	**64-bit**	**32-bit**
vpi.vi vd, vs2, simm, vm	The process is illustrated in Figure 8 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. The counter *lmul_cnt* in hardware indexes the row.	Yes	Yes

Vector ι instruction We propose the instruction *viota* to XOR a round constant with lane 0 in the first row of every Keccak state for the ι 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

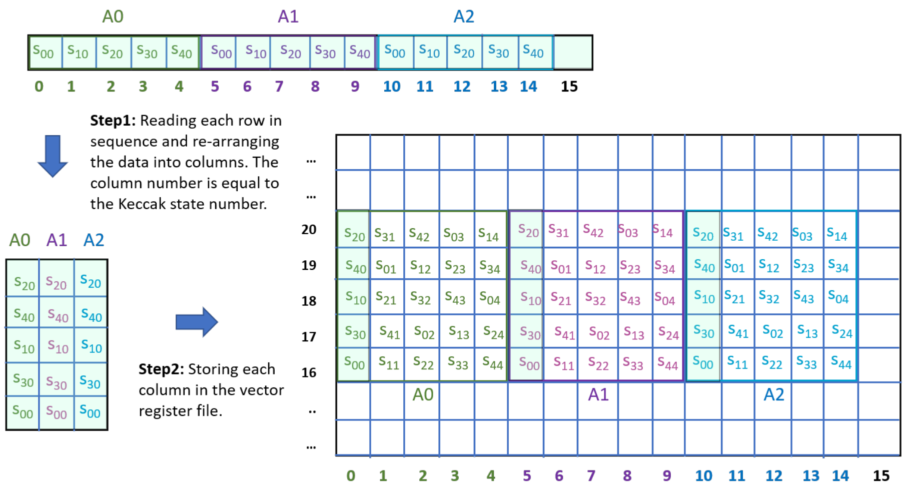
A0
A1
A2
Step1: Reading each row in sequence and re-arranging the data into columns. The column number is equal to the Keccak state number.
Step2: Storing each column in the vector register file.

Fig. 8: π operation in the design.

Table 5: Vector ι instruction.

Instruction	Description	64-bit	32-bit

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: The round constant values RC are shown in Table 6.
		Yes	Yes

constant for the 64-bit architecture is 64 bits, as shown in Table 5. 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

We will briefly illustrate how to use the RISC-V vector extensions and the custom extensions to realize the program for 64-bit and 32-bit architectures in Section 4.1. Then we will implement different architectures on a Xilinx Alveo U250 Data Center accelerator card, summarize and compare the execution time, throughput, and resource utilization with the C-code implementation and four reference works in Section 4.2.

4.1 Proposed Implementations Based on the SIMD Processor

After finishing the behavioral simulation to evaluate each instruction using the Vivado 2020.1 tools, we use a program written in assembly language to execute

Table 6: The round constant value in the ι step mapping.

RC[0]	0x0000000000000001	RC[1]	0x0000000000008082	RC[2]	0x800000000000808A
RC[3]	0x8000000080008000	RC[4]	0x000000000000808B	RC[5]	0x0000000080000001
RC[6]	0x8000000080008081	RC[7]	0x8000000000008009	RC[8]	0x000000000000008A
RC[9]	0x0000000000000088	RC[10]	0x0000000080008009	RC[11]	0x000000008000000A
RC[12]	0x000000008000808B	RC[13]	0x800000000000008B	RC[14]	0x8000000000008089
RC[15]	0x8000000000008003	RC[16]	0x8000000000008002	RC[17]	0x8000000000000080
RC[18]	0x000000000000800A	RC[19]	0x800000008000000A	RC[20]	0x8000000080008081
RC[21]	0x8000000000008080	RC[22]	0x0000000080000001	RC[23]	0x8000000080008008

