SIMD Instruction Set Extensions for Keccak with Applications to SHA-3, Keyak and Ketje

Hemendra K. Rawat, Patrick Schaumont
Bradley Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, USA
{hrawat, schaum}@vt.edu

ABSTRACT

Recent processor architectures such as Intel Westmere (and later) and ARMv8 include instruction-level support for the Advanced Encryption Standard (AES), for the Secure Hashing Standard (SHA-1, SHA2) and for carry-less multiplication. These crypto-instruction sets provide specialized hardware processing at the top of the memory hierarchy, and provide significant performance improvements over general-purpose software for common cryptographic operations. We propose a crypto-instruction set for the KECCAK cryptographic sponge and for the KECCAK duplex construction. Our design is integrated on a 128 bit SIMD interface, applicable to the ARM NEON and Intel AVX (128 bit) architecture. The proposed instruction set is optimized for flexibility and supports multiple variants of the KECCAK-$f[b]$ permutation, for b equal to 200, 400, 800, or 1600 bit. We investigate the performance of the design using the GEM5 micro-architecture simulator. Compared to the latest hand-optimized results, we demonstrate a performance improvement of 2 times (over NEON programming) to 6 times (over Assembly programming). For example, an optimized NEON implementation of SHA3-512 computes a hash at 48.1 instructions per byte, while our design uses 21.9 instructions per byte. The NEON optimized version of the LAKE KEYAK AEAD uses 13.4 instructions per byte, while our design uses 7.7 instructions per byte. We provide comprehensive performance evaluation for multiple configurations of the KECCAK-$f[b]$ permutation in multiple applications (Hash, Encryption, AEAD). We also analyze the hardware cost of the proposed instructions in gate-equivalent of 90nm standard cells, and show that the proposed instructions only require 4658 GE, a fraction of the cost of a full ARM Cortex-A9.

CCS Concepts

•**Security and privacy** → **Hash functions and message authentication codes;** *Hardware security implementation;*

Keywords

Instruction Set Extensions, KECCAK , ARM NEON, SIMD, SHA3, Authenticated Encryption

HASP 2016, June 18 2016, ,

DOI: http://dx.doi.org/10.1145/2948618.2948622

1. INTRODUCTION

In 2012, NIST announced KECCAK as the winner of the SHA-3 cryptographic hashing standard competition [18]. The cryptographic permutation of KECCAK can be used in a sponge construction to support hashing, MACing [7], encryption/decryption [4], and random number generation [5]. KECCAK can also be used in a duplex construction to support authenticated encryption [6]. Each of these applications is achieved through only minor variations of the KECCAK sponge or duplex.

On a standard block cipher, this type of multi-functional behavior can only be achieved through modes of operation (MOO), which can introduce additional complexity in the form of feedback, data dependencies, operations, and intermediate storage. The combination of an AES-128 block cipher (with only 128 bits of state) with multiple modes of operation can result in a larger and slower design than a multi-purpose KECCAK-$f[1600]$ permutation (with 1600 bits of state). Yalla *et al.* compared multi-function hardware designs based on AES and KECCAK for Virtex-7 FPGA. They conclude that an AES-based multi-functional design is not only 21% larger than a similar KECCAK-based design, but that it is also slower. The throughput of the AES-based design is three times – for hashing – to nine times – for authenticated encryption – slower [27].

In our research, we are motivated by the potential of new crypto -instructions in modern processors. Intel (Westmere, Sandybridge, Ivybridge and Haswell) and ARMv8 offer dedicated instructions to compute AES, SHA-1, SHA-256 and carry-less multiplication. These instructions operate on wide registers: 128 bit multi-media extensions (XMM) in the case of Intel and 128 bit NEON in the case of ARM. Their appearance in mainstream processors is motivated by the increasing proportion of cryptographic processing in the contemporary processor workload. Crypto-instructions are used to efficiently support secure connections (IPSec, SSL and TLS), bulk encryption and new applications such as blockchains. The crypto-instructions are specialized and used by experts; they are not meant to be targeted through a high-level language. This simplifies the software infrastructure requirements such as the need for compiler support, and it promotes the development of specialized libraries.

There are several strong arguments in favor of designing additional new instructions to support cryptographic processing. First, modern high-end processors are no longer monolithic, closed engines. ARM cores are licensed as soft-intellectual property (IP) to system design houses, which further customize and integrate these cores into their System-on-Chip (SoC) designs [17]. For example, Apple's A7 core [24] and Qualcomm's Snapdragon [19] contain customized and tested derivatives of the ARMv8 architecture. In those platforms, an optimized instruction-set serves as a market-differentiator. A second motivation for new crypto-instructions is

