Hardware/Software Co-Design of A Dynamically Configurable SHA-3 System-on-Chip (SoC)

Khaled E. Ahmed, Mohammed M. Farag
Electrical Engineering Department, Faculty of Engineering, Alexandria University, Alexandria, Egypt
Email: k.e.elsayed@ieee.org, mmorsy@alexu.edu.eg

***Abstract*—In this paper, we present a novel design of a dynamically configurable hardware accelerator for the new NIST SHA-3 standard, namely the Keccak hashing function. The SHA-3 accelerator is composed of a static datapath built based on two different folded architectures of the Keccak function and controlled by a programmable Finite State Machine (FSM) that can be dynamically configured at run-time to hash a message of arbitrary size and digest length. The proposed hardware architectures enable implementing all functions and modes of operation supported by the Keccak SHA-3 hashing standard. Two prototypes of the accelerator are developed and validated on a Xilinx Virtex-6 FPGA kit as a stand-alone system and on a ZedBoard kit featuring a ZynQ-7000 SoC FPGA, where the SHA-3 accelerator is implemented in the programmable logic and interfaced to an ARM Cortex-A9 processor. Hardware implementation is followed by a hardware/software co-design of a SHA-3 SoC running the keyed-Hash Message Authentication Code (HMAC) and Pseudo Random Number Generator (PRNG) security applications. The ARM processor runs the application software and offloads SHA-3 computations to the hardware accelerator. The implementation results illustrate the performance enhancement of the SHA-3 SoC over pure software implementations in addition to the unprecedented flexibility offered by the proposed accelerators.**

Keywords*—SHA-3 SoC, Keccak, Hardware Accelerator, HMAC, PRNG, Dynamic Configurable, FPGA, ZedBoard.

I. Introduction

Hash functions are mainly used for message authentication and are a major building block in other security protocols such as keyed-Hash Message Authentication Code (HMAC) [1] and Pseudo Random Number Generation (PRNG) [2]. In 2012, National Institute of Standards and Technology (NIST) has announced the selection of the Keccak function among five algorithm finalists to be the winner of the SHA-3 competition announced in 2007. The Keccak sponge function won the competition due to overcoming the security vulnerabilities of SHA-1 and SHA-2, its superior security and performance characteristics, and its suitability for hardware implementation [3].

As a sponge construction, the Keccak function depicted in Figure 1 accepts a binary message of an arbitrary length and computes a digest of a fixed size d bits. The number of inputs bits to a Keccak function is b bits. The message is divided into M blocks of r bits each, where r is called the rate. The security level of the function is indicated by the capacity $c = 2 \times d$ where $b = r + c$. A Keccak round is composed of a sequence of five operations called θ, ρ, π, χ and ι. The five operations operate in fixed sequence on a 5×5 W-bit state such that the lane width $W = b/25$. The functions θ, ρ, π, χ and ι perform successive permutations on the state by AND, logical rotate, and XOR operations as shown by Figure 2. The Keccak function is repeated for a number of rounds $N_R = 12 + 2 \times log_2 W$, on the same state, then another block of r-bits is XORed with the state and the N_R rounds are repeated. The Keccak function is specified by two parameters: b and c, where the notation Keccak-f[b](M,d) is the Keccak function on a message M with b-bits input and a digest size d [4]. There are four SHA-3 hash functions: SHA-3-224, SHA-3-256, SHA-3-384 and SHA-3-512 where SHA-3-d is Keccak-f[1600](M,d) or simply Keccak[c](M,d).

In this work, we propose two folded architectures for a Keccak hardware accelerator, namely Dynamic Configurable Capacity Keccak (DCC) and Dynamic Configurable Capacity and Lane width Keccak (DCCL). The DCC architecture enables implementing the functionality of any of the four hash functions Keccak-f[1600](M,d), thus changing the security level on demand. DCC can also implement the two SHA-3 variable length Extendable Output Functions (XOFs) SHAKE128 and SHAKE256 by adjusting the digest size and appending domain separation bits to the message to distinguish the XOFs from the four SHA-3 functions. The XOFs can be implemented directly from the Keccak-*f*, where SHAKE-c= $Keccak[c](M||11, d)$ and || denotes padding. The DCCL architecture enables changing both the capacity and the number of input bits b thus implementing any of the Keccak-f[b](M,d).

