FPGA-based SHA-3 Acceleration on a 32-bit Processor via Instruction Set Extension

Yi Wang
Institute for Infocomm Research
(I^2R),A*STAR, Singapore
wangy@i2r.a-star.edu.sg

Youhua Shi
Waseda University
Tokyo, Japan
shi@aoni.waseda.jp

Chao Wang
Institute of Microelectronics
(IME), A*STAR, Singapore
wangc@ime.a-star.edu.sg

Yajun Ha
Institute for Infocomm Research
(I^2R),*STAR, Singapore
hay@i2r.a-star.edu.sg

Abstract*—As embedded systems play more and more important roles Internet of Things (IoT), the integration of cryptographic functionalities is an urgent demand to ensure data and information security. Recently, KECCAK was declared as the winner of the third generation of Secure Hashing Algorithm (SHA-3). However, implementing SHA-3 on a specific 32-bit processor failed to meet the performance requirement. On the other hand, implementing it as a cryptographic coprocessor consumes a lot of extra area and requires customized driver program. Although implementing KECCAK on a 64-bit platform is more efficient, this platform is not suitable for embedded implementation. In this paper, we propose a novel SHA-3 implementation using instruction set extension based on a 32-bit LEON3 processor (an open source processor), with the goals of reducing execution cycles and code size. Experimental results show that the proposed design reduces around 87% execution cycles and 10.5% code size as compared to reference designs. Our design takes up only 9.44% extra area with negligible speed overhead compared to the standard LEON3 processor. Compared to the existing hardware accelerators, our proposed design occupies only half of area resources and does not require extra driver programs to be developed when integrated into the overall system.

I. INTRODUCTION

Embedded security attracts more and more attention with increasing requirements for secure communication in Internet of Things (IoTs). However, this extra security requirement becomes a huge burden for these resource-constrained devices, which are limited in area, memory and power. Traditional solutions include developing extra software programs to support cryptographic primitives and implementing these primitives by a dedicated coprocessor. The drawback of the first solution is the degradation of performance due to reusing the general purpose instruction set to support specific cryptographic functions. On the other hand, the drawbacks of the second solution are lack of flexibility, requiring extra driver developments and the limited portability to fulfill the needs in the future. Alternatively, we could partition the computation intensive part of a design into hardware, which is supported by the customized instruction. Finally, we need integrate these instructions into a general purpose processor. This is called Application Specific Instruction Set Processor (ASIP).

ASIP technique is commonly used to extend the general purpose instruction set with customized instructions. It can accelerate the targeted design's performance by identifying computation intensive part of an application and employing customized hardware components to substitute this part. Accelerating Advanced Encryption Standard (AES) [1] using ASIP technique has been fully studied in numerous works [2] [3] [4] [6]. Good and Benaissa implemented AES on a very small FPGA application-specific instruction processor [2]. They used PicoBlaze (KCPSM3) as the main processor and the final customized design could compute the AES encryption and decryption by 365 instructions within 13,546 cycles and 18,885 cycles, respectively. Another work was proposed by Tillich and Groβsch$\ddot{a}$dl [3], which was based on the LEON2 processor. They reused the instructions extended for Elliptic Curve Cryptography (ECC) to support AES computations. Based on that work, they also proposed the instruction set extension for the 128-bit AES design [4] on the same processor. Their design took up 704 cycles for performing the AES encryption. Our recent work also addressed that the performance could be improved when applied to higher-order masked AES using ASIP [6] (detailed masked AES can be found in [7]). Besides the acceleration for AES algorithm, Grabher et al. proposed a light-weight instruction set extension for bit-sliced cryptographic algorithms such as DES, Serpent, AES, PRESENT and SHA-1 [5].

