Vector Instruction Set Extensions for Efficient Computation of KECCAK

Hemendra Rawat, *Student Member, IEEE,* and Patrick Schaumont, *Senior Member, IEEE*

Abstract—We investigate the design of a new instruction set for the KECCAK permutation, a cryptographic kernel for hashing, authenticated encryption, keystream generation and random-number generation. KECCAK is the basis of the SHA-3 standard and the newly proposed KEYAK and KETJE authenticated ciphers. We develop the instruction extensions for a 128-bit interface, commonly available in the vector-processing unit of many modern processors. We examine the trade-off between flexibility and efficiency, and we propose a set of six custom instructions to support a broad range of KECCAK-based cryptographic applications. We motivate our custom-instruction selections using a design space exploration that considers various methods of partitioning the state and the operations of the KECCAK permutation, and we demonstrate an efficient implementation of this permutation with the proposed instructions. To evaluate their performance, we integrate a simulation model of the proposed ARM NEON vector instructions into the GEM5 micro-architecture simulator. With this simulation model, we evaluate the performance improvement for several cryptographic operations that use the KECCAK permutation. Compared to a state-of-the-art NEON software implementation, we demonstrate a performance improvement of 2.2x for SHA-3. Compared to optimized 32-bit assembly programming, we demonstrate a performance improvement of 2.6x, 1.6x, and 1.4x for RIVER KEYAK, KETJESR and KETJEJR respectively. The proposed instructions require 4,658 gate-equivalent (GE) in 90 nm, which represents only a tiny fraction of the hardware cost of a modern processor.

Index Terms—KECCAK, SHA-3, instruction set, hardware/software codesign

1 INTRODUCTION

KECCAK is at the heart of the new hashing standard, adopted in 2013 by NIST as SHA-3. KECCAK is a generic cryptographic kernel which can also be used for pseudo-random number generation, keystream generation, and authenticated encryption [1], [2], [3]. In this contribution, we present the design of a vector instruction set extension for the KECCAK sponge and duplex construction. A vector instruction set extension is a mechanism to tightly integrate a custom-hardware module into the micro-architecture of a processor as a new instruction. These new instruction extensions are aimed at improving the performance of KECCAK computation over the baseline instruction set. We demonstrate the application of the proposed instruction extensions in the SHA-3 hashing algorithm, for the KEYAK and KETJE authenticated-encryption algorithms, and for pseudo-random number stream generation.

There are two reasons why software written with instruction extensions is faster. The first reason is that new instructions can be tuned to the specific computational needs of an application, such as frequently-occurring operations and operation patterns. The second reason is that new instructions can exploit the full parallelism of dedicated hardware. To a software developer who uses the instruction set extensions to develop the application, the performance gain appears as a reduction in the instruction count required to complete the application. Of course, the increased hardware parallelism has also energy benefits, because more work gets done more efficiently [4]. In an embedded and mobile environment, it may improve energy-efficiency and hence battery-life.

In this contribution, we propose vector instruction extensions to compute KECCAK. The proposed vector instructions balance flexibility against performance. The flexibility is required because the KECCAK kernel supports several state sizes (of 1,600-bit, 800-bit, and smaller), and because we wish to implement multiple cryptographic algorithms, such as hash and authenticated encryption, with KECCAK instructions. Hence, the challenge is to create instruction extensions that are faster than a classic software design of KECCAK, but that are also sufficiently generic to support a broad range of applications. In contrast, current vector instruction extensions for cryptography (such as AESNI and SHA-1/SHA-2 for Intel Processors [5], [6]) are optimized towards a single task.

The vector instruction extensions are designed for a 128-bit vector processing unit, which is commonly found in mainstream processor architectures including ARM (NEON) and Intel (SSE, AVX). Vector processing units support wide operands of 128-bit or more. They support a vector processing paradigm by partitioning the wide word into sub-word vectors (of 8, 16, 32, 64 bit). For cryptographic processing, the wide operand size is attractive because crypto-algorithms have a large state size. In recent years, several such crypto-instruction sets for vector processing unitss have been proposed, such as for the Advanced Encryption Standard, the SHA-1 and SHA-2 hash algorithms, and for carry-less multiplication [7].

• *The authors are with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, VA 24061. E-mail: {hrawat, schaum}@vt.edu.*

Manuscript received 27 Sept. 2016; revised 14 Apr. 2017; accepted 19 Apr. 2017. Date of publication 2 May 2017; date of current version 14 Sept. 2017. Recommended for acceptance by C.K. Koc.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2017.2700795

The contributions of this paper are as follows.

- We present an implementation of the KECCAK-*f*/*p* permutation as custom instructions for a vector processing unit. The proposed instructions support sponge and duplex constructions with KECCAK in four different state sizes (200, 400, 800 and 1,600 bits). The design is made for a (128-bit x 128-bit → 128-bit) ARM NEON interface. We discuss the trade-offs made in the design space of vector-instruction extensions.
- We demonstrate the proposed instructions by implementing a collection of six cryptographic algorithms based on KECCAK, including the SHA3-512 hash, the authenticated ciphers LAKE KEYAK and RIVER KEYAK, the authenticated ciphers KETJE SR and KETJE JR, and a pseudo-random stream generator [2], [8], [9].
- We evaluate the performance of the resulting designs in the context of an ARMv7 micro-architecture. We use the GEM5 architectural simulator, which provides binary compatibility with the ARM ABI, but which does not have an identical cycle-accurate behavior [10]. Therefore, we adopt the performance metric instructions/byte. In the results section, we elaborate on the relevance of the proposed metric and on the accuracy of our results compared to actual implementations.

An earlier version of this work was presented at HASP'16 [11]. This paper extends upon the preliminary workshop version by adding following contributions.

- An in-depth design space exploration for a flexible KECCAK instruction set is presented. We elaborate on all the important aspects of the design process from instruction design methodology to possible ways of partitioning the large KECCAK state into register size blocks. We have also included a design comparison with a patent application by Intel for KECCAK (SHA-3) that uses 512-bit vector unit.
- We have added performance results for a KECCAK based Pseudo random stream generator to further strengthen our claim that our instruction set extensions are capable of supporting all symmetric cryptography operations built on top of KECCAK.
- A new section has been added that provides example of implementing an efficient `In-place` KECCAK permutation using the proposed instruction extensions. We provide optimal register allocations alongside code-snippets for implementing KECCAK on an ARMv7 based processor with the NEON SIMD unit.
- To further increase the usefulness of this work, three design patterns have been proposed for KECCAK. These design patterns are independent of instruction set architecture, and thus can be used to design instruction set extensions for any platform ranging from an 8-bit microcontroller to larger platforms that support 512-bit wide SIMD processing.

The paper is organized as follows. Section 2 justifies the context of the proposed solution, and documents earlier efforts in this area. Section 3 discusses the background of the KECCAK permutation function, and its use in various cryptographic algorithms. Section 4 presents a design space exploration, and explains the design principles behind our instruction set extensions for computation of KECCAK. Section 5 gives the detailed definition of each proposed instruction. Section 6 describes the use of the instruction extensions, and the main cryptographic operations with KECCAK. Sections 7 and 8 evaluate the performance results, and discuss portability aspects. We then conclude the paper.

TABLE 1
SIMD Crypto Instructions and Throughput for Crypto Primitives on Mainstream CPU

Primitive	Intel (# instr)	ARMv8 (# instr)	Throughput (# rounds/instr)
AES	6	4	1
SHA-1	4	6	4
SHA-2	3	4	2
Carrier-less Mul	1	1	NA

A set of instructions is needed to support key schedule, encryption, decryption. Once the computation is set up, multiple rounds can be completed per instruction.

2 MOTIVATION

In this section, we motivate the proposed design and we discuss related work.

2.1 Hardware Acceleration with Custom Instructions

First, we justify the strategy of integrating cryptographic hardware as vector instruction-extensions in a processor. It is commonly assumed that the addition of new instructions to the instruction set of a processor is a risky and complex undertaking. Not only does it require a thorough understanding of the microprocessor micro-architecture, it also impacts the processor's software tool-chain, including the compiler and assembler.