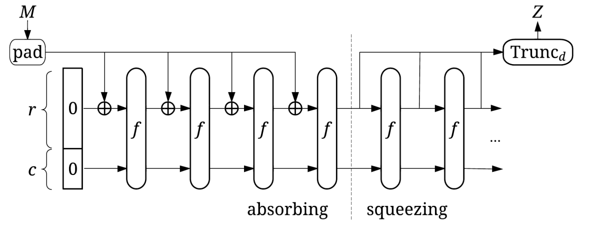

Fig. 1. Keccak sponge construction: Z = SPONGE[*f*, pad, r](*M, d*) [4].

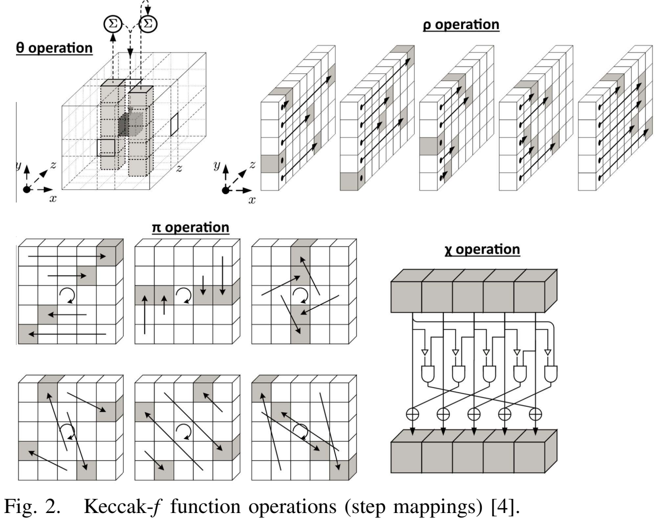

Fig. 2. Keccak-f function operations (step mappings) [4].

The Keccak hardware accelerator is interconnected as an I/O peripheral to a processor that dynamically controls the SHA-3 accelerator configuration at run-time according to the application requirements where the trade-off is always security versus latency. Both the DCC and DDCL are implemented using the Xilinx Vivado High-Level Synthesis (HLS) design flow. The HLS flow allows early exploration of different architectural alternatives and optimizations such as loop unrolling and pipelining using simple compiler directives added to the same source code. HLS enables quick implementation and validation of the developed accelerator in addition to its easy integration to the full system using the AXI interface protocol.

An intensive research effort was devoted prior to the NIST's selection of Keccak to evaluate the SHA-3 competition finalists. The premier two reasons for choosing Keccak to implement SHA-3 are security and performance. A significant amount of cryptanalysis was applied to Keccak and the main conclusion is that Keccak has a huge security margin qualifying it to be the hashing standard for the next decade [3]. The software performance of the candidates was benchmarked on general purpose processors and embedded microcontrollers, and Keccak has achieved reasonable speed [3]. The hardware implementation results from various sources were a major pro for Keccak. ASIC and FPGA implementations were introduced for all the finalists, multiple architectural alternatives were evaluated, Keccak has the highest throughput/area ratio and it is the only candidate that gives better results than SHA-2 [3].

In [5], an efficient implementation of the Keccak-*f*[512] and Keccak-*f*[1024] functions is presented, regular RTL practices and HDL coding techniques are used to optimize the design. A high-speed Keccak-*f*[512] is implemented in [6] in which all lane operations inside the state are executed in parallel. Keccak-*f*[512] is compared to SHA-2-256 and demonstrates higher throughput per area ratio. An extremely compact Keccak-*f*[512] is presented in [7] for lightweight applications. The small area is achieved by iterating the operations on the lane, thus imposing long latency. In [8], an efficient low-area implementation was presented for each of the five candidates. For the SHA-3 function, an iterative folded architecture was built to minimize the area. The throughput is reduced by folding the architecture as well, but this implementation achieves high throughput to area ratio. In [9], the five nominees for the SHA-3 where implemented using both the traditional HDL flow and the HLS flow. A comparison was drawn between both approaches, although the HLS approach imposes a noticeable overhead on the design, the ranking of the HLS implemented nominees was the same as that for the traditional HDL design flow.