However, accelerating the performance of newly announced SHA-3 algorithm using ASIP has not been explored so far. Since the winner of SHA-3 was selected, some researchers worked on how to implement KECCAK efficiently on both software and hardware platforms. Gouiem [10] compared the software implementation on Intel 8051 and ARM7 TDMI. The total execution cycles of his design on these two platforms are 237,352 and 50,834, respectively. Homsirikamol et al. optimized KECCAK-256 and KECCAK-512 on various FPGA platforms [11] [12]. They compared the implementation results of the above algorithms in detail. Their design achieved the highest throughput among the existing works. Baldwin et al. implemneted KECCAK-256 which achieved 8,518 Mbps throughput using 1,117 slices on Xilinx xc5vlx330 [13]. S. Matsuo et al. provided a fair and consistent hardware evaluation for SHA-3 candidate. They proposed a unified Input/Output interface to evaluate SHA-3 candidates [14]. Among them, KECCAK-256 took up 1,411 slices with the throughput of 8,397 Mbps on Xilinx Virtex-5 FPGA.

Contributions: in this paper, we adopt ASIP technique to support KECCAK algorithm, which is the winner of the third generation for Secure Hash Algorithm (SHA-3) [8]. The targeted platform selected in this paper is LEON3 processor, which is an open source processor provided by Gaisler Research AB [9] . Its performance is similar to the standard commercial processors, such as ARM® and MIPS®. We propose to improve the performance of KECCAK algorithm using customized instructions with the goals of reducing execution cycles and code size without compromising the original performance of the LEON3 processor. In order to achieve this, we analyze the computation intensive part of SHA-3. Then, we partition this computation intensive part, bit rotations of consecutive multiple words, into hardware. Finally, we adjust the original execution unit of 32-bit LEON3 processor to be able to process 64-bit data operations. The proposed design can support up to four lengths (224-bit, 256-bit, 384-bit and 512-bit) of KECCAK. The experimental results show that the critical path of the proposed design is nearly the same as the original LEON3 processor. Around 87% reduction of execution cycles is achieved compared to the reference design without instruction set extension.

The rest of the paper is organized as follows. Section II introduces the KECCAK algorithm. Section III proposes the automatic design flow for instruction set extension on LEON3 processor and implements the KECCAK algorithm using the cutomized instruction. Section IV gives the experimental results. Section V draws the conclusion.

II. BACKGROUND OF KECCAK ALGORITHM

KECCAK has been selected as the final winner of SHA-3 competition [8]. It is based on the sponge construction which has a very short critical path for reaching very high frequencies. It consists of three phases: padding, absorbing and squeezing phases. Algorithm 1 shows the procedure of sponge function of SHA-3, where c represents parameter capacity, r represents bit rate and d represents diversifier. First, message M is padded with parameter d and r. Then, the output result of padding is noted as P, where $P = \{P_0, P_1, \cdots, P_i\}, i \in \{0, 1, \cdots, n_r\}$ (n_r is defined as follows). The absorbing phase includes two operations, Xoring with the padded value and performing KECCAK$-f[b]$ function. Finally, the squeezing phase output the result, Z. In this algorithm, KECCAK$-f[b]$ function is the permutation of KECCAK. It is also the main operation in KECCAK algorithm.

KECCAK$-f[b]$ is the permutation function ($b \in \{25, 50, 100, 200, 400, 800, 1600\}$), where $b = 25w$ ($w \in \{1, 2, 4, 8, 16, 32, 64\}$). The number of rounds depends on the width of permutation as $n_r = 12 + 2l$ ($2^l = \frac{b}{25}$). In this paper, we select KECCAK$-f[1600]$ (stated in the SHA-3 standard proposal) as the targeted algorithm. Algorithm 2 shows the detailed operation steps for KECCAK$-f$, where C, D and B are the intermediate results. $r[x, y]$ is the coefficient of each round. It consists of five steps. They are θ step, ρ step, π step, χ step and ι step.

From Algorithm 2, it is obvious that the main operations of KECCAK algorithm consist of XOR, ROT$[x, i]$, NOT and AND functions. ROT$[x, i]$ function will perform a right rotating shift of x, where x is two consecutive integers in KECCAK algorithm.