However, there are several strong arguments in favor of the proposed custom-instruction methodology, and they are clarified as follows.

- There is a trend towards open processor architectures, which are more amendable to tweaking. In this spirit, the academic community is developing the RISC-V processor [12]. But also commercial micro-architectures are no longer completely closed. Using a proper licensing scheme, ARM processors can be modified before integration into a System-on-chip, and silicon system houses are using this capability as a feature differentiator. While we advocate for a KECCAK vector instruction set as a standard feature in every embedded processor with a cryptographic processing requirement, it is clear that contemporary processor design flows support instruction specialization.
- Cryptographic acceleration by custom-instructions has already been used by processor vendors in other algorithms. For example, several families of ARM and Intel processors support the Advanced Encryption Standard (AES-NI), SHA-1 and SHA-2, and carry-less multiplication (Table 1). Each primitive is supported by a handful of instructions.
- Because the proposed instruction extensions have a specific purpose, we do not require extensive

compiler support for them such as low-level code-generation out of a high-level language. Instead, support at the assembly level is adequate. This limits the complexity of the required software infrastructure.

- For highly complex system-on-chip design, the use of instruction extensions is preferable compared to the use of memory-mapped acceleration units. Memory-mapped acceleration units face increasing latency because of complex on-chip communication structures, such as multi-level buses and shared communication pathways. Moreover, the management of sensitive cryptographic state over distributed, memory-mapped acceleration units in an SoC is an additional risk, when we consider that the memory space is a shared resource.
- Vector-instructions operate at the apex of the memory-hierarchy. Provided that the cryptographic state of an algorithm can be kept inside of the processor registers, vector-instructions will eliminate the effects of memory latency. When crypto-kernels iterate over the cryptographic state for several rounds, this performance benefit becomes significant because the full processor speed is available for the entire duration of the kernel execution.

2.2 Hardware Acceleration of Keccak

Keccak is well known for its outstanding performance as a dedicated hardware kernel [13]. For our purpose, we are looking for outstanding performance *including the integration with software*. This subtle point is motivated as follows. Cryptographic hardware does not operate in isolation, but instead it is needed to support information security services. This system integration requires an efficient interface to software, such as for example a network protocol stack, an operating system device driver, or a file system.

Several authors have studied cryptographic hardware over a standard hardware interface. Saarinen describes the integration of Keccak-based authenticated encryption hardware as a memory-mapped coprocessor [14]. Another example of a standard interface for crypto is by Homsirikamol [15]. Unlike custom instructions however, coprocessor hardware cannot leverage the highly optimized micro-architecture infrastructure of the processor.

On the other hand, there are very few efforts on custom-instruction design for Keccak. Constantin et al. propose a custom-instruction design for a 16-bit micro-controller [16]. They describe a set of three instructions that offers a 30 percent cycle count reduction for SHA-3, and a 30 percent reduction in memory footprint (`text` segment). Wang et al. describe the integration of a 64-bit Keccak datapath into a 32-bit LEON3 processor for accelerating SHA-3 [17]. They report 87 percent reduction in cycle count and a 10 percent reduction in memory footprint. A recent patent application describes a Keccak instruction set for Intel AVX512, which is a 512-bit register extension [18].

We are not aware of vector-instruction designs for SHA-3 on a 128-bit ARM-NEON Interface. There are several earlier efforts in implementing Keccak software using normal (existing) vector instructions. A prominent collection is the KeccakCodePackage, a software library with hand-optimized SHA-3 implementations for 8-bit (AVR), 32/64-bit, 128-bit (ARM-NEON), 256/512-bit (Intel AVX2, AVX512) Keccak operations [19]. We will compare our results against the results reported in this library, and demonstrate that our proposed design clearly outperforms existing implementations even as they use the regular ARM NEON instruction set.

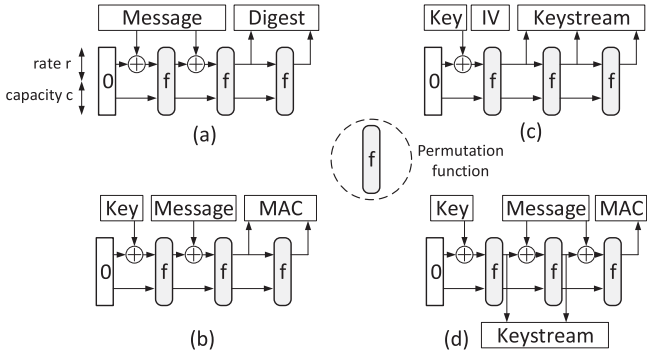


Fig. 1. Constructions with sponge: (a) Hash, (b) Message authentication code, (c) Keystream generation. Construction with duplex: (d) Authenticated encryption.

3 The Cryptographic Sponge

Cryptographic sponge functions are a class of algorithms that operate on variable-length inputs and that produce variable-length outputs based on a fixed-length permutation function. The ability to generate variable-length outputs with a tunable security level allows the sponge to perform multiple types of symmetric cryptography. The sponge construction is built using three components: a b bit state, a state permutation function f and a padding rule to adjust the input stream length to a multiple of the sponge bitrate r. The sponge capacity c is defined by $b - r$ and defines the security level. A sponge operates in two phases, called the absorbing and squeezing phase respectively. The absorbing phase integrates r bits of padded input at a time, each time permuting the state. The squeezing phase extracts r bits of output at a time, each time further permuting the state. An alternate operation of sponge, called the duplex construction [3], interleaves absorbing and squeezing phases. Fig. 1 demonstrates hashing, MACing and keystream generation using the sponge mode, and authenticated encryption (AE) using the duplex mode.

3.1 Keccak Sponge Function

This work is based on a specific family of sponge functions built on top of the Keccak cryptographic permutation. We specifically chose Keccak sponge family as it is the fundamental building block for SHA-3 [20], and for two authenticated encryption candidates in the CAESAR competition, namely Keyak [8] and Ketje [9]. Additional Keccak-based symmetric cryptographic applications have also been proposed in literature recently, such as a pseudo-random generator mode (PRNG) [2] and KangarooTwelve, a parallel hashing mode [21]. A detailed description of these applications is out of scope of this paper, and our emphasis will be on the computational core of Keccak applications, the Keccak-f and Keccak-p permutations.

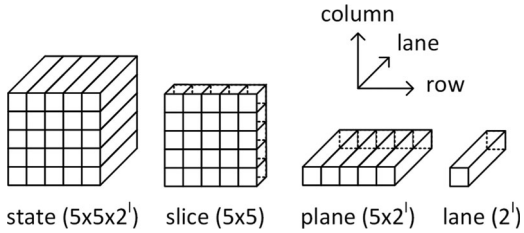


Fig. 2. Definition of terms in the KECCAK state.

3.2 KECCAK-*f* and KECCAK-*p* Permutations

The KECCAK-*f*[b] permutation is a sequence of n_r rounds on a finite state of size b bits. The size b of the state is defined as 25×2^l, where l ranges from 0 to 6. The state is organized as a three-dimensional data structure as shown in Fig. 2. Several sub-states are defined over the full KECCAK state. A *slice* is defined as a 5×5 matrix in the full state with a constant lane coordinate. A *slice* is always 25 bits irrespective of the type of KECCAK-*f* permutation. A *lane* is an array of w bits of state with constant row and column coordinate. Depending upon the KECCAK-*f* permutation 1,600, 800, 400 or 200, the *lane* size changes to 64, 32, 16, 8 bits respectively. Five *lanes* in a row comprise one *plane*. The bit-sequence in the KECCAK state is lane-wise, row-major. If $a[i, j, k]$ is the bit mapped into row i, column j and lane k, then that bit has index $(5.i + j).w + k$ in the $5 \times 5 \times 2^l$ bit input block.