In this paper, we advance the first implementation, to the best of our knowledge, of a dynamically configurable SHA-3 System-on-Chip (SoC) using the Xilinx Vivado HLS design flow. The rest of this paper is organized as follows, the architecture of the SHA-3 accelerator is illustrated in Section II. In section III, we compare the performance of our system to the relevant results reported in the literature. Conclusions and future work directions are portrayed in Section IV.

II. Dynamic Configurable SHA-3 Architecture

We advance two folded architectures for the Keccak-*f* function. Both architectures are implemented using the Vivado

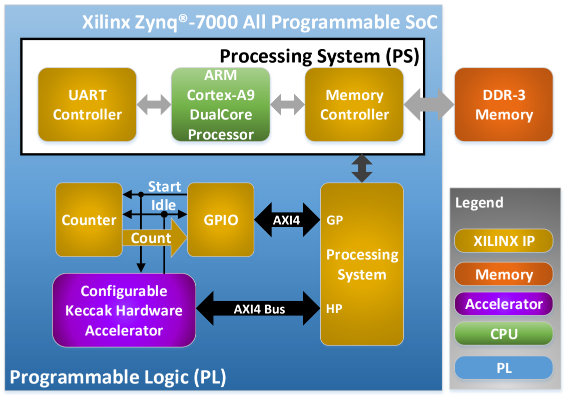

Fig. 3. Dynamic configurable SHA-3 SoC.

HLS flow and interfaced to an ARM processor in the SoC illustrated in Figure 3. The developed SoC is hosted and tested on a ZynQ-7000-based ZedBoard FPGA kit. The ZynQ-7000 architecture comprises: a programmable system (PS) featuring an ARM Cortex-A9 dual-core processor and its I/O peripherals as software counters and UARTs; and programmable logic (PL) which is configured to implement custom hardware modules. The SHA-3 system is composed of five main components which are the Keccak Accelerator, an ARM processor, a DDR-3 memory, an AXI bus, and a counter.

- **ARM Processor** runs the software application needing the hash functionality and controls the Keccak hardware accelerator. The interaction between the processor and the accelerator is done via the memory. The processor places a data header in a specific memory location that contains necessary information for the accelerator to run. The information contained in the header is the memory address of the message to be hashed, the message size, the memory address to place the digest, and the configuration data which is the capacity for the DCC architecture and the capacity and lane width for the DCCL architecture. After the header is placed, the ARM initializes the accelerator operation via asserting a *Start* signal and waits for the accelerator to assert the *Idle* signal which indicates the operation completeness.
- **Counter**: It is cleared when the ARM issues the start signal to the accelerator and starts counting clock cycles while the *Idle* signal is deasserted. When the *Idle* signal goes high, the counter pauses and the ARM can read the accelerator latency from the counter for performance evaluation.
- **SHA-3 Accelerator**: Both DCC and DCCL architectures are implemented in HLS and only one accelerator is interfaced to the processor. When the *Start* signal from the processor is asserted the accelerator (i) initially reads the header from the memory; (ii) reads the message starting from the address provided in the header where a block of r bits is read each time; (iii) performs the hash function on the fetched blocks until the entire message is hashed; and (iv) writes back the computed digest to the memory address given by the header.

In the DCC-SHA-3 architecture shown in Figure 4, the software sets the capacity of the Keccak sponge function to hash a message of an arbitrary size and desired hash length with a minimum capacity of $c = 448$ bits. Therefore, the DCC-SHA-3 accelerator can be configured as any of the four Keccak-*f*[1600] functions enabling the software designer to trade off security for latency on the fly according to the application requirements. The configuration data is transferred

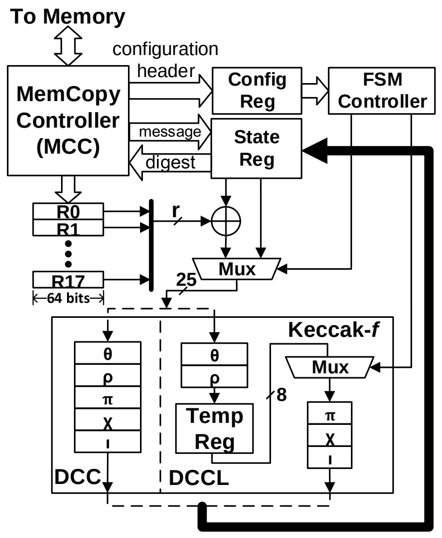

Fig. 4. High-level architecture of the DCC and DCCL Keccak accelerators.