III. PROPOSED SHA-3 IMPLEMENTATION USING INSTRUCTION SET EXTENSION

As discussed before, we focus on KECCAK$-f[1600]$ algorithm in this paper because it is recommended by the standard proposal. The parameters of this algorithm are defined as : $b = 1600$, $l = 5$ and $n_r = 24$. First, we need port 64-bit program to 32-bit software platform to suit 32-bit LEON3 processor. Bit interleaving technique is used to represent one 64-bit value (x) with two consecutive 32-bit values ($x.h$ and $x.l$). The operations in θ step, ρ step, π step, χ step and ι step are mainly Xoring, ROT$[x, i]$, NOT and AND. Bit interleaving of a 64-bit data does not affect the Xoring, NOT and AND operations. However, it will affect ROT$[x, i]$ function (a right rotating shift of x) because x is two consecutive 32-bit integers in this design. It is obvious that performing right rotating shift of a 64-bit integer represented as two consecutive 32-bit integers will require two right shift operation and two splicing operations of two 32-bit integers on the 32-bit processor. This will become the main bottleneck on the LEON3 processor when implementing KECCAK. We will propose to reuse the existing datapath of LEON3 processor to support this 64-bit

Algorithm 1 KECCAK sponge function

```
Input: KECCAK$[r, c, d](M)$, $M$ is the message to be hashed
Output: $Z$
    Initialization and Padding
    $S[x, y] = 0$, $\forall(x, y)$ in $(0 \cdots 4, 0 \cdots 4)$
    $P = M||$0x01$||$byte$(d)||$byte$(\frac{r}{8})||$0x01$||$0x00$|| \cdots ||$0x00
    Absorbing phase
    For every block $P_i$ in $P$
    $S[x, y]$=$S[x, y] \oplus P_i[x + 5y]$, $\forall(x, y)$
    (such that $x + 5y < \frac{r}{w}$)
    S= KECCAK$-f[r + c](S)$
    Squeezing phase
    $Z$ = empty string
    while output is requested
    $Z = Z||S[x, y]\forall(x, y)$, such that $x + 5y < \frac{r}{w}$
    S= KECCAK$-f[r + c](S)$
    return $Z$
```

Algorithm 2 KECCAK$-f$ function

```
Input: KECCAK$[b](S)$, $S$ is the processed value in
    Algorithm 1
Output: $S$
    for $i$ in $0 \cdots n_r - 1$
    $S$ = Round$[b](A, RC[i])$, $RC$ is the constant values for each
    round
    return $S$
    Where Round$[b](S, RC[i])$ consists of the following five
    functions, $\forall x$ in $0 \cdots 4$ and$\forall y$ in $(0 \cdots 4)$ .
    (1) $\theta$ step
    $C[x] = S[x, 0] \oplus S[x, 1] \oplus S[x, 2] \oplus S[x, 3] \oplus S[x, 4]$
    $D[x] = C[x - 1] \oplus$ ROT$(C[x + 1], 1))$
    $S[x, y]$=$S[x, y] \oplus D[x]$
    (2) $\rho$ and (3)$\pi$ step
    $B[y, 2x + 3y]$ = ROT$(S[x, y], r[x, y])$
    (4) $\chi$ step
    $S[x, y] = B[x, y] \oplus (($NOT$B[x + 1, y])$AND$B[x + 2, y])$
    (5)$\iota$ step
    $S[0, 0] = S[0, 0] \oplus RC[i]$
```