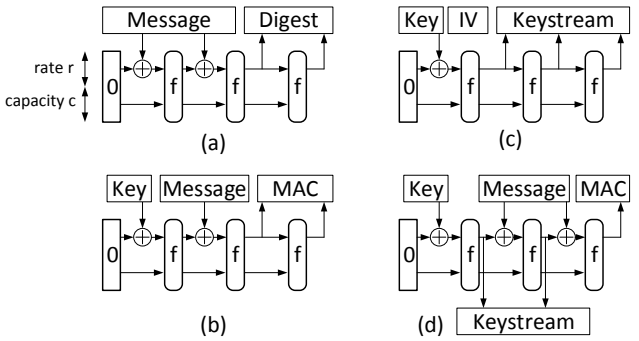

Figure 1: Constructions with sponge: (a) Hash, (b) Message Authentication Code, (c) Keystream Generation. Construction with duplex: (d) Authenticated Encryption

the unique performance advantage that can be achieved compared to traditional hardware/software designs based on memory mapped co-processors [2, 22] or dedicated stand-alone engines [13]. In contrast to these traditional designs, crypto-instructions execute at the top of the memory hierarchy. They are free from memory-latency effects as long as the entire cryptographic state fits into the processor registers. Furthermore, modern CPU engines use sophisticated (multi-issue) instruction scheduling engines, which will benefit these new instructions as well.

Common techniques in high-performance software optimization often require a trade-off between throughput and latency and/or memory footprint. For example, bitslicing, a well known technique for software throughput optimization [21], only provides high throughput when large quantities of data can be processed. The throughput gain of bitslicing comes at the expense of increased latency. In contrast, new instructions operate on the fully parallel cryptographic state, and they offer the potential of high throughput without the cost of high latency. They are also well suited when the data payload to process is small.

The paper is structured as follows. We will first summarize our novelty claims and we review related work in design of instruction-sets for KECCAK. Section 2 reviews the operation of the cryptographic sponge and motivates how this design achieves its flexibility as a universal crypto-kernel. Section 3 analyzes how to partition the KECCAK sponge/duplex into instruction-sized operations. Section 4 describes the proposed instruction set. Sections 5 offer performance and hardware-cost analysis, while section 6 discusses about the applicability of the proposed instructions to different platforms. We conclude the paper in Section 7.

1.1 Novelty Claims

- In this paper, we describe a novel mapping of the KECCAK-f permutation as six new custom instructions, that can be utilized to compute a KECCAK-based sponge or duplex of four different sizes (1600, 800, 400 and 200 bits wide).
- We demonstrate the use of these instructions by implementing 5 different KECCAK-based algorithms: the SHA3-512 hash [18], the LAKE KEYAK and RIVER KEYAK authenticated ciphers [10], and the KETJESR and KETJE JR authenticated ciphers [9]. Together, these algorithms cover all relevant primitives KECCAK-$f$and KECCAK-p[1600, 800,400, 200].
- We present a detailed performance analysis of the resulting design for native ARM binary code on ARMv7 using the GEM5 architectural simulator [12]. GEM5 provides a cycle-accurate simulation mode, but its processor timing models used are not guaranteed to match with the actual ARMv7. We therefore report our results in instructions/byte. We also provide a hardware cost analysis of the proposed instructions in 90 nm standard cell GE.
- Our design executes the KECCAK-f[1600] permutation in 65 instructions, while the most optimized NEON implementation available in the KECCAK code package still requires 144 instructions. Our design therefore not only leads to compact code, but will also deliver a significant performance increase on ARMv7 with NEON SIMD. We demonstrate similar gains for other configurations as well.

1.2 Related Work

So far, there have been only a few efforts to design instruction sets for KECCAK; most of the effort has gone into optimized implementations in hardware or in software. Constantin *et al.* propose custom-instruction designs for a 16 bit micro-controller [14]. They describe a set of three instructions that offer a 30% cycle count reduction for SHA3, and a 30% reduction in memory footprint (`text`). Wang *et al* describe the integration of a 64 bit KECCAK datapath into a 32 bit LEON3 processor for accelerating SHA3[26]. They report 87% reduction in cycle count and a 10 % reduction in memory footprint. In comparison to these designs, our work targets a multi-purpose instruction set for KECCAK applications based on a 128 bit SIMD unit.

2. CRYPTOGRAPHIC SPONGE AND KECCAK

This section gives a brief overview of cryptographic sponge functions and their modes of operation. We describe the KECCAK-f permutation, which is the fundamental building block for SHA3, and for two candidates in the CAESAR competition, namely KEYAK and KETJE.

2.1 Cryptographic Sponge and Duplex Construction with KECCAK

Figure 1 demonstrates hashing, MACing and keystream generation using the sponge mode, and authenticated encryption (AE) using the duplex mode. Cryptographic sponge functions are a class of algorithms that operate on input of variable length and produce variable length output based on a fixed length permutation. The sponge functions have 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: Absorbing and Squeezing. 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 KECCAK, called the duplex construction [6], interleaves absorbing and squeezing phases.

2.2 KECCAK-f and KECCAK-p Permutations

KECCAK-$f[b]$ is a set of set of seven iterated permutations, consisting of a sequence of n_r rounds on a finite state of b bits. The value of b is defined as 25×2^l where l ranges from 0 to 6. KECCAK-$f[b]$ organizes the b bit state as a 3D matrix of dimension $5 \times 5 \times w$, where w is defined as 2^l. The number of rounds n_r is determined by the width of the permutation, $n_r = 12 + 2l$. In each round of KECCAK-f the state undergoes a set of 5 steps (transformations).