The number of rounds of the permutation n_r is determined by the width of the permutation, $n_r = 12 + 2l$. In each round of KECCAK-*f* the state goes through 5 steps (transformations)

$$R = \theta \; o \; \rho \; o \; \pi \; o \; \chi \; o \; \iota.$$

An in-depth explanation of the KECCAK permutation can be found in the KECCAK reference [22]. A brief definition of each KECCAK step is as follows:

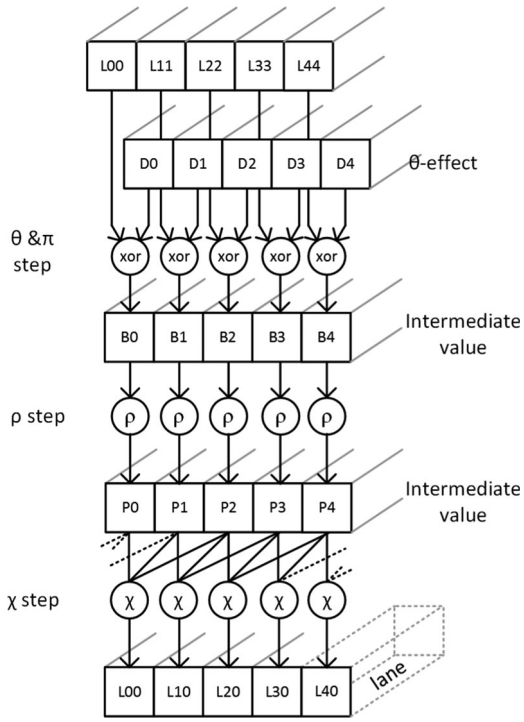


Fig. 3. Data dependencies of θ, ρ, π and χ step (one plane).

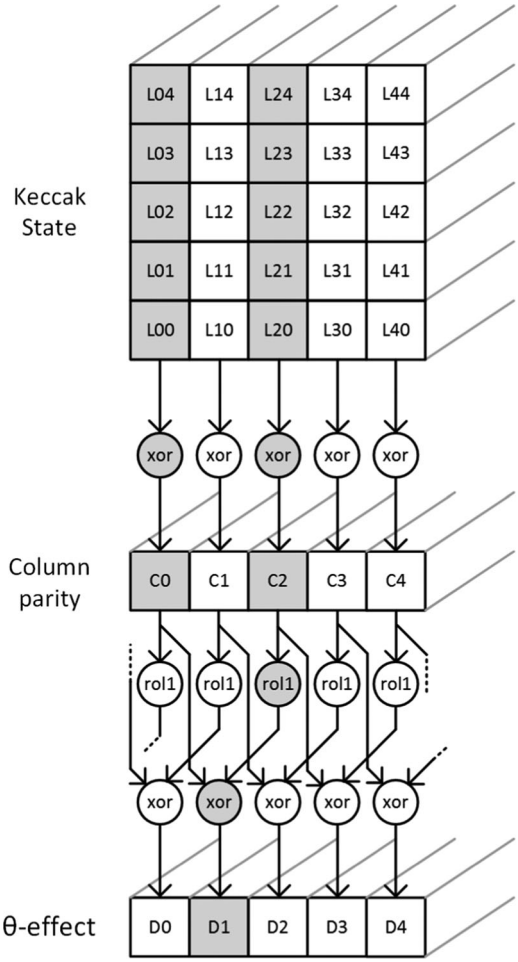


Fig. 4. Data dependencies of θ-effect values. The shading illustrates the dependencies for a single lane.

θ : For each column i, the column parity C_{i-1} and C_{i+1} of columns $(i-1)\ mod\ 5$ and $(i+1)\ mod\ 5$ is calculated respectively. C_{i+1} is left rotated by offset 1 and XORed with C_{i-1}. The resulting value D_i is XORed into each lane of column i.

ρ : All lanes in the state are left-wise rotated by a fixed offset.

π : All lanes in the state are scrambled in a fixed pattern.

χ : Each bit of each lane is non linearly combined with the bits of nearby lanes.

ι : A w bit constant is XORed into a single lane.

The KECCAK-*p*[b, n_r] permutation is a generalized version of KECCAK-*f* permutation with tunable number of rounds n_r. They form the basis of KEYAK and KETJE.

Fig. 3 shows the data dependencies of θ, ρ, π and χ step for one KECCAK plane. The lanes are denoted by L_{xy} and are selected based on π step. B_i denotes the intermediate values after θ and π step. P_i denotes the intermediate values after ρ step. Fig. 4 shows the data dependency graph for calculation of θ-effect (D_i). The column parity for each column is denoted by C_i.

4 DESIGN SPACE EXPLORATION

A sponge function can be used in constructions to design a wide range of cryptographic applications. Fig. 5 illustrates

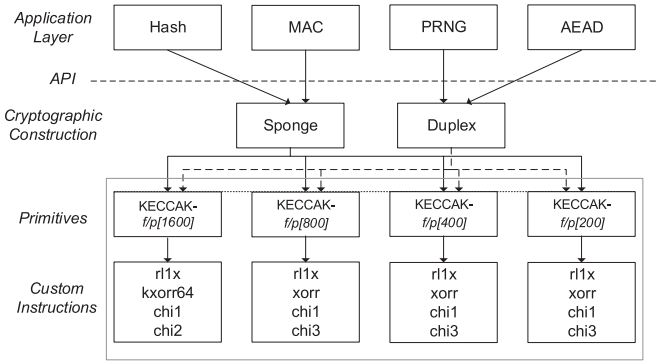


Fig. 5. KECCAK modes of operation (hierarchical view).

the hierarchical dependencies of these constructions in a cryptographic library. All the KECCAK modes are implemented using either the Sponge or else the Duplex construction. These constructions in turn are created with one of the KECCAK-*f* or KECCAK-*p* primitives depending upon the security goal and throughput requirement of the application.

Although KECCAK was one of the top designs in the SHA-3 competition in terms of software and hardware performance, the current software implementations are still compute-intensive in the permutation. For example, the 24 rounds of KECCAK-*f*-[1,600] require 4,296 operations [13]. These can be broken apart as 55 operations per round for θ, 24 for ρ, 75 for χ and 1 for ι. Furthermore, the π step requires 24 register reordering operations. As a result, profiling results of SIMD implementations of KECCAK applications like SHA3-512 and LAKE KEYAK show that permutation functions KECCAK-*f* [1,600] and KECCAK-*p* [1,600, 12] consume 99 and 65 percent of CPU cycles respectively.

Our design goal is to develop an efficient instruction set that can accelerate a range of applications based on KECCAK. To achieve this within a reasonable hardware overhead and design complexity, our instructions target the main computational core of KECCAK applications: KECCAK-*f* and KECCAK-*p* permutations. Our design supports all the four relevant sizes of 1,600, 800, 400 and 200 bits, so that all the KECCAK based cryptographic applications that have been proposed in the literature can benefit from our proposed instructions

4.1 Custom-Instruction Design Methodology

The basic methodology of custom-instruction design is to partition the data-dependency graph (Fig. 3) into multiple sub-graphs that fit the template of two-operand custom-instructions (128-bit x 128-bit $\rightarrow$ 128-bit), and then to allocate a custom-instruction for each sub-graph. This partitioning is driven by several constraints. First, the design should try to cover the entire graph with a minimum set of unique sub-graphs (custom-instructions) [23].

Second, the schedule-length of these sub-graphs needs to be as short as possible, so that the custom-instructions have low logic complexity and they are combinational. The rationale behind this constraint is that we want the new instructions to be as fast as the existing ones, and compatible with the modern CPU features such as out-of-order execution, the memory hierarchy, and automatic detection of instruction-level parallelism.

While programming the KECCAK round with custom instructions, we aim to store as few intermediate results as possible, in order to minimize the register pressure on the processor. For example, when possible, we would like to utilize in-place operations (destination register is same as one of the two source registers) when mapping the KECCAK round into custom-instructions. In addition, we aim to minimize the register re-ordering overhead (register-to-register moves).