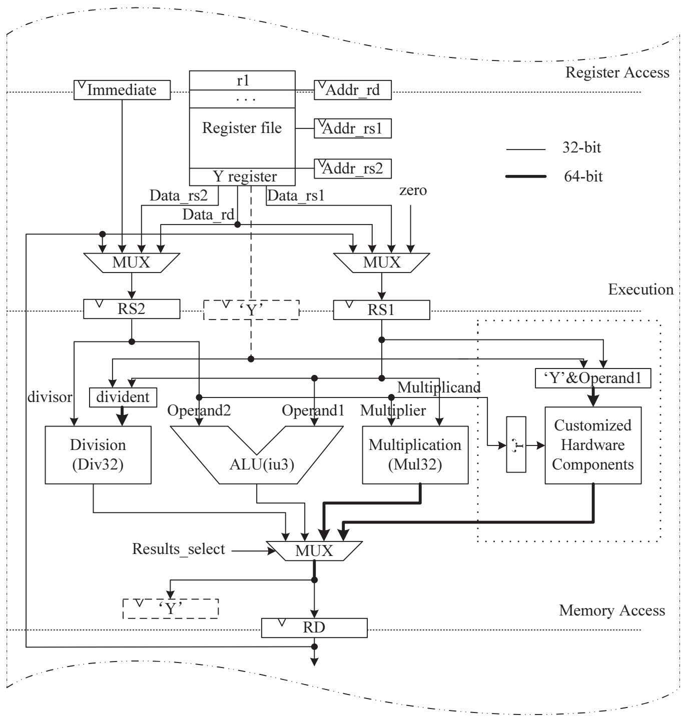

Fig. 1. Hardware architecture for the extended datapath for KECCAK

operation in the following.

Since a 32-bit architecture supports at most two 32-bit register operations, such as 32-bit multiplication and 32-bit division, this provides the possibility to support bit rotations of two 32-bit integers in our design. LEON3 processor supports up to seven pipelined stages: fetch, decode, register access, execute, memory access, exception and write back. Fig. 1 shows the proposed customized hardware component integrated at the execution stage of the LEON3 processor. In Fig. 1, it consists of register access, execution and memory access stages, which includes register file, ALU, multiplication, division units. At the register access stage, the controller sends the addresses (Addr_rs1, Addr_rs2, Addr_rd) of source register 1, source register 2 and destination register to the register file unit. Then, the register file transfers corresponding data values (Data_rs1, Data_rs2, Data_rd) to source registers, RS1 and RS2, respectively. The immediate value can be directly sent to RS2. At the execution stage, RS1 and RS2 send operand1 and operand2 to ALU and multiplication, respectively. For division, 'Y' register is used to store high 32-bit of dividend and RS1 is used to store low 32-bit of dividend. At the memory access stage, the output of ALU and division is 32-bit, while the output of multiplication is 64-bit, in which higher 32-bit is stored at 'Y' register and the lower 32-bit is stored at destination register (RD).

The proposed design shares the same input as the division (div32), in which the 64-bit word are concatenated by the 'Y' register and RS1. The number of bit shifting is indicated by r stored in RS2. The final 64-bit output of the proposed hardware component is stored at the 'Y' register and the destination register, which shares the same output as the 32-bit multiplication (mul32).

As a result, one extra instruction is required to perform bit rotations of two 32-bit integers. It can perform r bits rotation within one clock cycle. Note here, we need to store higher 32-bit operand to the 'Y' register before performing this instruction. We also need to retrieve higher 32-bit result from Y register after executing this instruction.

IV. Experimental Results

In this section, we have implemented our proposed designs on LEON3 processor. It has been ported to Xilinx ML605 evaluation platform (Virtex-6 XC6VLX240T FPGA chip with the supported peripherals). We use Xilinx ISE 13.4 tools to synthesize the original LEON3 processor with the customized hardware components, which have been implemented using a hardware description language (VHDL). In order to properly compile the customized design, the original BCC compiler [9] has been modified to output the binary executable code. Except the two tools above, we use GRMON debug tool [9] to perform inline debugging and to verify the correctness of the proposed design.

The targeted goal in this paper is to reduce the required execution cycles and code size of KECCAK algorithm when porting it to a 32-bit embedded processor. In order to count the executing cycles, we use the timer of LEON3 processor, which counts the required cycles to run the proposed design. We also re-implemented KECCAK-224, KECCAK-256, KECCAK-384 and KECCAK-512 on LEON3 processor using C program, called "reference design based on LEON3" in this paper.