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

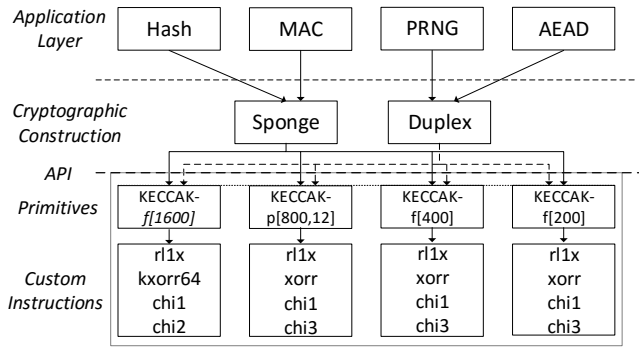

Figure 2: KECCAK application stack

The $5 \times 5 \times w$ state matrix (Figure 3) can be split into slices as well as lanes. A slice is defined as a 5×5 matrix in the state with a constant z coordinate. A lane is an array of w bits of state with constant x and y coordinate. Depending upon the KECCAK-f permutation 1600, 800, 400 or 200, the lane size changes to 64, 32, 16, 8 bits respectively. A slice is always 25 bits irrespective of the type of KECCAK-f permutation

In brief, the 5 KECCAK steps can be defined as follows:

θ : For each column i, calculate the column parity C_{i-1} and C_{i+1} of columns $(i-1) mod 5$ and $(i+1) mod 5$ respectively. Left rotate C_{i+1} by one and XOR with C_{i-1}. XOR the resulting value D_i into each lane of column i. θ step requires support for XOR and ROL (rotate left) CPU instruction.

ρ : Left rotate all lanes in the state by a fixed offset. For efficient implementation of ρ step, the CPU should support ROL instruction for word size equal to lane size.

π : All lanes in the state are transposed in a fixed pattern. This step can be done using only MOV instruction.

χ : Each bit of the lane is non linearly combined with the bits of nearby lanes using AND, XOR and NOT operations.

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

The KECCAK-$p[b,nr]$ permutation is a generalized version of KECCAK-f permutation which takes number of rounds n_r as input parameter. They form the basis of KEYAK and KETJE. An in-depth explanation of the KECCAK permutation can be found in the KECCAK reference [8].

3. DESIGN EXPLORATION FOR KECCAK CUSTOM INSTRUCTIONS

Figure 2 illustrates how the KECCAK permutation is utilized as a universal cryptographic kernel. All the KECCAK modes are implemented using either the Sponge or else the Duplex construction. These constructions use one of the four relevant KECCAK-f or KECCAK-p primitives (1600, 800, 400, 200) depending upon the security goal and throughput requirement. The design approach for KECCAK applications simplifies the software development process by abstracting the implementation details of the lower layers (primitives) from the generic top application layer (modes). It also localizes the optimization scope of KECCAK based applications to the primitives. Our profiling results for hashing (SHA3) show that 99% of CPU cycles were used for KECCAK-f [1600]. For LAKE KEYAK, 65% of the total CPU cycles were spent in doing KECCAK-p[1600, 12]

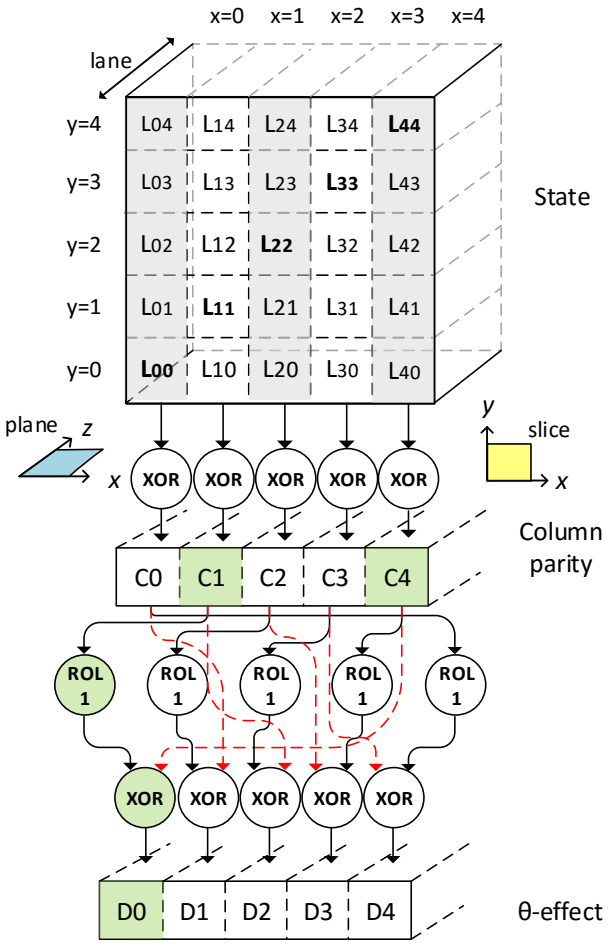

Figure 3: Data dependencies of θ-effect values

Our work takes advantage of the layered design of KECCAK based applications. We propose a set of six custom instructions to accelerate four KECCAK primitives 1600, 800, 400 and 200 and show the flexibility of our design by accelerating five different KECCAK applications namely SHA3, LAKE KEYAK, RIVER KEYAK, KETJE SR and KETJE JR. Our instructions are stateless, which means that we keep the entire KECCAK state into processor registers. Therefore, we first explain how to partition a relatively large KECCAK state (1600 bits) into smaller pieces that can be processed as instruction operands of 128 bits each. Later we will briefly discuss some features of the ARMv7 ISA and NEON SIMD that we used for designing the proposed instructions.