to a *ConfigReg* controlling a programmable *FSM controller*. The *StateReg* is a set of 25 64-bit registers storing data needed by the round state. At the beginning of each round, the Memory Copy Controller *(MCC)* block copies r bits from the main memory, where $r = 1600 - c$. Each 8 bytes are placed in one of the 18 64-bit registers *R* which are designed to accommodate the largest possible rate $r = 1152$ corresponding to the smallest possible capacity $c = 448$. The r bits are then XORed with the *state* and passed to the sponge function *Keccak-f*. Each clock cycle, the sponge function is executed on the state and the result is fed back to the function for the next round. After 24 rounds, the d-bit digest is written back from the *StateReg* to *MCC*. The *Keccak-f* and *MCC* modules are pipelined, thus the *MCC* block can continue saving the message into the registers while the sponge is being executed.

In the DCCL architecture depicted in Figure 4, both the lane width and capacity of the Keccak function can be configured. The lane width can be configured as a multiple of 8 bits, therefore all Keccak functions from Keccak[200] up to Keccak[1600] can be implemented by simply modifying the message header. The θ and ρ functions include rotation operations so they must accommodate the maximum possible lane width of 64 bits to prevent overflow. The resultant state form θ and ρ is stored into a *TempReg*. The width of π, χ, and ι modules is set to one byte, and the folded datapath is used iteratively to operate on the W-bit state to save area at the expense of throughput reduction. For instance, for $W = 64$, the 8-bit datapath needs 8 cycles to complete the operation.

A. Hardware/Software Co-Design of SHA-3 Applications

Hash functions are mainly used for message authentication, but they are also widely used for other security protocols such as HMAC and PRNG. HMAC is used to simultaneously verify both data integrity and message authenticity and involves a cryptographic hash function such as SHA-3 in conjunction with a secret key. The HMAC standard function developed by NIST in [10] is computed using the following equation:

$$\begin{aligned} MAC(text) &= HMAC(K, text) \\ &= H((K_0 \oplus opad)||H((K_0 \oplus ipad)||text)) \end{aligned}$$

where K_0 is the secret key, $opad$, $ipad$ are constant pads and H is a hash function.

PRNG is an algorithm for generating unpredictable random numbers based on a given seed. PRNGs are central in many security and non-security applications. Good statistical properties are an essential requirement for the PRNG output which must not be predictable from previous outputs. Because they satisfy this requirement, hash functions are a basic building block in a widely-used class of PRNGs. We implemented the PRNG algorithm presented in [2]. The algorithm is composed of two steps: $feed(\sigma)$ and $fetch(l)$. $feed(\sigma)$ feeds the Keccak with the seed σ (message) by splitting the seed into N r-bit blocks and feeding them in sequence to the sponge function as shown in Figure 1. $fetch(l)$ squeezes l bits from the Keccak by reapplying the sponge on the state without feeding any seed and reading l bits as the output from the state. It is clear that HMAC and PRNG functions mainly rely on the SHA-3 hash function which forms the computational intensive part of both applications. Therefore, we partition the SHA-3 SoC shown in Figure 3 into the dynamic configurable hardware accelerator running the SHA-3 hash function and a software program on the processor running the remaining application tasks such as padding and other logical operations.

III. System Implementation and Evaluation

The throughput obtained from the DCC and DCCL architectures can be calculated using the following equation:

$$Th = \frac{b}{T_c * N} \tag{1}$$

where Th is the throughput in Mbps, b is the number of input bits, T_c is the clock period and N is the number of cycles required to hash a single block. For both architecture, the number of cycles N required to hash a block of size b is:

$$N = \max(N_{MC}, N_S) \tag{2}$$

where N_{MC} is the number of cycles required to copy b bits from memory, and N_S is the number of clock cycles required for the sponge function computation which equals to 25 cycles.