Algorithm 2 Keccak-f[1 600] Permutation in the 64-bit architecture with LMUL equal to 1. Before the permutation, s1 is set to EleNum. s2 is set to -1 to make every bit 1 in order to perform a NOT operation through an XOR with 1. s3 is set to be 0, s4 is set to 24. The initial Keccak states are put in vector register 0, 1, 2, 3, 4. The program uses base instructions, RVV1.0 vector instructions [20] and custom vector instructions. cc means clock cycles

```
1: vsetvli x0,s1,e64,m1,tu,mu # 2 cc
2: permutation:
3:    #theta step
4:    vxor.vv v5,v3,v4 # 2 cc
5:    vxor.vv v6,v1,v2
6:    vxor.vv v7,v0,v6
7:    vxor.vv v5,v5,v7

8:    vslideupm.vi v6,v5,1 # 2 cc

9:    vslidedownm.vi v7,v5,1 # 2 cc
10:    vrotup.vi v7,v7,1 # 2 cc

11:    vxor.vv v5,v6,v7

12:    vxor.vv v0,v0,v5
13:    vxor.vv v1,v1,v5
14:    vxor.vv v2,v2,v5
15:    vxor.vv v3,v3,v5
16:    vxor.vv v4,v4,v5

17:    # rho step
18:    v64rho.vi v0,v0,0 # 2 cc
19:    v64rho.vi v1,v1,1
20:    v64rho.vi v2,v2,2
21:    v64rho.vi v3,v3,3
22:    v64rho.vi v4,v4,4

23:    # pi step
24:    vpi.vi v5,v0,0 # 3 cc
25:    vpi.vi v5,v1,1
26:    vpi.vi v5,v2,2
27:    vpi.vi v5,v3,3
28:    vpi.vi v5,v4,4
29:    # chi step
30:    vslidedownm.vi v10,v5,1 # 2 cc
31:    vslidedownm.vi v11,v6,1
32:    vslidedownm.vi v12,v7,1
33:    vslidedownm.vi v13,v8,1
34:    vslidedownm.vi v14,v9,1
35:    vxor.vx v10,v10,s2
36:    vxor.vx v11,v11,s2
37:    vxor.vx v12,v12,s2
38:    vxor.vx v13,v13,s2
39:    vxor.vx v14,v14,s2

40:    vslidedownm.vi v15,v5,2 # 2 cc
41:    vslidedownm.vi v16,v6,2
42:    vslidedownm.vi v17,v7,2
43:    vslidedownm.vi v18,v8,2
44:    vslidedownm.vi v19,v9,2
45:    vand.vv v10,v10,v15 # 2 cc
46:    vand.vv v11,v11,v16
47:    vand.vv v12,v12,v17
48:    vand.vv v13,v13,v18
49:    vand.vv v14,v14,v19

50:    vxor.vv v0,v5,v10
51:    vxor.vv v1,v6,v11
52:    vxor.vv v2,v7,v12
53:    vxor.vv v3,v8,v13
54:    vxor.vv v4,v9,v14

55:    # iota step
56:    viota.vx v0,v0,s3 # 2 cc

57:    # jump
58:    addi s3,s3,1
59:    blt s3,s4,permutation
```

the whole process. We first realize the program for the 64-bit architecture. To get a clear picture of the process, we first set LMUL to 1 such that one vector register is operated each time under one vector instruction, as shown in Algorithm 2.

Before the Keccak permutation, in line 1, the configuration-setting instruction *vsetvli* sets VL to EleNum through the scalar register s1, LMUL to 1 through m1, ELEN to 64 through e64. The θ step mapping is realized from line 4 to line 16. From line 4 to 7, the vector XOR instruction, *vxor*, computes the XOR of the five vector registers. Then, to get the parity of the adjacent columns, the intermediate outputs of line 7 slide down with offset 1 under instruction *vslidedownm* in line 8 for the first operands. Next, they slide up with offset 1 by instruction *vslideupm* in line 9 and rotate up by instruction *vrotup* in line 10 to get the second operand. The two operands are XORed by *vxor* to get the parity results in line 11. From lines 12 to 16, the parity results are XORed with all initial five vector registers, where the original Keccak states reside. From lines 18 to 22, five *v64rho* instructions are used to calculate the ρ step mapping, with the immediate value denoting the row number. From lines 24 to 28, the custom instruction *vpi* works to get the results from the π step. The destination operands of the five instructions are the same because the re-arranged elements are put in columns with a start addresses from v5, which is explained in Table 4 and Figure 8. The χ intermediate results are calculated from lines 30 to 54. Three types of instructions, *vslidedownm*, *vxor*, and *vadd* are applied here. As there is no NOT operation in RISC-V vector extensions, *vxor* is used to do NOT by XORing every bit with 1. *s2* is a scalar register assigned to be -1, with every bit equal to 1 because of complement storing. Instruction *viota* processes the ι step mapping in line 56. All operations work without loading or storing intermediate data to/from memory. This is very efficient and can save a significant portion of the execution time.

Algorithm 3 Modified implementation of ρ, π, χ and ι in the 64-bit architecture with LMUL = 8. Before the permutation, s5 is set to 5 $\times$ EleNum.

```
1:   # rho step
2:   vsetvli x0,s5,e64,m8,tu,mu # 2 cc
3:   v64rho.vi v0,v0,-1 # 6 cc

4:   # pi step
5:   vpi.vi v8,v0,-1 # 7 cc

6:   # chi step
7:   vslidedownm.vi v16,v8,1 # 6 cc
8:   vxor.vx v16,v16,s2 # 6 cc
9:   vslidedownm.vi v24,v8,2 # 6 cc
10:  vand.vv v16,v16,v24 # 6 cc
11:  vxor.vv v0,v8,v16 # 6 cc

12:  # iota step
13:  vsetvli x0,s1,e64,m1,tu,mu # 2 cc
14:  viota.vx v0,v0,s3 # 2 cc
```

Then, we set LMUL to be greater than 1 to consume less time for every permutation. According to RVV1.0 [20], when more than one vector register work together, LMUL should be an integer with the value of 1, 2, 4, or 8. Here, we choose LMUL to be equal to 8 to enable the processing of the five vectors, corresponding to the five rows of the Keccak state, under the same instruction.