3.1 Cutting the KECCAK State

A $5 \times 5 \times w$ bit KECCAK state can be either viewed as w slices of 25 bits each or 5 planes with 5 lanes of w bits each. B. Jungk *et al.* propose a slice-oriented KECCAK hardware [20] which is based on the observation that all KECCAK steps except ρ can be done efficiently with slice-wise processing. They stored the KECCAK state in 25 8×8 distributed RAMs and rescheduled the KECCAK round function to perform π, χ, ι and θ steps together. Such an approach is good for custom hardware where the state can be stored in distributed RAMs and the hardware can read the state in lane-wise or slice-wise fashion efficiently depending upon the KECCAK step. From the point of view of software executing on a processor, slice-wise design may not be very efficient. First, input messages for

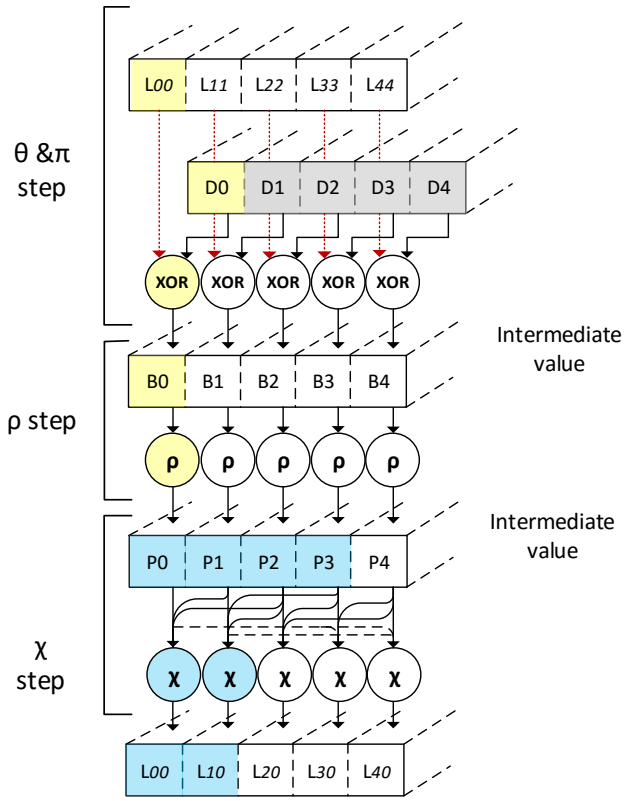

Figure 4: Data dependencies of θ, ρ, π and χ step (one plane)

absorption normally arrive in lane-oriented fashion, making slice-wise storage expensive in terms of data movement. Second, after the $\pi\ o\ \chi\ o\ \iota\ o\ \theta$ steps are done on the slices, the state needs to be transformed back into lane-wise orientation for ρ step. Therefore, all optimized software implementations [3] whether SIMD or 64/32 bit use plane wise processing.

The design of custom instructions requires the partitioning of the dataflow graph of the target algorithm into instruction patterns, such that the schedule length of the graph becomes as short as possible [23]. In the case of SIMD-like instructions, we are using instruction patterns of 2×128 bit input and a 128 bit output.

KECCAK-f can have have a state size of up to 1600 bits (25×64 bits). For a processor with 32 64 bit registers, just holding a complete KECCAK-$f[1600]$ state in CPU register consumes 25 registers, leaving just 7 free registers to hold intermediate values during the round computation. Our custom instructions have been chosen such that register spills to main memory (LOAD/STORE) during the KECCAK round are avoided, and such that that register reordering operations (MOV and VEXT) are minimized.

Figure 3 shows the data dependency graph for calculation of θ-effect (D_i). The column parity for each column is denoted by C_i. The lane size depends upon the KECCAK-f permutation, but here we will assume KECCAK-$f[1600]$ with lane size and intermediate round values of 64 bits. ARM NEON already supports XOR instructions on 128 bit registers, so the column parities can be very efficiently calculated. To support calculation of θ-effect from column parities, we add an instruction which combines XOR with ROL(1) (rotate left by 1).