Burst access is used to copy data to and from the DDR-3 memory, thus N_{MC} required to copy b bits is computed as:

$$N_{MC} = 9 + b/64$$

where 9 cycles are required to initiate a transaction to the memory through the AXI bus, then one clock cycle is required to copy each 64 bits of data. Therefore, the latency ranges from 25 to N_{MC} according to the configurable capacity c.

The number of iterations executed in the DCCL depends on the width W, and it is independent of the rate and capacity. The rate only changes the number of bytes copied into the internal buffer before execution. Therefore the bandwidth for a given W can be calculated using Eqs 1 and 2 given that:

$$\begin{aligned} N_S &= N_R \times (3 + W/8) \qquad (3) \\ &= (12 + 2 \times log_2 W) \times (3 + W/8) \qquad (4) \end{aligned}$$

where the number of cycles per round $(3 + W/8)$ is decomposed into one cycle for both θ and ρ, one cycle for extracting the input byte to π and the following functions, one cycle to merge back the lane, and one cycle per byte in the lane.

The DCC and DCCL architectures for Keccak-*f*[512] are implemented and validated on a Xilinx Zedboard and Virtex-6

TABLE I. HARDWARE IMPLEMENTATION RESULTS OF KECCAK-f[512]

Architecture	Platform	Flow	Slices	MHz	MBits/s	$\frac{MBits/s}{Slice}$
DCC SA	Virtex 6	HLS	1,541	182	4,992	3.23
DCCL SA	Virtex 6	HLS	1,888	175	424	0.22
DCC SoC	Zedboard	HLS	2,891	110.74	1,676	0.57
DCCL SoC	Zedboard	HLS	2,923	104	256	0.08
B. Jungk [8]	Virtex 6	HDL	397	197	1,071	2.69
N. Moreira et al. [1]	Virtex 5	HDL	1,062	317	13,530	12.74
E. Homsirikamol [9]	Virtex 6	HLS	1,494	211.2	8,838	5.92
J. Kaps et al. [7]	Virtex 6	HDL	106	-	136	1.28
K. Latif et al. [5]	Virtex 6	HDL	915	301.5	13,670	14.94
T. Honda et al. [6]	Virtex 6	HDL	1,181	251.7	5,660	4.79

FPGA kit. The Zedboard is selected due to the availability of an embedded processor enabling SoC design. The DCC and DCCL architectures are implemented as a stand-alone (SA) system on a Virtex-6 FPGA kit, where most relevant results from the literature are based on this kit. The area utilization in slices, maximum obtainable frequency in Mhz, throughput in Mbps, and throughput to area ratio in Mbps/slice are displayed in Table I for different SHA-3 implementations. In our DCC and DCCL SHA-3 SoC accelerators, the storage and buffering resources increase the resource utilization compared to the stand-alone implementation. The low throughput to area ratio of the DCCL compared to DCC is attributed to the increased control overhead to fold Keccak operations which increases the resource utilization, in addition to increasing the computation time by $W/8$ clock cycles per round which reduces the throughput. Comparing our HLS implementation of the SA DCC to other implementations shows that HDL-based design can achieve much better throughput to resource ratio as stated by [1], [5] compared to the HLS-based flow which adds a significant control overhead degrading both resource utilization and throughput metrics. The price paid for adding the reconfiguration capability to our Keccak accelerators is reducing resource utilization to throughput ratio.

To evaluate the performance of the configurable hash SoC, we implemented the HMAC and PRNG algorithms using two approaches: pure software and mixed hardware/software co-design. The SHA-3 SoC depicted in Figure 3 is realized on a ZynQ-7000 SoC FPGA incorporating an ARM processor. The pure software implementation of the algorithm runs on the processor and calls a software-implemented Keccak function whenever it is required in the algorithm. In the hardware/software implementation, whenever the hash function is called, the processor passes the configuration header and message to the Keccak hardware accelerator and the processor waits for the accelerator to compute the message digest. Figure 5 shows the execution time in ms of the PRNG and HMAC for both the software and hardware/software implementations for various message sizes. Both DCC and DCCL are used to implement the hardware Keccak accelerator, the software configures the accelerator to implement the Keccak-f[512] function. The HMAC algorithm uses a 256-bit key with message sizes ranging from 256 to 1024 bytes. The PRNG algorithm implements two $feed(\sigma)$ and one $fetch(l)$ requests on a seed of size ranging from 256 to 1024 bytes. The comparison shows the latency enhancement achieved by the DCC and DCCL SoC over the direct software design.