Another way is choosing LMUL to be 4 and 1. This way, a group of 4 registers is operational, followed by a group of 1 register. We do not do this, because we would need to configure the LMUL value in an alternating way, which would consume more time. VL is set to be $5 \times$ EleNum. It is not better to make the θ and ι step mappings run with LMUL equal to 8 because in the θ step mapping, the five rows in sequence are XORed separately to get the column parities, and in the ι step mapping, only the first row is processed. LMUL equal to 8 is feasible for other step mappings, so we need to re-configure the LMUL value before the ρ step mapping and after the χ step mapping. We rewrite the ρ, π, χ, and ι step mappings accordingly in Algorithm 3.

For the 32-bit architecture, we also set LMUL to be 8 for ρ, π, and χ step mappings. As illustrated in Figure 6, we put the least-significant part in vector register 0 to 4 and the most significant part in vector register 16 to 20. The 32 vector registers in the SIMD processor are enough for the whole Keccak permutation. The program for the 32-bit architecture is similar to the 64-bit program, except for the rotation processes in the θ and ρ step mappings, where the specific vector rotation instructions for the 32-bit architecture, including *v32lrotup*, *v32hrotup*, *v32lrho* and *v32hrho*, are used.

4.2 Experimental Results

This design uses one RISC-V GNU Compiler Toolchain[5] 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 2.3, 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.

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 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. For the 64-bit architecture with LMUL equal to 1, the latency of one round (consisting of the 5 step mappings) is 103 cycles, and the latency of the Keccak permutation, i.e. 24 rounds plus a few additional instructions, is 2 564 cycles. For the 64-bit architecture with LMUL equal to 8, the latency of one round is 75 cycles, and the latency of the Keccak permutation is 1 892 cycles. Furthermore, for the

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

Table 7: 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 (cycles/round). The execution time to complete the entire permutation is reported as the number of cycles per byte (cycles/byte).

Implementation	**Execution time**		**Throughput**	**Area**
	cycles/round	**cycles/byte**	$(bits/cycle) \times 10^{-3}$	**(slices)**
Vector Extensions [19]	66	-	1 010.1	(only simulation)
64-bit with LMUL 1 (EleNum=5, 1 state)	103	12.8	624.02	7 323
64-bit with LMUL 1 (EleNum=15, 3 states)	103	12.8	1 872.07	24 789
64-bit with LMUL 1 (EleNum=30, 6 states)	103	12.8	3 744.15	48 180
64-bit with LMUL 8 (EleNum=5, 1 state)	75	9.5	845.67	7 323
64-bit with LMUL 8 (EleNum=15, 3 states)	75	9.5	2 537.00	24 789
64-bit with LMUL 8 (EleNum=30, 6 states)	**75**	**9.5**	**5 073.00**	**48 180**

Table 8: Results of our 32-bit architectures and comparison with 32-bit reference architectures. The execution time for one round is reported as the number of cycles it takes to complete one round (cycles/round). The execution time for the entire 24-round Keccak permutation is reported as the number of cycles it takes per byte to complete the permutation (cycles/byte).

Implementation	**Execution time**		**Throughput**	**Area**
	cycles/round	**cycles/byte**	$(bits/cycle) \times 10^{-3}$	**(slices)**
LEON3 ISE [24]	-	369	21.68	8 648
MIPS Native ISE [10]	-	178.1	44.92	6 595
MIPS Co-processor ISE [10]	-	137.9	58.01	7 643
OASIP [18]	-	291.5	27.44	981
DASIP [18]	-	130.4	61.35	1 522
Ibex core (C-code)	2908	355.69	22.45	432
32-bit with LMUL8 (EleNum=5, 1 state)	147	18.1	441.98	6 359
32-bit LMUL=8 (EleNum=15, 3 states)	147	18.1	1 325.97	23 408
32-bit LMUL=8 (EleNum=30, 6 states)	**147**	**18.1**	**2 651.93**	**48 036**

32-bit architecture, the latency of one round is 147 cycles, and the latency of the Keccak permutation is 3 620 cycles. 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. However, the throughput increases as EleNum increases. For example, when EleNum equals 30, 6 Keccak states can be processed in parallel. In this case, the number of bits processed per cycle is 3.74, 5.07, and 2.65, respectively, for the three architectures, compared to 0.62, 0.85, and 0.44 when only one state is processed in parallel.