Figure 4 shows the data dependencies of θ, ρ, π and χ step for one KECCAK plane. ι step is not shown for the sake of brevity. 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

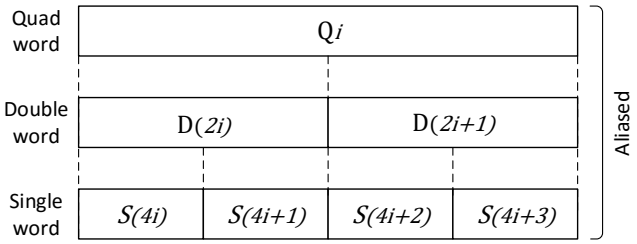

Figure 5: Aliased NEON registers in ARMv7

intermediate values after ρ step. In a round, a single processed KECCAK lane after ι step depends on 3 lanes, 3 θ-effect values, ρ offsets (not shown) and ι constant. Two 128 bit input registers of an instruction cannot hold all these values. One solution is to split the 64 bit lane into two independent 32 bit lanes using the bit-interleaving technique [3], so that up to eight 32 bit lanes can be packed in two 128 bit registers. We do not take this approach because of two reasons. First, breaking a 25 lane (64 bit) state into a 50 lane (32 bit) state doubles the number of register transpositions required in π step.

Second, doing the ρ step on multiple lanes of a plane in parallel requires multiple barrel shifter units in hardware which will be expensive in terms of gates. So we take an alternative approach and combine θ and ρ steps with π in a single instruction which requires one variable shifter unit in hardware. To accelerate the χ step we propose custom instructions that apply NOT, AND, XOR operation to nearby lanes packed in two 128 bit registers.

3.2 ARMv7 ISA and NEON SIMD

NEON instructions in ARM execute in a separate pipeline with its own register file [1]. The ARMv7 NEON unit has 16 quad-word (128 bit) registers, which are aliased with 32 double-word (64 bit) registers (Figure 5). In addition, the first 16 double-word registers are aliased with 32 single-word (32 bit) registers. NEON instructions 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 use a three-register format (2 source operands and 1 destination operand) or a format with 3 registers and an immediate value (VEXT instruction). Instructions also specify the vector alias format. For example, the `I32` specifier in `VADD.I32 q1, q2, q3` signifies that quad registers `q1`, `q2` and `q3` have 4x32-bit integer data.

4. PROPOSED INSTRUCTION SET EXTENSIONS FOR KECCAK

We propose a set of six custom instructions for KECCAK-$\{f, p\}$ $[1600, 800, 400, 200]$ primitives. Similar to other crypto-instructions (e.g. Intel AES-NI and SHA), our instructions take advantage of the wide SIMD registers. Not all of our instructions are SIMD in nature. They operate on quad, double or single word (scalar) registers depending upon the step and KECCAK permutation. Their shape and functionality is highly customized. In general, we optimized our instructions for the 1600 and 800 primitives, and we reuse the same instructions for implementing 400 and 200 primitives. The proposed instructions are of the register-to-register format, with two source operands and one destination operand. Operands are registers, and one of the source operands can also take an immediate argument. The proposed instructions are compatible with other NEON instructions and do not require special architectural features such as non-standard register files or lookup tables. Figure 6 shows the mapping of the three dimensional KECCAK state to the NEON registers. For the 1600 bit primitive, we map each lane to a double-

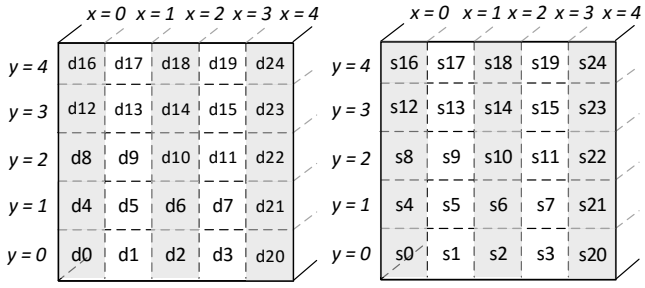

Figure 6: a) Register allocation for KECCAK-f[1600] and KECCAK-f[800,400,200]

word register D0-D24 and for the 800, 400, 200 bit primitives, we map each lane to single-word register S0-S24. The following sections are a discussion of each of the six proposed instructions. Unless specified, the examples assume the KECCAK-f[1600] permutation with 64 bit lanes.

4.1 Instruction `rl1x`

Instruction `rl1x` (*rotate left by 1 and XOR*) takes 2 registers as input (Figure 7 a), 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 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 1600, 800, 400, 200.

4.2 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. Figure 7 b 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).

Accommodating five bits as an immediate value in the instruction encoding is challenging. VEXT supports a four-bit immediate field but for supporting 25 different rotation offsets for 25 KECCAK lanes, at least 5 bits are needed. Since our instructions only support double registers, we used bit 6 of the instruction (used for defining Quad or Double register type) for encoding one extra bit of the immediate value. `kxorr64` is only used for implementing KECCAK-f,p[1600] permutations. For permutations of other sizes we provide a separate instruction `xorr`.

4.3 Instruction `xorr`

The functionality of `xorr` instruction is similar to `kxorr64` in Figure 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. Since, all 32 rotations for a 32 bit word can be encoded in 5 bit values, unlike `kxorr64`, `xorr` supports full range of rotation.