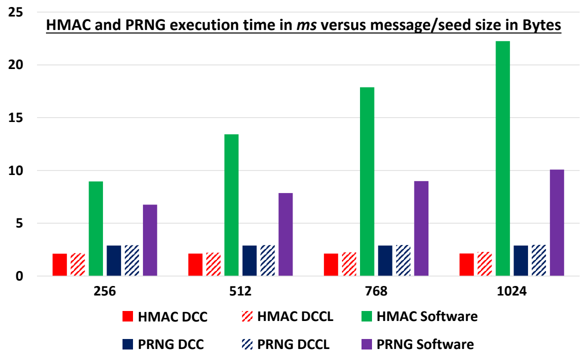

Fig. 5. HMAC and PRNG execution time for different message/seed sizes.

IV. CONCLUSIONS

In this paper we advanced two novel architectures for the SHA-3 hardware accelerator; a capacity configurable architecture and a capacity and lane-width configurable architecture. The SHA-3 dynamic configurable accelerator comprises a static hardware block implemented based on a folded architecture of the Keccak-f function and controlled by a programmable FSM. The dynamic configuration feature enables the accelerators to implement any of the Keccak-f functions, any of the four SHA-3 functions, and the two XOF functions. The two accelerators are implemented and validated as a stand-alone system on a Virtex-6 FPGA and as an integrated SoC on a ZynQ-7000 FPGA incorporating an ARM Cortex-A9 processor. The HMAC and PRNG security functions are used as a typical hardware/software co-design application of the SHA-3 SoC. As a future work, we will investigate architectural optimizations of the SHA-3 accelerators using the HDL-based design flow to improve the performance metrics.

REFERENCES

[1] Naiara Moreira, Armando Astarloa, and Uli Kretzschmar. SHA-3 based message authentication codes to secure IEEE 1588 synchronization systems. In *Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE*, pages 2323–2328. IEEE, 2013.

[2] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge-based pseudo-random number generators. In *Cryptographic Hardware and Embedded Systems, CHES*, pages 33–47. Springer, 2010.

[3] Shu-jen Chang, Ray Perlner, William E Burr, Meltem Sönmez Turan, John M Kelsey, Souradyuti Paul, and Lawrence E Bassham. *Third-round report of the SHA-3 cryptographic hash algorithm competition*. US Department of Commerce, NIST, 2012.

[4] NF Pub. DRAFT FIPS PUB 202: SHA-3 standard: Permutation-based hash and extendable-output functions. *FIPS Publication*, 2014.

[5] Kashif Latif, M Muzaffar Rao, Arshad Aziz, and Athar Mahboob. Efficient hardware implementations and hardware performance evaluation of SHA-3 finalists. In *The Third SHA-3 Candidate Conference*, 2012.

[6] T Honda, H Guntur, and A Satoh. FPGA implementation of new standard hash function keccak. In *Consumer Electronics (GCCE), 2014 IEEE 3rd Global Conference on*, pages 275–279. IEEE, 2014.

[7] Jens-Peter Kaps, Panasayya Yalla, Kishore Kumar Surapathi, Bilal Habib, Susheel Vadlamudi, and Smriti Gurung. Lightweight implementations of SHA-3 finalists on FPGAs. In *The Third SHA-3 Candidate Conference*, 2012.

[8] Bernhard Jungk. Evaluation of compact FPGA implementations for all SHA-3 finalists. In *The Third SHA-3 Candidate Conference*, 2012.

[9] Ekawat Homsirikamol and Kris Gaj. Hardware benchmarking of cryptographic algorithms using high-level synthesis tools: The SHA-3 contest case study. In *Applied Reconfigurable Computing*, pages 217–228. Springer, 2015.

[10] NIST FIPS. 198: The keyed-hash message authentication code (HMAC). *National Institute of Standards and Technology, Federal Information Processing Standards*, page 29, 2002.