To compare our architectures with other implementations, we first use the Keccak C-code from the PQ-M4 project as the baseline implementation [13]. We run the baseline code on the Ibex core [16], a pure 32-bit RISC-V based processor without RISC-V vector extensions, and determine the latency and resource utilization. Then we compare our results to the four reference designs introduced in Section 2.3. All results and comparisons are shown in Table 7 and Table 8.

All references [24,10,18] 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 [10] uses cycles/byte to denote the execution time; reference [18] adopts bytes/cycle to compare the performance. In addition, reference [19] 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 7, 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. C-code When comparing the C-code implementation with our 32-bit architecture (LMUL = 8 and EleNum = 30), we get a performance improvement of 117.9 times and utilize 111.2 times more FPGA resources.

32-bit architecture vs. MIPS Co-processor ISE [10] When compared with the Co-processor ISE in [10], 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 [18] Our 32-bit architecture (LMUL = 8 and EleNum = 30) is 43.2 times faster but 31.5 larger than DASIP [18], which supports data-level and instruction-level parallelism.

64-bit architecture vs. Vector Extensions [19] 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 [19] because more Keccak states can be processed simultaneously.

5 Conclusion and Future work

In this paper, we start from an existing SIMD RISC-V based processor to 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 vectorize the Keccak-f[1 600] permutation to make more than one Keccak state work simultaneously. 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 assembly code program 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 [10,18]. The 64-bit architecture offers optimization of 5.3 times compared to an existing design where vector extensions are supported [19]. Our work uses fine-grained 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 in the implementation of PQC algorithms, such as CRYSTALS-Kyber [1] and Saber [23] 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 CRYSTALS-Kyber and Saber 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. Bernstein, D.J.: Salsa20 specification. estream project algorithm description (2005)

3. 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)
4. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Sponge functions. In: ECRYPT hash workshop. vol. 2007. Citeseer (2007)
5. Bider, D., Baushke, M.: Sha-2 data integrity verification for the secure shell (ssh) transport layer protocol. Request for Comments **6668** (2012)
6. Chaves, R., Sousa, L., Sklavos, N., Fournaris, A.P., Kalogeridou, G., Kitsos, P., Sheikh, F.: Secure hashing: Sha-1, sha-2, and sha-3. Circuits and Systems for Security and Privacy; Taylor & Francis Group: Abingdon, UK pp. 105–132 (2016)
7. Cid, C.: Recent developments in cryptographic hash functions: Security implications and future directions. Information security technical report **11**(2), 100–107 (2006)
8. Debnath, S., Chattopadhyay, A., Dutta, S.: Brief review on journey of secured hash algorithms. In: 2017 4th International Conference on Opto-Electronics and Applied Optics (Optronix). pp. 1–5. IEEE (2017)
9. Dworkin, M.J.: Sha-3 standard: Permutation-based hash and extendable-output functions (2015)
10. 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)
11. 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)
12. Jain, M.K., Balakrishnan, M., Kumar, A.: Asip design methodologies: survey and issues. In: VLSI Design 2001. Fourteenth International Conference on VLSI Design. pp. 76–81. IEEE (2001)
13. Kannwischer, M.J., Rijneveld, J., Schwabe, P., Stoffelen, K.: pqm4: Testing and benchmarking nist pqc on arm cortex-m4. Cryptology ePrint Archive, Report 2019/844 (2019), https://ia.cr/2019/844
14. Krawczyk, H., Bellare, M., Canetti, R.: Hmac: Keyed-hashing for message authentication (1997)
15. 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
16. lowRISC: Ibex documentation. https://ibex-core.readthedocs.io/en/latest/01_overview/index.html (2021)
17. NIST: Post quantumr round3. https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions (2020)
18. 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)
19. Rawat, H., Schaumont, P.: Vector instruction set extensions for efficient computation of keccak. IEEE Transactions on Computers **66**(10), 1778–1789 (2017)
20. 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)
21. Schliebusch, O., Chattopadhyay, A., Kammler, D., Ascheid, G., Leupers, R., Meyr, H., Kogel, T.: A framework for automated and optimized asip implementation supporting multiple hardware description languages. In: Proceedings of the 2005 Asia and South Pacific Design Automation Conference. pp. 280–285 (2005)

22. 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)
23. Vercauteren, F., Sinha Roy, S., D'Anvers, J.P., Karmakar, A.: Saber: Mod-lwr based kem (round 3 submission) (2020)
24. 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)
25. Waterman, A., Lee, Y., Patterson, D.A., Asanović, K.: The risc-v instruction set manual, volume i: User-level isa, version 2.1 (2016)
26. Zeh, A., Glew, A., Spinney, B., Marshall, B., Page, D., Atkins, D., Dockser, K., Saarinen, M.J.O., Menhorn, N., Newell, R., et al.: Risc-v cryptographic extension proposals volume i: Scalar & entropy source instructions (2021)