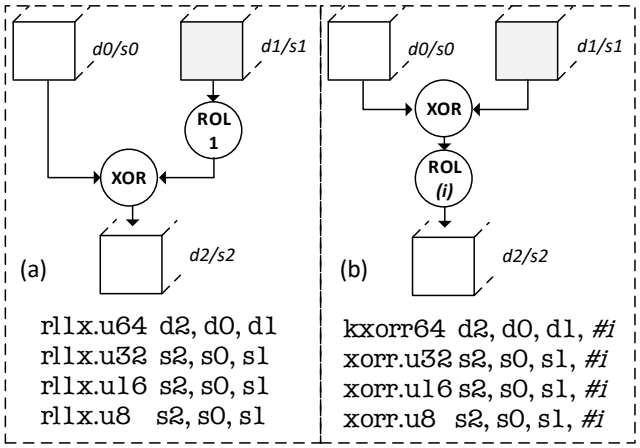

Figure 7: a) Functionality of `rl1x` instruction, b) Functionality of `kxorr64, xorr` instruction

4.4 Instruction `chi1`

`chi1` instruction aids χ step of KECCAK. `chi1` instruction accepts 4 lanes (in 2 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, `chi1` instruction also support multiple register views. A quad register can be viewed as 2×64 bit lanes (used in KECCAK-f[1600]) or 4×32 bit lanes (for KECCAK-f[800,400, 200]). The functionality of both the forms is shown in Figure 8 a.

4.5 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. Figure 8 b shows the functionality of `chi2`. `chi2` instruction accepts 2 quadword 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 KECCAKplane. The `chi2` instruction applies only to KECCAK-f,p[1600] permutations. For processing other variants of KECCAK, we have designed a `chi3` instruction.

4.6 Instruction `chi3`

Just like the `chi2` instruction, the `chi3` (Figure 8 c) instruction is also an auxiliary instruction that helps fixing the last lane of a plane without 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].

5. RESULTS

We used the open-source GEM5 simulator for implementation and benchmarking of the proposed custom instructions. We added

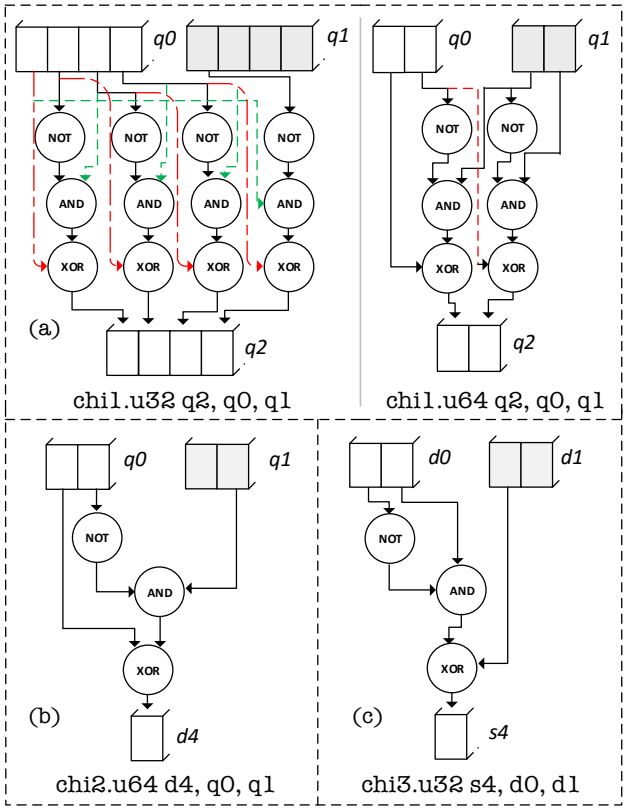

Figure 8: Functionality of instruction a) `chi1`, b) `chi2` and c) `chi3`

our instructions to the ARMv7 ISA part of GEM5 and described their functionality using GEM5's ISA description language. The NEON SIMD unit operates using a separate register file and pipeline. Instructions are sent to 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. 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. To get a reasonable estimate of the performance improvements we chose a simple timing CPU model available in GEM5. The timing simple CPU model executes instructions at a rate of 1 CPI (cycles per instruction) but takes the latency of memory system into account by adding stalls on cache accesses. We present our results in terms of dynamic instructions committed by the CPU normalized to input message size of the crypto algorithm.

5.1 Setup

We simulated a single core ARM CPU at 1 GHz with an L1 I and D cache of 32KB and L2 cache of 2MB. For generating the executable binaries we used cross-compiled GCC-4.9.2 and added the instruction encodings to ARM specific part of the GNU assembler. We followed the encoding format for NEON instructions and made sure that the instruction codes do not conflict with the existing NEON and ARM instructions. For comparison purposes, we used the optimized implementations of SHA3 (c = 1024), KEYAK and KETJE available in KECCAK code package [11]. We used Known Answer Tests to verify the functional correctness of SHA3(c=1024). For KEYAK and KETJE we used CAESAR testbench in the KECCAK code package.

5.2 Performance

Table 1 shows the results of our optimized assembly implementations using 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 with the setup discussed above. The simulation statistics were taken for the complete crypto application with an input block size of 10, 100 and 1000 blocks. 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 2 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. The expected speedup on a real hardware should also be very similar for three 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. Hence, KECCAK applications implemented using custom instructions should be able to achieve a CPI very close to 1 on a superscalar processor with low memory latency.