In addition to the above generic design principles, we also made two specific considerations with respect to mapping the KECCAK state into registers. These are explained in the next section.

4.2 Cutting the KECCAK State

Compared to the standards like AES, SHA-1, 2, KECCAK has a relatively large state size of upto 1,600 bits. *Cutting the* KECCAK *state* refers to the partitioning of the 1,600 bits of state over processor storage elements, including the registers. The efficient management of the KECCAK state is of paramount importance, since poor allocation of the state over registers may result in excessive register-move operations or spills to memory.

In the case of SIMD-like instructions, which use instruction patterns of 2×128 bit input and a 128-bit output, this large state needs to be partitioned into multiple 128-bit registers. There are multiple ways to achieve this, namely using slice-wise, plane-wise and bit-interleaving techniques. A good overview for software/hardware implementations that use these techniques can be found in [13] and [24]. Here, we focus on the merits and demerits of following one or more of these approaches in context to our custom-instruction design methodology.

4.2.1 Slice-Wise versus Plane-Wise Processing

A $5 \times 5 \times w$ bit KECCAK state can be either viewed as w slices of 25 bits each (slice-wise partitioning) or else as 5 planes consisting of 5 lanes of w bits each (plane-wise partitioning).

So far, two important slice-wise implementations of KECCAK have been proposed in literature. Jungk et al. proposed a very compact slice-oriented KECCAK hardware, based on the observation that all KECCAK steps except ρ can be done efficiently with slice-wise processing [25]. They stored the KECCAK state in 25 8×8 distributed RAMs and rescheduled the KECCAK round function to perform the π, χ, ι and θ steps together. The ρ step is done separately in a multi-cycle fashion by re-arranging the contents of the distributed RAMs for lane rotations.

A second slice-wise design is described in a patent application by Intel [18] for performing SHA-3 on AVX-512 using 512-bit registers. The authors describe a methodology in which 64 slices of the KECCAK state are divided into four registers with 16 slices each. A copy of state is maintained in another four registers. Two instructions *keccak_theta* and *keccak_round* of the format (src/dest x src x src $\rightarrow$ dest) process a KECCAK round in approximately 8 instructions. The implementation details of the instructions are not clearly stated but the instructions appear to be multi-cycle hardware units that maintain state.

From the point of view of designing a flexible instruction set for KECCAK, slice-wise processing appears to be a very attractive approach. Indeed, all KECCAK steps except ρ can

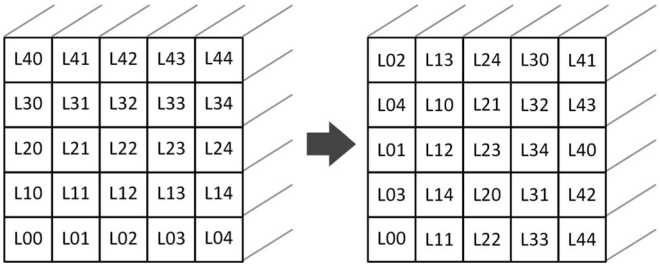
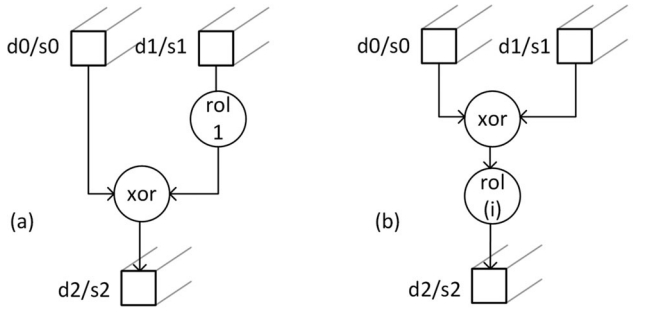


Fig. 6. Data dependencies of the π *step*.




```
rl1x.u64   d2, d0, d1      kxorr64  d2, d0, d1, #i
rl1x.u32   s2, s0, s1      xorr.u32 s2, s0, s1, #i
rl1x.u16   s2, s0, s1      xorr.u16 s2, s0, s1, #i
rl1x.u8    s2, s0, s1      xorr.u8  s2, s0, s1, #i
```

Fig. 7. a) Functionality of `rl1x` instruction, b) Functionality of `kxorr64`, `xorr` instruction.

be done on independent slices, and the size of a slice is always fixed to 25 bits. However, this approach becomes complicated for a complete permutation since input messages for absorption generally arrive in lane-oriented fashion, making slice-wise storage expensive in terms of data movement for the absorption phase and squeezing phase of sponge. Also, a 128-bit wide register can hold only 5 complete slices, so it takes a minimum of $\lceil 64/5 \rceil = 13$ registers to store the KECCAK state. Calculating the ρ step, which requires state from all the 13 slices to perform lane rotations, is difficult using standard instructions of format 128-bit x 128-bit $\rightarrow$ 128-bit.

For these reasons, we abandoned the slice-wise partitioning in favor of the plane-wise partitioning, in which adjacent bits in a register belong to the same lane. We note that this choice was also made by other software implementations for SIMD or 64/32-bit [13]. The plane-wise partitioning approach allows for simple absorption and squeezing phases and low logic complexity as shown in the results of this work. We address the main challenge of the plane-wise approach by designing our custom instructions for bit-wise data processing. The instructions compute on the w bits of a lane using w parallel single-bit operations.

4.2.2 Interleaved versus Non-Interleaved Lanes

After a KECCAK round completes, each KECCAK lane depends on 3 lanes, 3 θ-effect values, ρ offsets and a ι constant (not shown in Fig. 3). For KECCAK $f[1{,}600]$, this requires access to at least 6×64-bits of register storage to finalize a lane in a single custom-instruction. This is above the register access bandwidth for NEON-style instructions. An alternative approach called bit-interleaving [13] can be used to break large 64-bit lanes of KECCAK into smaller chunks.

In *factor 2 bit-interleaving*, a 64-bit lane U is coded as two 32-bit words U_0 and U_1, one with lane bits with even lane-coordinates and the other with odd lane-coordinates. Boolean operations can be simply implemented with bitwise Boolean instructions operating on the interleaved words and any even 2τ bit lane rotation can be achieved by rotating U_0 and U_1 independently as $U_0 \leftarrow ROT32(U_0, \tau)$ and $U1 \leftarrow ROT32(U1, \tau)$. A rotation with an odd offset $2\tau + 1$ can be achieved by $U_0 \leftarrow ROT32(U_1, \tau + 1)$ and $U_1 \leftarrow ROT32(U_0, \tau)$.

If the lanes L_{xy} in Fig. 3 are 32-bit wide, then up to 8 lanes can be packed together in two 128-bit registers. A NEON instruction can operate on these registers to finalize as many as four 32-bit lanes, if we assume that correct ρ offsets are somehow implicitly given to the instruction. However, we do not take this approach because of two reasons. First, the π step of the permutation scrambles the lanes in every round (Fig. 6). This adds to the complexity, as for every round we have to first reshuffle the lanes scattered over different registers into two 128-bit registers. Second, doing the ρ step on multiple lanes of a plane in parallel requires multiple barrel shifter units in hardware, which increases the hardware cost. Finally, bit-interleaving and de-interleaving the lanes has its own performance overhead which itself requires dedicated instruction support. For these reasons, we decided to work with non-interleaved lanes in our design, and to break up the KECCAK round in multiple instructions. In the next section, we describe the proposed custom instructions.

5 PROPOSED INSTRUCTION SET EXTENSIONS

Our design is mapped into six custom instructions based upon our analysis of data-flow diagrams of KECCAK in the context of our design principles. Similar to other crypto-instructions (e.g., Intel AES-NI and SHA), our instructions take advantage of the wide SIMD registers but not all of them operate on sub-vector formatted data. Their shape and functionality is highly customized depending upon the step and KECCAK permutation and they operate on quad (128-bit), double (64-bit) or single word (32-bit) registers. These registers are represented as `qi`, `di` and `si` respectively in NEON assembly programming and are aliased, i.e., a 128-bit quad word register can be viewed as two double word or four single word registers. We optimized our instructions for the 1,600 and 800 primitives, and we reuse the same instructions for implementing 400 and 200 primitives. The proposed instructions comply to ARM NEON instruction format and the instruction encoding widths as well. They do not require any special architectural features such as non-standard register files, implicit register operand or pipelined execution units.