Table I shows experimental results between the reference designs based on LEON3 and the proposed designs. The counting point of starting the timer is set to start KECCAK$-f[1600]$ function, since padding periods might be different with different lengths of messages. Our proposed design achieved 87.47% to 87.65% reduction in execution cycles and 10.5% reduction in code size compared to the reference designs.

Till date, there have no designs about integrating KECCAK algorithm to the embedded system based on the LEON3 processor. Therefore, it is hard to directly compare our experimental results with the existing approaches. Although hardware accelerators are able to achieve high throughput, they have the shortcomings of occupying extra hardware resources and requiring integration effort. It needs extra effort to develop driver programs for customized designs. In order to have a fair comparison, we make an assumption that the LEON3 processor is the basic platform to be used for a secure embedded application, in which the hardware implementation of KECCAK is the additional security feature required by the system. Table II shows comparisons between the existing hardware accelerator designs and our proposed designs. It also shows the comparison between the original LEON3 processor and the proposed designs. Compared to the original LEON3 processor, our proposed designs take up 9.4% slices/10.6% LUTs more, but it is with a slight effect on the critical path of the original LEON3 processor (only 0.069% overhead in the critical path).

TABLE I
COMPARISONS BETWEEN THE REFERENCE DESIGNS BASED ON LEON3 AND THE PROPOSED DESIGNS

Reference designs	Execution	Code Size(KBytes)		ASIP designs	Execution	Code Size (KBytes)		Enhancement
based on LEON3	Cycles	ROM	RAM		Cycles	ROM	RAM	Cycles
KECCAK-224	188444	17.4	2.6	KECCAK-224	100435	15.6	2.5	87.63%
KECCAK-256	188485	17.4	2.6	KECCAK-256	100444	15.6	2.5	87.65%
KECCAK-384	188599	17.4	2.6	KECCAK-384	100551	15.6	2.5	87.57%
KECCAK-512	188708	17.4	2.6	KECCAK-512	100663	15.6	2.5	87.47%

TABLE II
COMPARISONS WITH THE EXISTING HARDWARE DESIGNS

	Designs	Freqency	Area	TP	Platform
		(MHz)	(Slices)	(Mbps)	
Reference	LEON3	99.315	7902	-	Virtex-6
[11]	KECCAK-256	273.2*	1395	12777	Virtex-5
	KECCAK-512	273.2*	1165	6556	Virtex-5
	KECCAK-256	301.0*	1220	11843	Virtex-6
	KECCAK-512	301.0*	1231	7225	Virtex-6
[13]	KECCAK-224	189.0	1117	5915	Virtex-5
	KECCAK-256	189.0	1117	6263	Virtex-5
	KECCAK-384	189.0	1117	8190	Virtex-5
	KECCAK-512	189.0	1117	8518	Virtex-5
[14]	KECCAK-256	205	1433	8397	Virtex-5
Our Work	ALL-IN-ONE	99.246	746$^{\triangle}$	†	Virtex-6

TP: Throughput; -: not supported; ALL-IN-ONE: supporting KECCAK-224, KECCAK-256, KECCAK-384 and KECCAK-512; *: the author computed based on the original paper; $\triangle$: extra hardware resources needed in ASIP implementation compared to reference design; †: throughput may vary according to different block size.

The area of our design shown in Table II is the extra area needed when integrating with customized hardware components. The design proposed in [11] is around two times faster than our design, but it has double the area as compared to our design. Our design can support up to four KECCAK algorithm with different parameters, where as Homsirikamol's designs are only targeted for KECCAK-256 and KECCAK-512. We also conclude that the speed and the throughput in hardware accelerator design is better than the proposed designs, but they occupy larger area compared to our proposed designs. On the other hand, when integrating these hardware accelerators into the original system, it requires extra driver programs to be compatible to the original system.