Table 1: Performance in instructions/byte for various KECCAK modes

Mode	C	32-bit ASM	NEON	CI†	Speed -up
SHA3(c=1024)	243.5	143.9	48.1	**21.9**	2.2
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

Table 2: Instructions executed per KECCAK round

Primitive	C	32-bit ASM	NEON	CI
KECCAK-f[1600]	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**

5.3 Hardware Cost

To get a reasonable estimate of 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 upto 2×128 bit inputs and a 5 bit immediate field and generate a 128 bit output. For synthesis, we used Synopsys Design Compiler that generates a netlist using UMC′s 90nm Process. The area estimates for each instruction have been provided in Table 3 with corresponding

gate equivalent (GE) and transistor equivalent counts. An ARMv7-A based quad-core processor[25] uses an area of $3.8mm^2$ in 28nm technology. On converting the area into GE [16], a single ARM core has around 3.8 million gates. Adding KECCAK instruction extensions will cost an extra 4658 gates, which adds an overhead of 0.1% while significantly improving the 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, 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.

Table 3: Area, GE and Transistor count estimates for the proposed custom instructions

Instruction	μ^2	Gate equiv.	Transistor equiv.
`rl1x`	1238	310	1238
`kxorr64`	7474	1869	7474
`xorr`	4884	1221	4884
`chi1`	3576	894	3576
`chi2`	966	242	966
`chi3`	486	122	486
Total	18624	4658	18624

6. PORTABILITY ASPECTS

The instruction design and results presented in this paper are for ARMv7 ISA with NEON SIMD. We will next describe how these instructions can be ported to other platforms. While we do not claim that the speedup and code footprint reduction will be same as shown in Section 5 for these other targets, we will explain how our instructions can improve KECCAK's performance on those platforms.

Table 4 shows the feasibility of instructions with respect to three different architectures, namely Intel AVX (128 bit SIMD), generic 64 bit platforms and 32 bit embedded platforms. This feasibility analysis is based on the number of operands supported by the ISA, the register width and the instruction encoding width.

Table 4: Feasibility of the proposed instructions on different platforms

Instruction	Intel AVX	64-bit Arch	32-bit Arch
`rl1x`	✓	✓*	✓*
`kxorr64`	✓	✓	✗
`xorr`	✓	✓	✓
`chi1`	✓	✗	✗
`chi2`	✓	✗	✗
`chi3`	✓	✓	✗

* Not all variants supported

6.1 Intel AVX (128 bit SIMD)

AVX[15] is the SIMD instruction set supported by Intel processors. It supports 128 bit wide registers and instructions with three operand, nondestructive format (`dest = src1 + src2`) like ARM NEON. AVX supports sixteen 128 bit XMM registers, which were later extended to 256 and 512 bits in AVX2 and AVX-512, with backward compatibility to 128 bit XMM registers. Since our design is targeted for 128 bit wide registers, we will only discuss the portability aspects of our instructions on 128 bit SIMD. The instruction design space will change with AVX2 and AVX-512 as a chunk of the KECCAK state can be stored in wider registers. Unlike NEON, AVX does not support aliasing of 128 bit registers to two 64 bit or four 32 bit registers . Since, some of the instructions like (`rl1x`, `xorr`) need access to an independent 64 bit or 32 bit scalar value inside a 128 bit register, we propose to encode scalar index in the instruction encoding. Intel Architectures support instruction encodings of up to 120 bits, so a 2 bit scalar index, for each register in the instruction can be encoded in the instruction encoding to apply the operation only to a particular scalar. Thus, all of the six instructions can be implemented on an Intel platform supporting AVX without any major architectural changes.

6.2 64-bit Architectures

Some of the instructions can also be implemented on 64 bit architectures like ARMv8, x64. All proposed instructions except `chi1` and `chi2` operate on 64/32 bit operands and produce a 64/32 bit result. Implementing `kxorr64` and `rl1x` instructions on 64 bit platforms can give a performance boost to KECCAK as they can accelerate the θ, ρ and π step. The `chi1 & chi2` instructions require 128 bit wide registers and cannot be supported on a 64 bit architecture, but alternatively χ step can be accelerated by supporting ternary instruction like `ternchi(dest = dest`$\oplus$`(`$\sim$`src1 & src2))` or `andn (dest =` $\sim$`src1 & src2)`. Smaller versions of KECCAK can also benefit from these instructions, if a w bit lane is stored in a 64 bit register and the word size for rotation 64, 32, 16 and 8 is passed to hardware as instruction encoding.

6.3 32-bit Architectures

For 32 bit embedded platforms, `rl1x` and `xorr` instructions can be very useful. All optimized 32 bit software implementations use bit-interleaving technique to break a bigger 64 bit lane to smaller 32 bit lane. These 32 bit lanes are used independently for rotations and other KECCAK steps. A 32 bit version of `rl1x` and `xorr` instruction can help accelerate the θ, ρ and π step on smaller embedded platforms. Like 64 bit, for 32 bit architectures also, `andn` or `ternchi` can be useful for accelerating χ step.