5.1 Instruction rl1x

Instruction `rl1x` (*rotate left by 1 and XOR*) takes 2 registers as input (Fig. 7a), left rotates the value in source register 2 by one, and XORs the resulting value to source register 1. The `rl1x` instruction accelerates the calculation of θ-effect value that is needed for the θ step. Once the parity of all the

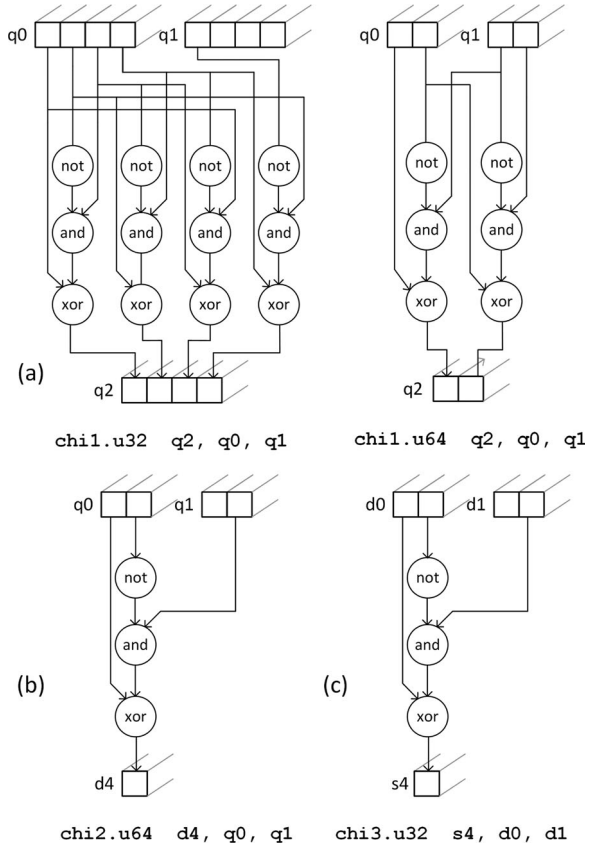


Fig. 8. Functionality of instruction a) `chi1`, b) `chi2` and c) `chi3`.

five columns is calculated and stored in five registers, the `rl1x` instruction operates on any two parity values and computes one θ-effect value per instruction. The instruction supports double and single-word NEON registers with vector sizes of 64, 32, 16 and 8 bits for supporting θ-effect calculation for KECCAK variants 1,600, 800, 400, 200.

5.2 Instruction kxorr64

Instruction `kxorr64` (KECCAK *XOR and rotate*) uses two registers and an immediate value as operands, XORs the source registers, left-rotates the result by an immediate offset and returns the result in a destination register. `kxorr64` combines θ, ρ and π steps in a single instruction. Fig. 7b shows the functionality of `kxorr64` instruction. Source operand 1 contains a lane that needs to be transposed to a new location for π step, source operand 2 contains the corresponding θ-effect value, and the immediate field contains the required ρ offset value. `kxorr64` applies θ and ρ steps and assigns the result to a new destination register (π step).

`kxorr64` is only used for implementing KECCAK-$f, p[1{,}600]$ permutations as it requires a 5-bit immediate value in the instruction encoding. This is to support 25 different rotation offsets for 25 KECCAK lanes. Accommodating 25 different offsets for all the four permutations in a single instruction is challenging, as NEON instruction format supports only 32-bit encodings for instructions. Hence, for permutations of other sizes we provide a separate instruction `xorr`.

5.3 Instruction xorr

The functionality of `xorr` instruction is similar to `kxorr64` in Fig. 7b. The only difference is that it operates on single-word registers. Similar to `kxorr64`, `xorr` combines θ, ρ and π steps in a single instruction, but supports 32-, 16- and 8-bit rotations for KECCAK-$f[800, 400, 200]$. Depending upon the vector size specified, `xorr` treats the values in the registers as 1×32, 2×16 or 4×8 bit vectors and applies the same operation on all the vectors. The rotation-argument is encoded as a 5-bit immediate constant.

5.4 Instruction chi1

`chi1` instruction supports the χ step of KECCAK. The `chi1` instruction accepts four lanes (in two quad registers) containing the intermediate values after π step, and applies the χ step on them to finalize two KECCAK lanes. Like other NEON instructions, the `chi1` instruction also supports multiple register views. A quad register can be viewed as 2×64 bit lanes (used in KECCAK-$f[1{,}600]$) or 4×32 bit lanes (for KECCAK-f [800, 400, 200]). The functionality of both the forms is shown in Fig. 8a.

5.5 Instruction chi2

Since a KECCAK plane has an odd number of lanes, the `chi1` instruction can finalize only the first four lanes of the plane. Finalizing the last lane for every plane requires register rearrangement followed by `VBIC` and `VEOR` NEON instructions. To aid this step, we provide `chi2` instruction which can save these extra computations. Fig. 8b shows the functionality of `chi2`. `chi2` instruction accepts two quad-word registers and produces a double-word register. The `chi2` instruction fits in the category of narrow instructions—the destination register is smaller then the source registers. In general, two `chi1` instructions paired with a `MOV` and a `chi2` instruction can apply the χ step to a complete KECCAK-plane. The `chi2` instruction applies only to KECCAK-$f, p[1{,}600]$ permutations. For processing other variants of KECCAK, we have designed a `chi3` instruction.

5.6 Instruction chi3

Just like the `chi2` instruction, the `chi3` (Fig. 8c) instruction is also an auxiliary instruction that helps fixing the last lane of a plane without requiring register rearrangement and `VEOR` and `VBIC` instructions. It takes two double-word registers as input source operands, producing a 32-bit single-word as result. A `chi1` instruction followed by a `chi3` instruction can apply χ step on a complete KECCAK plane. `chi3` instruction is used for KECCAK-$f[800, 400, 200]$ permutations. Since the χ step only contains XOR, NOT and AND operations we support only `U32` vector size for `chi3`. Lanes of sizes 16- and 8-bits can also be stored in 32-bit single-word registers and `chi1` followed by `chi3` can perform χ step for KECCAK-$f[400]$ and KECCAK-$f[200]$.

6 KECCAK USING CUSTOM INSTRUCTIONS

In this section, we describe how the proposed custom instructions can be used efficiently to code a functionally correct KECCAK permutation. The coding of an optimized assembly implementation of KECCAK requires a complex interplay of decisions related to register allocation, instruction minimization and various other architectural parameters like instruction level parallelism and register pressure. In our work, we created hand-optimized

TABLE 2
Keccak Applications

Application	Permutation
SHA3-512	KECCAK-*f*[1,600]
LAKE KEYAK	KECCAK-*p*[1,600,12]
RIVER KEYAK	KECCAK-*p*[800,12]
KETJE SR	KECCAK-*p*[400,*n*]
KETJE JR	KECCAK-*p*[200,*n*]
KECCAK PRNG	KECCAK-*p*[1,600,*n*]

versions of four KECCAK permutations (1,600, 800, 400 and 200) for applications listed in Table 2. Here, we use the example of in-place calculation of KECCAK-*f*[1,600] permutation using our custom instructions on a NEON based ARMv7 architecture.

6.1 NEON Instruction Format

The 128-bit SIMD unit in ARMv7 supports sixteen quad-word (128-bit) registers, which are aliased with thirty-two double-word (64-bit) registers as shown in Fig. 9. Additionally, the first sixteen double-word registers are aliased with thirty-two single-word (32-bit) registers.