V. CONCLUSIONS

We successfully applied ASIP technique to a new cryptographic algorithm, KECCAK, which would meet the requirements in embedded systems. The critical part of KECCAK algorithm has been identified, which is bit rotation of multiple words. We proposed to offload this part into hardware and it can be integrated into the original system by the customized instruction. We make use of 64-bit data processing based on the 32-bit LEON3 processor, which leads to the reduction on execution cycles and code size. To have a fair evaluation, we re-implemented KECCAK-224, KECCAK-256, KECCAK-384 and KECCAK-512 based on LEON3 processor. Compared to these reference designs, our proposed design achieved around 87% reduction in execution cycles and 10.5% reduction in code size, with only slight increase in the critical path. Compared to the original LEON3 processor, the proposed designs take up 9.4% extra area without affecting the original critical path. Compared to the existing hardware accelerators, the proposed designs only take up half of the area resources and do not require development of extra driver programs when integrating into the overall system. In the future work, we will integrate other cryptographic algorithms into the system to suit security requirements.

REFERENCES

[1] National Institute of Standards and Technology. Advanced Encryption Standard (AES), FIPS-197, 2001.
[2] T. Good and M. Benaissa, "Very small FPGA application-specific instruction processor for AES," IEEE Transactions on Circuit and Systems I, vol. 53, no. 7, pp. 1477–1485, July 2006.
[3] S. Tillich and J. Groβschädl, "Accelerating aes using instruction set extensions for elliptic curve cryptography," in ICCSA (2), LNCS 3481, Springer, 2005, pp. 665–675.
[4] S. Tillich and J. Groβschädl, "Instruction set extensions for efficient aes implementation on 32-bit processors," in CHES, ser. LNCS 4249, Springer, 2006, pp. 270–284.
[5] P. Grabher, J. Groβschädl, and D. Page, "Light-weight instruction set extensions for bit-sliced cryptography," in CHES, LNCS 5154, Springer, 2008, pp. 331–345.
[6] Y. Wang and Y. Ha, "A performance and area efficient ASIP for higher-order DPA-resistant AES", IEEE Journal on Emerging and Selected Topics in Circuits and Systems, vol. 4, no. 2, pp. 190-202, June 2014.
[7] Y. Wang and Y. Ha, "FPGA-based 40.9-gbits/s masked AES with area optimization for storage area network," IEEE Transactions on Circuit and System II, vol. 60, no. 1, pp. 36-40, January 2013.
[8] http://keccak.noekeon.org/
[9] http://www.gaisler.com/.
[10] M. Gouicem, "Comparison of seven SHA-3 candidates software implementations on smart cards", https://eprint.iacr.org/2010/531.pdf.
[11] E. Homsirikamol, M. Rogawski and K. Gaj, "Throughput vs. Area Trade-offs in High-Speed Architectures of Five Round 3 SHA-3 Candidates Implemented Using Xilinx and Altera FPGAs", CHES2011, LNCS 6917, Springer, pp. 417–506.
[12] K. Gaj, E. Homsirikamol and M. Rogawski, "Comprehensive Comparison of Hardware Performance of Fourteen Round 2 SHA-3 Candidates with 512-bit Outputs Using Field Programmable Gate Arrays," http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/papers/GAJ_SHA3_512.pdf
[13] B. Baldwin, A. Byrney, L. Luz, M. Hamilton, N. Hanley, M. O'Neillz and W. P. Marnane, "FPGA implementations of the round two SHA-3 candidates," FPL 2010, 2010, pp. 400–407.
[14] S. Matsuo, M. Knezevic, P. Schaumont, I. Verbauwhede, A. Satoh, K. Sakiyama and K. Ota "How Can We Conduct 'Fair and Consistent' Hardware Evaluation for SHA-3 Candidate?" Second SHA-3 Candidate Conference, 2010, Available online at http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/