7. CONCLUSION

This paper presents a thorough analysis of the instruction set design space for KECCAK primitives. We have proposed a set of six custom instructions based on NEON instruction set in ARMv7 ISA, which can accelerate KECCAK-f and KECCAK-p primitives of size 1600, 800, 400 and 200 bits. To demonstrate the flexibility of our extensions we implemented five different KECCAK applications: SHA3, LAKE KEYAK, RIVER KEYAK, KETJE SR and KETJE JR and we analyzed the portability of the proposed instructions on Intel AVX and generic 64 and 32 bit platforms. To support our claims, we show performance improvements in instructions committed per byte of input data, using cycle accurate GEM5 simulator and provide hardware cost of the proposed extensions in GE using UMC's 90nm technology. We show that with the proposed extensions, KECCAK applications can achieve a performance gain between 1.4 - 2.6× over hand optimized assembly on ARMv7 at a hardware overhead of just 4658 GEs.

8. ACKNOWLEDGMENT

This research was supported in part through the National Science Foundation Grant 1441710, and in part through the Semiconductor Research Corporation.

9. REFERENCES

[1] ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition Issue C. online at infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406c/index.html, 2014.

[2] L. Batina, D. Hwang, A. Hodjat, B. Preneel, and I. Verbauwhede. Hardware/software co-design for hyperelliptic curve cryptography (HECC) on the 8051μp. In *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, pages 106–118, 2005.

[3] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, , and R. V. Keer. KECCAK implementation overview. online at http://keccak.noekeon.org/Keccak-implementation-3.2.pdf, May 2012.

[4] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge functions. Ecrypt Hash Workshop, May 2007.

[5] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge-Based Pseudo-Random Number Generators. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2010.

[6] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography (SAC)*, 2011.

[7] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the security of the keyed sponge construction. Symmetric Key Encryption Workshop (SKEW), February 2011.

[8] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. The KECCAK reference. online at http://keccak.noekeon.org/Keccak-reference-3.0.pdf, January 2011.

[9] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. CAESAR submission: Ketje v2. online at http://ketje.noekeon.org/Ketje-1.1.pdf, March 2014.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. CAESAR submission: Keyak v2. online at http://keyak.noekeon.org/Keyak-2.1.pdf, December 2015.

[11] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer. The Keccak Code Package. online at https://github.com/gvanas/KeccakCodePackage, 2016.

[12] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, et al. The gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[13] R. Buchty, N. Heintze, and D. Oliva. Cryptonite - A programmable crypto processor architecture for high-bandwidth applications. In *Organic and Pervasive Computing - ARCS 2004, International Conference on Architecture of Computing Systems, Augsburg, Germany, March 23-26, 2004, Proceedings*, pages 184–198, 2004.

[14] 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. *IACR Cryptology ePrint Archive*, 2012:50, 2012.

[15] Intel. Corporation. Intel 64 and IA-32 Architectures Software Developers Manual. online at http://download.intel.com/design/processor/manuals/253665.pdf, May 2011.

[16] Samsung. Corporation. Samsung Foundry 32/28nm Low-Power High-K Metal Gate Logic Process and Design Ecosystem. online at http://www.samsung.com/us/business/oem-solutions/pdfs/Foundry_32-28nm_Final_0311.pdf, March 2011.

[17] C. Demerjian. A long look at how ARM licenses chips. online at http://semiaccurate.com/2013/08/07/a-long-look-at-how-arm-licenses-chips/, August 2013.

[18] M. J. Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Federal Inf. Process. Stds. (NIST FIPS) - 202, August 2015.

[19] A. Frumusanu. Qualcomm Announces Snapdragon 625, 425 & 435 Mid- and Low-end SoCs. online at http://anandtech.com/show/10030/qualcomm-announces-snapdragon-625-425-435-mid-and-owend-socs, February 2016.

[20] B. Jungk and J. Apfelbeck. Area-efficient fpga implementations of the sha-3 finalists. In *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, pages 235–241, Nov 2011.

[21] E. Käsper and P. Schwabe. Faster and timing-attack resistant AES-GCM. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, pages 1–17, 2009.

[22] K. Sakiyama, L. Batina, B. Preneel, and I. Verbauwhede. Superscalar coprocessor for high-speed curve-based cryptography. In *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, pages 415–429, 2006.

[23] K. Seto and M. Fujita. Custom instruction generation with high-level synthesis. In *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 14–19, June 2008.

[24] A. L. Shimpi. Apple's Cyclone Microarchitecture Detailed. online at http://www.anandtech.com/show/7910/apples-cyclone-microarchitecture-detailed, March 2014.

[25] Y. Shin, K. Shin, P. Kenkare, R. Kashyap, H. J. Lee, et al. 28nm high- metal-gate heterogeneous quad-core cpus for high-performance and energy-efficient mobile application processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2013 IEEE International*, pages 154–155, Feb 2013.

[26] Y. Wang, Y. Shi, C. Wang, and Y. Ha. Fpga-based sha-3 acceleration on a 32-bit processor via instruction set extension. In *Electron Devices and Solid-State Circuits (EDSSC), 2015 IEEE International Conference on*, pages 305–308, June 2015.

[27] P. Yalla, E. Homsirikamol, and J. Kaps. Comparison of multi-purpose cores of keccak and AES. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE 2015, Grenoble, France, March 9-13, 2015*, pages 585–588, 2015.