NEON instructions use a three-register format (2 source operands and 1 destination operand) or a format with 3 registers and an immediate value (VEXT instruction). They have a fixed encoding size of 32 bits and operate on Quad (Q), Double (D) and Single (S) word registers depending upon the instruction definition. NEON instructions also use the data type specifiers. For example, the `I32` specifier in `VADD.I32 q1, q2, q3` signifies that quad registers `q1`, `q2` and `q3` have 4x32-bit integer data. NEON instructions can re-interpret the format of a register (Q, D, S) from instruction to instruction.

6.2 Implementing KECCAK f/p[1,600]

Fig. 10a) demonstrates an efficient approach of organizing 25 lanes of a KECCAK-*f*[1,600] permutation into 25 of the available 32 NEON registers. Since ARM NEON already supports SIMD type XOR operations on 128-bit registers, the column parities required for θ step can be very efficiently calculated if the lanes are organized in contiguous registers.

The column parities of five columns can be calculated using `VEOR` instructions. After the parity calculation step, 30 out of 32 registers of NEON are in use. Next, the θ-effect can be computed from the column parities using the instruction `rl1x`. This can be done without any additional register spills.

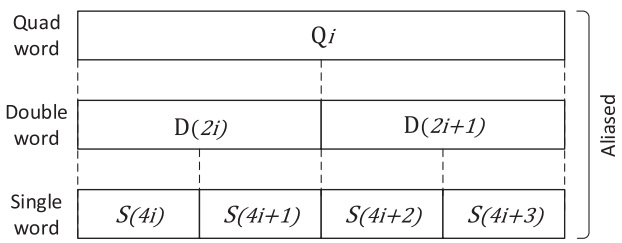


Fig. 9. Aliased NEON registers in ARMv7.

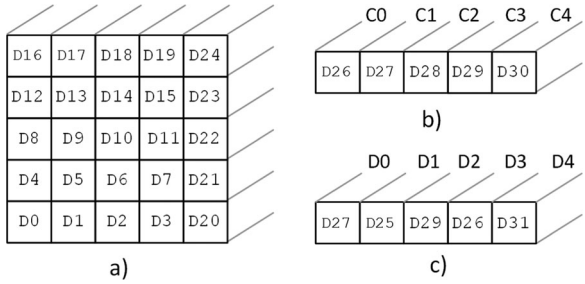


Fig. 10. Register allocation for KECCAK-*f*$p[1,600]$ a) Pre-θ step b) Column parities c) θ-effect constants.

```
@Create theta effect
veor.64  q13, q0,  q2
veor.64  q14, q1,  q3
veor.64  q15, q10, q11
veor.64  q13, q4,  q13
veor.64  q14, q5,  q14
veor.64  d30, d30, d31
veor.64  q13, q6,  q13
veor.64  q14, q7,  q14
veor.64  q13, q8,  q13
veor.64  q14, q9,  q14
veor.64  d30, d24, d30
rl1x.u64 d25, d2,  d20
rl1x.u64 d31, d21, d2
rl1x.u64 d2,  d20, d24
rl1x.u64 d20, d3,  d21
rl1x.u64 d21, d24, d3
```

Next, the θ, ρ and π step can be computed. We use a single instruction, `kxorr`, to achieve their combined effect. This instruction accepts one lane of the state, a θ-effect value and a ρ rotation table index and computes a single KECCAK lane. The resulting register organization after completion of θ, ρ and π step is shown in Fig. 11a.

```
@Do rho, pi theta
kxorr.u64 d0,  d0,  d21, #0
kxorr.u64 d6,  d14, d20, #12
kxorr.u64 d7,  d19, d2,  #18
kxorr.u64 d14, d15, d2,  #13
kxorr.u64 d19, d18, d20, #17
kxorr.u64 d15, d26, d31, #19
kxorr.u64 d18, d9,  d25, #11
kxorr.u64 d26, d29, d2,  #23
kxorr.u64 d9,  d10, d20, #7
kxorr.u64 d29, d12, d21, #15
kxorr.u64 d10, d8,  d21, #10
kxorr.u64 d12, d23, d31, #4
kxorr.u64 d8,  d1,  d25, #1
kxorr.u64 d23, d30, d31, #24
.....
```

Instructions `chi1` and `chi2` can then be used to apply *chi* step on the state. For a single KECCAK plane this looks as follows:

```
@chi
vmov.64 d21, d0
chi1.u64 q13, q1, q10
chi2.u32 d20, q10, q0
chi1.u64 q0, q0, q1
```

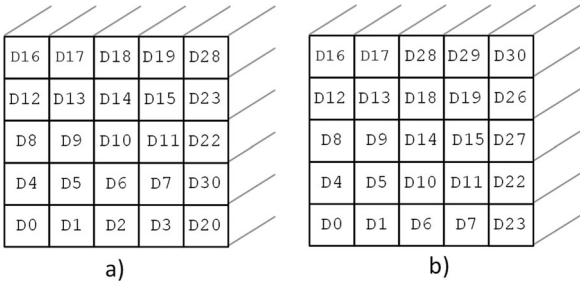


Fig. 11. Register allocation for KECCAK-*f*[1,600] a) post π step b) post χ step.

The ι step requires a load operation (reading the round constant) followed by an XOR operation

```
@Do iota
vld1.64  d30, [r1:64]!
veor.64  d0, d0, d30
```

Fig. 11b shows the final register organization. We did not keep the starting and ending organization of the state same in our implementation as it allowed us to save few `MOV` instructions. This is no problem because for most applications, the required number of KECCAK rounds is even. By unrolling the permulation twice, the original register organization shown in Fig. 11a is achieved every two rounds.

The same approach can be used for implementing the smaller versions of KECCAK by using the same register organization on 32-bit scalar registers along with proper data type specifiers (`u32, u16 and u8`) supported by our KECCAK instructions.

7 EVALUATION

In order to evaluate the performance of the proposed instructions, we designed several experiments to compare them against previous proposed software implementations of the KECCAK permutation. We used the open-source GEM5 simulator for implementation and benchmarking of the proposed custom instructions.

GEM5 is a modular simulator that supports cycle accurate simulation of system architecture and processor micro-architecture. It has support for all major instruction sets (ARMv7, x86 and more) on top of generic CPU models. For our experiments we were just interested in relative performance comparisons of KECCAK applications on ARMv7. So, we added our instructions to the ARMv7 part of GEM5 (shown in Fig. 12) which includes the instruction decoder and ISA module. The functionality of the instructions was defined in GEM5's ISA description language. Since, GEM5 executes native ARM binaries, we modified the assembler part of the GNU C compiler to create the binaries. We selected unused NEON opcodes from the ARM architecture reference manual and were able to add the instruction encoding for KECCAK instructions to the assembler very easily. The assembler modifications are enough for the C compiler to recognize the special instructions in the assembly source code.

Futhermore, GEM5 simulator does not support CPU model of actual ARM processors like Cortex-A8 or A9, but has a generic ISA independent CPU models (Simple, In-Order and OoO) that can inherit any supported instruction set, for example ARMv7. This allows for architectural experiments on a simulated ARM processor. In real-hardware ARM processor, the NEON SIMD unit operates with a separate register file and a dedicated pipeline of its own. Instructions are issued to the NEON pipeline from the ARM side via an instruction queue. Both the ARM pipeline and NEON pipeline can work independently as long as the queue is not full. The ARM core as well as the NEON SIMD unit also support multi-issue and factors like instruction latency, instruction queue length, issue width of the CPU are all implementation-specific and difficult to model with a high degree of accuracy on a simulator.

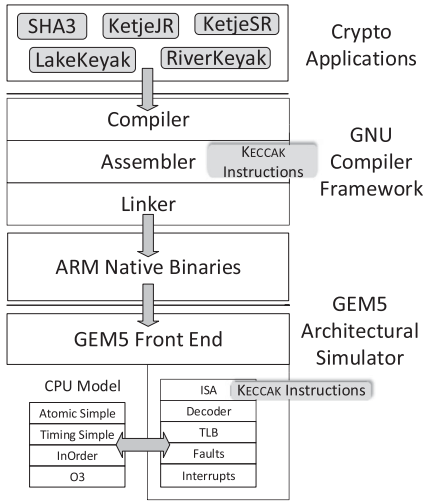


Fig. 12. Design flow for KECCAK instructions.

To get a reasonably correct estimate of the performance improvements of our custom instructions we chose a performance metric that is dependent on the instruction set of the processor and the source code, but independent of the actual hardware implementation details of the processor. Hence, all our experimental results for KECCAK applications are presented in `Instructions/Byte` metric rather than `Cycles/Byte` metric. The `Instructions/Byte` metric gives the count of dynamic instructions committed by processor per byte of input data. This gives a fair idea of the reduction in the number of committed instructions by using custom instructions and hence the net speedup. And given that all the other architectural features like issue width, instruction level parallelism (ILP), instruction timing and memory latency remains same or become better for the application with specialized instruction set when compared to baseline instruction set, the speedup should be same on a real-hardware platform as well.

7.1 Setup

For our experiments, we selected a simple-CPU timing model available in GEM5. The simple-CPU timing model executes instructions at a rate of 1 CPI (1 cycle per instruction) but takes the latency of memory system into account by adding stalls on cache accesses. We simulated a single core ARM CPU at 1 GHz with a level-1 instruction- and data-cache of 32 KB and a level-2 cache of 2 MB. For generating the executable binaries we used version 4.9.2 of the GNU Compiler Collection for ARM. For comparison purposes, we also evaluated the optimized implementations of SHA-3 (c = 1,024), KEYAK, KETJE and PRNG available in KECCAK code package [19]. We used Known Answer Tests to

TABLE 3
Performance in Instructions/Byte for Various Keccak Modes

Mode	C	32-bit ASM	NEON	CI†	Speed-up
SHA3(c=1,024)	243.5	143.9	48.1	**21.9**	2.2
KECCAK PRNG	370.9	219.3	78.1	**39.9**	1.9
LAKE KEYAK(E)	61.0	NA	13.4	**7.7**	1.7
LAKE KEYAK(D)	61.7	NA	14.9	**9.2**	1.6
RIVER KEYAK(E)	55.2	39.3	NA	**14.8**	2.6
RIVER KEYAK(D)	57.2	40.6	NA	**16.1**	2.5
KETJE SR (E)	166.1	87.9	NA	**55.0**	1.6
KETJE SR (D)	166.1	87.9	NA	**55.0**	1.6
KETJE JR (E)	309.1	146.6	NA	**106.5**	1.4
KETJE JR (D)	309.1	146.6	NA	**106.5**	1.4

† *This work. E = Encryption, D = Decryption, NA = Not Available.*

verify the functional correctness of SHA-3(c = 1,024) and PRNG. For KEYAK and KETJE we used the CAESAR testbench of the KECCAKCODEPACKAGE.

7.2 Performance

Table 3 shows the results of our optimized assembly implementations using the proposed custom instructions (CI) compared to the optimized C, hand optimized 32-bit assembly and NEON assembly implementations in the KECCAK code package. All the measurements were taken on the GEM5 simulator using the setup discussed previously. The simulation statistics were taken for the complete application with an input block size of 10, 100 and 1,000 blocks for hash and AEAD. The averaged results are shown in instructions committed by CPU per byte of input data. The expected speedup is calculated against the optimized NEON or 32-bit assembly implementations based on the availability of implementation in the KECCAK code package. Table 4 gives the instructions committed for processing a single round of KECCAK-f for different software implementations. Our implementations using CI reduce the instructions committed by CPU by a factor of 1.4 to 2.6× depending upon the application.

We believe these results are reasonably accurate compared to real hardware because of the following reasons. First, our implementations of KECCAK primitives do not incur any branches during the round computations and offer ILP (instruction-level parallelism) ranging from degree 2 to 4. Second, our instructions use simple operations like XOR, AND, NOT and rotations, which can be completed in single clock cycle in execution stage of processor's pipeline. Third, our implementations of KECCAK round do not cause any register spills that might affect throughput. Fourth, we compare all implementations (optimized C, assembly, NEON assembly and custom-instructions) under the same simulation environment. Hence, we believe that KECCAK applications that use our custom instructions should be able to achieve a speedup very close to the stated values on a real superscalar processor with low memory-latency.

TABLE 4
Instructions Executed Per Keccak Round

Primitive	C	32-bit ASM	NEON	CI
KECCAK-f[1,600]	713	414	145	**66**
KECCAK-f[800]	271	194	NA	**56**
KECCAK-f[400]	370	217	NA	**55**
KECCAK-f[200]	361	168	NA	**57**

TABLE 5
Area, GE and Transistor Count Estimates for the Proposed Custom Instructions

Instruction	μ^2	Gate equiv.	Transistor equiv.
`rl1x`	1,238	310	1,238
`kxorr64`	7,474	1,869	7,474
`xorr`	4,884	1,221	4,884
`chi1`	3,576	894	3,576
`chi2`	966	242	966
`chi3`	486	122	486
Total	18,624	4,658	18,624

7.3 Hardware Cost

To estimate the hardware overhead of our proposed design, we created RTL designs for all the six custom instructions. We assumed that the functional units will get up to 2×128 bit of input data and a 5 bit immediate field, and that they generate a 128-bit output. For synthesis, we used Synopsys Design Compiler and synthesized the RTL to a 90 nm standard cell library. The area estimates for each instruction have been provided in Table 5 with corresponding gate-equivalent (GE) and transistor-equivalent counts. An ARMv7-A based quad-core processor [26] uses an area of 3.8 mm^2 in 28 nm technology. On converting the area into GE [27], a single ARM core has around 3.8 million gates. Adding KECCAK instruction extensions will only cost an extra 4658 gates. This represents a negligible overhead compared to the significantly improved performance of hashing, MACing and a range of other cryptographic applications based on KECCAK sponge and duplex construction. Apart from the hardware overhead of functional units, the instruction decode logic will also have some hardware overhead of additional custom instructions. But since our instructions follow similar encoding format as that of other NEON instructions, the decoding overhead should be negligible.

7.4 Side-Channel Resistance of Our Design

Our current work did not investigate the vulnerability to side channel attacks on the KECCAK implementations that use the proposed custom instructions. In general, KECCAK can be implemented in a way such that its execution time is independent to the size of input block. KECCAK also does not use large table lookups, so it is not vulnerable to timing- and cache-based side channel attacks [13]. Our custom-instruction based design will benefit from these same advantages that are inherent to KECCAK. However, KECCAK modes that use secret keys may still be vulnerable to power and EM analysis based attacks. Possible side-channel countermeasure techniques like masking as discussed in [13] can be used to protect KECCAK software implementations against these attacks. However, in this work we did not employ any masking techniques.

8 PORTABILITY TO OTHER ARCHITECTURES

The instruction designs that we have proposed in this work are based on the operational context of a 128-bit SIMD

TABLE 6
Design Patterns for KECCAK

Pattern	KECCAK step
ROL(1) → XOR	θ
XOR → ROL(i) → ASSIGN	θ, ρ *and* π
NOT → AND → XOR	χ

instruction set of ARM. Any modification to the operational context of the proposed special instructions will also affect their detailed design. On the other hand, the proposed custom instructions have generic properties that we expect to be reusable in other realizations of the KECCAK custom instructions, and we will refer to these properties as *design patterns*. In this paper, the design patterns are extracted by the careful analysis of the data flow graph of KECCAK, and we caution that they are not the only way of defining such patterns. Additional design constraints such as hardware overhead, multiple algorithm variants and width of registers, may lead to significantly different design patterns. For example, the design by Yap et al., which started from the requirement of 512-bit registers, resulted in significantly different design patterns for SHA-3 [18].

Our design space exploration for KECCAK instructions, yielded three such design patterns. These patterns are listed in Table 6. We derived six instructions from these patterns using the features (register widths, data type specifiers) and specifications (opcode space, instruction encoding length) of NEON instruction set. However, the same three patterns can be used to derive instructions for any platform ranging from 8-bit micro-controller to a 512-bit wide SIMD platforms.

The application of KECCAK kernels are highly diverse ranging from lightweight algorithms like KETJE that work on a small permutation of size 200 bits, to high performance applications like SHA-3, LAKE KEYAK and Fast Parallel Hash. They utilize large permutation of size 1,600 bits and may even require multiple parallel instances. To facilitate these algorithms on smaller platforms, the bit-interleaving technique may be very useful. Once the large lanes have been broken down into n-bit lanes that can be efficiently handled on an n-bit platform, the instruction patterns in Table 6 can be applied to develop specialized instructions. Similarly, when the same patterns are extended in an SIMD fashion to develop instructions for machines with wide registers (512 bits), they can be useful to accelerate parallelized instances of KECCAK like KEYAK and Fast Parallel hash.

9 CONCLUSION

We presented the design of a vector-unit oriented instruction set for cryptographic primitives based on the KECCAK permutation. We show that six instructions are sufficient to cover permutations with four different state sizes (1,600, 800, 400 and 200 bits). We present a detailed design space exploration for the ARM NEON environment, and show that the proposed instructions can implement the largest KECCAK state with only 32 NEON registers. This allows to execute the entire KECCAK permutation inside of the ARM NEON unit.

We evaluated the proposed new instructions for six different KECCAK applications: SHA3, LAKE KEYAK, RIVER KEYAK, KETJE SR, KETJE JR and a PRNG. We did detailed simulations based on a cycle accurate GEM5 simulator. We conclude that we can achieve performance gains from 1.4 to 2.6 times over hand-optimized assembly and/or hand-optimized NEON assembly for ARMv7. The proposed instructions require less than 5,000 GE within the processor.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation Grant no 1314598.

REFERENCES

[1] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Sponge functions," Ecrypt Hash Workshop 2007, May 2007.
[2] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Sponge-based pseudo-random number generators," in *Proc. Int. Workshop Cryptographic Hardware Embedded Syst.*, 2010, pp. 33–47.
[3] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications*, 2012. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28496-0_19
[4] P. Biswas, S. Banerjee, N. Dutt, P. Ienne, and L. Pozzi, "Performance and energy benefits of instruction set extensions in an FPGA soft core," in *Proc. 19th Int. Conf. VLSI Des. Held Jointly With 5th Int. Conf. Embedded Syst. Des.*, 2006, p. 6.
[5] S. Gueron, "Intel Advanced Encryption Standard (AES) New Instruction Set," Intel White Paper, May 2010.
[6] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich, "Intel SHA Extensions – New Instructions Supporting the Secure Hash Algorithm on Intel Architecture Processor," Intel White Paper, July 2013.
[7] S. Gueron and M. E. Kounavis, "Intel Carry-less Multiplication Instruction and its Usage for Computing the GCM Model," Intel White Paper, April 2014.
[8] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "CAESAR submission: Keyak v2," Dec. 2015. [Online]. Available: http://keyak.noekeon.org/Keyak-2.1.pdf
[9] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "CAESAR submission: Ketje v2," Mar. 2014. [Online]. Available: http://ketje.noekeon.org/Ketje-1.1.pdf
[10] N. L. Binkert, et al., "The GEM5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[11] H. K. Rawat and P. Schaumont, "SIMD instruction set extensions for keccak with applications to SHA-3, keyak and ketje," in *Proc. Hardware Architectural Support Security Privacy*, 2016, pp. 4:1–4:8.
[12] B. Zimmer, et al., "A RISC-V vector processor with simultaneous-switching switched-capacitor DC-DC converters in 28 nm FDSOI," *J. Solid-State Circuits*, vol. 51, no. 4, pp. 930–942, 2016.
[13] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "KECCAK implementation overview," May 2012. [Online]. Available: http://keccak.noekeon.org/Keccak-implementation-3.2.pdf
[14] M. O. Saarinen, "Simple AEAD hardware interface (SÆHI) in a SoC: Implementing an on-chip keyak/whirlbob coprocessor," in *Proc. 4th Int. Workshop Trustworthy Embedded Devices TrustED '14 Scottsdale Arizona USA*, 2014, pp. 51–56.
[15] E. Homsirikamol, W. Diehl, A. Ferozpuri, F. Farahmand, M. U. Sharif, and K. Gaj, "A Universal Hardware API for Authenticated Ciphers," in *Proc. Int. Conf. ReConFigurable Comput. FPGAs ReConFig 2015 Riviera Maya Mexico*, 2015, pp. 1–8.
[16] J. Constantin, A. Burg, and F. K. Gürkaynak, "Investigating the Potential of Custom Instruction Set Extensions for SHA-3 Candidates on a 16-bit Microcontroller Architecture," 2012. [Online]. Available: http://eprint.iacr.org/2012/050
[17] Y. Wang, Y. Shi, C. Wang, and Y. Ha, "FPGA-based SHA-3 Acceleration on a 32-bit Processor via Instruction Set Extension," in *Proc. IEEE Int. Conf. Electron. Devices Solid-State Circuits*, Jun. 2015, pp. 305–308.
[18] K. Yap, G. Wolrich, J. Guilford, V. Gopal, E. Ozturk, S. Gulley, W. Feghali, and M. Dixon, "Method and apparatus to process keccak secure hashing algorithm," Oct. 17 2013. [Online]. Available: https://www.google.com/patents/US20130275722
[19] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "The Keccak Code Package," 2016. [Online]. Available: https://github.com/gvanas/KeccakCodePackage

[20] M. J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," *Federal Inf. Process. Stds. (NIST FIPS) - 202*, August 2015.
[21] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "KangarooTwelve: Fast Hashing based on Keccak-p," *Cryptology ePrint Archive, Rep. no*. 2016/770, 2016
[22] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The Keccak reference," 2011. [Online]. Available: http://keccak.noekeon.org/Keccak-reference-3.0.pdf
[23] K. Seto and M. Fujita, "Custom Instruction Generation with High-Level Synthesis," in *Proc. Symp. Appl. Specific Processors*, Jun. 2008, pp. 14–19.
[24] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "1001 Ways to Implement Keccak," NIST Third SHA-3 Conference, March 2012. [Online]. Available: http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/BERTONI_paper.pdf
[25] B. Jungk and J. Apfelbeck, "Area-Efficient FPGA Implementations of the SHA-3 Finalists," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, Nov. 2011, pp. 235–241.
[26] Y. Shin, et al., "28NM High-metal-gate heterogeneous quad-core CPUs for high-performance and energy-efficient mobile application processor," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, Feb. 2013, pp. 154–155.
[27] Samsung. Corporation, "Samsung Foundry 32/28nm Low-Power High-K Metal Gate Logic Process and Design Ecosystem," 2011. [Online]. Available: http://www.samsung.com/us/business/oem-solutions/pdfs/Foundry_32-28nm_Final_0311.pdf

Hemendra Rawat received the BTech degree in electronics engineering from Indian Institute of Technology (BHU), Varanasi, in 2012, and the MS degree in computer engineering from Viginia Tech, in 2016. His research interests include applied cryptography and computer architecture, and in particular, high-performance and trustworthy secure systems. He is currently working with National Instruments. He is a student member of the IEEE.

Patrick Schaumont received the PhD degree in electrical engineering from UCLA in 2004, and the MS degree in computer science from Ghent University, in 1990. He is a professor in computer engineering at Virginia Tech. From 1992 to 2000 he was a staff researcher at IMEC, Belgium. From 2001 until 2005 he was the graduate-level and post-doctoral researcher at UCLA. He joined Virginia Tech in 2005. From 2012 to 2014 he served as director for the Center for Embedded Systems for Critical Applications (CESCA), Virginia Tech. In 2014-15 he was a visiting researcher at the National Institute for Information and Telecommunications Technology (NICT) in Japan. His research interests are in design and design methods of secure, efficient and real-time embedded computing systems. He is senior member of the IEEE.