FPGA Implementation of CubeHash, Grøstel, JH, and SHAvite-3 Hash Functions

A. H. Namin, G. Li, J. Wu, J. Xu, Y. Huang, O. Nam, R. Elbaz and M. A. Hasan
Department of Electrical and Computer Engineering, University of Waterloo
Waterloo, Ontario N2L 3G1 Canada
Email: {anamin,g27li,bxwu,j27xu,y43huang,onam,reouven}@engmail.uwaterloo.ca,
ahasan@uwaterloo.ca

Abstract

In this work, FPGA implementation of the compression function for four of the second round candidates of the SHA-3 competition are presented. All implementations were performed using the same technology and optimization techniques to present a fair comparison between the candidates. For our implementations we have used the Stratix III FPGA family from Altera. Achieved results are compared with similar implementations to provide a fair comparison of candidates performance in hardware.

Index terms : Hash functions, SHA-3, CubeHash, Grøstel, JH, SHAvite-3, hardware implementation, compression function, FPGA.

I. Introduction

A cryptographic hash function (or algorithm) is a function that takes a variable-size message and returns a fixed-size output, which is called the message-digest [1]. Hash functions are used in a variety of applications including, message authentication, digital signature, fingerprinting, and data indexing [1]. Secure Hash Algorithm-1 (SHA-1) is a hash function designed by the National Security Agency (NSA) and published as a U.S. Federal Information Processing Standard [2]. It is by far the most widely used hash algorithm for security applications and protocols. Recently a mathematical weakness has been found in the architecture of SHA-1 suggesting a more secure hash function would be desirable for future use [3].

National Institute of Standards and Technology (NIST) is currently conducting an open, public competition to identify suitable candidates for the new hash algorithm (SHA-3) [4]. NIT has also recommended rapid transition from SHA-1 to SHA-2 (a stronger version of SHA-1) till complete development of SHA-3. At present, fourteen candidates exist in the second round of the SHA-3 competition. The software source code of all the candidates is available online which can be easily compiled for different platforms. A comprehensive comparison of software implementation and performance of the candidates can be found in [6].

Regrettably, only a few of the candidates contain hardware implementation results of their proposal. Also, a fair comparison of the implementation results could not be done easily since different platform and technologies have been used for the implementations. The main goal of this work is to implement a number of SHA-3 candidates in hardware using the same technology and design techniques and create a fair comparison of their hardware performance. An earlier version of this work resulted in hardware implementation of five of the candidates namely, Blue Midnight Wish, Luffa, Skein, Shabal, and Blake [13]. In this work we present hardware implementation results of four other candidates namely, Cubehash, Grøstel, JH, and SHAvite-3 along with that of SHA-2. For our hardware implementations we have used the Statix III FPGA family from Altera. A complete comparison of hardware performance of all nine candidates is presented in the comparison section.

The rest of this work is structured as follows. In Section II, a general discussion about hash functions and our design approach is presented. A summary of CubeHash, Gostel, JH, SHAvite-3 and SHA-2 hash algorithms and their hardware implementation is given in Section III-VII. Section VIII discusses a summary of hardware implementation results and comparison to other candidates hardware implementations. Some concluding remarks are presented in Section IX.

II. A Brief Review of Hash Algorithms and general Design Approach

To process a variable-size input message, most hash functions are based on a two stage process. The first stage called the *Preprocessing stage* receives the variable-size input, pads it to the proper size and then breaks it down into fixed-size message blocks. Padding the message to the proper size varies between different algorithms and might include adding the message length, a counter value, or even a constant value to the input.

The message blocks then arrive consecutively at the second stage referred to as the *Hash computation stage*. Here, the message blocks are processed iteratively by a function called the compression function through a number of rounds. The inputs of the compression function are the message block and the output of the compression function from the previous round. Using this feed-back structure, the final fixed-size output would be a digest hash which is a function of all previous inputs.

NIST has demanded that all candidates to provide different message digests of 224, 256, 384 and 512 bits sizes. However, in this work we just focus on the 256-bit versions of the algorithms. Our implementations have been carried out using the VHDL to describe the algorithms in hardware. The VHDL files were then compiled with the Quartus II software package from ALtera under UNIX environment. For our implementations we target the Stratix III FPGA family. All designs contain serial-in parallel-out (SIPO) and parallel-in serial-out (PISO) modules to transfer the data in/out of the FPGA. Using these modules, we would take into account the limited number of I/O pins and the I/O pins speed limits of the FPGA. More details of these modules are given in the comparison section.

III. CubeHash Hash Function

CubeHash has been proposed by Dan Bernstein in [8]. It makes use of four different operations in its compression function; 32-bit addition, rotate, swap and XOR operations. Two tunable parameters (r, b) are used to set the security/performance trade-offs of the system. For CubeHash-256, r and b are equal to 1 and 32, respectively.

The compression function starts by initializing a 1024 bit internal state from the input message and the previous hash value. Then there exists one round of transformation to the state. The round is composed of two addition, two rotate, four swap, and two XOR operations. Details about these operations and their order of applications can be found in [8]. After the round operation, integer '1' is XORed into the last state word and the finalization stage begins which is made of ten identical round operations similar to the initial round. The result is a 1024 bit state, of which the least significant 256-bits create the output hash. The general block diagram used for the hardware implementation of CubeHash-256 is shown in Fig. 1.

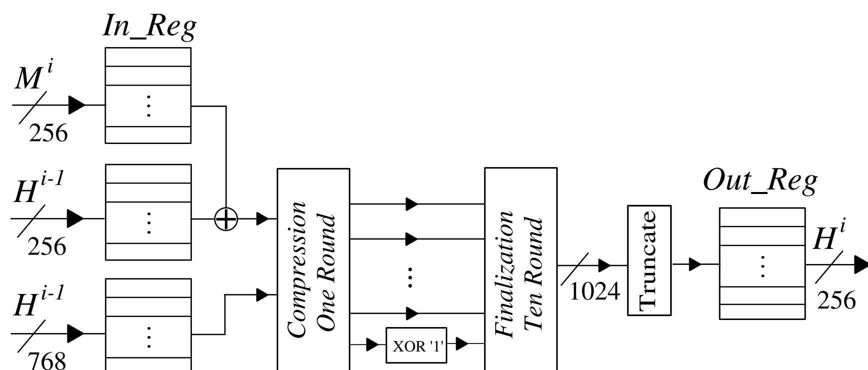

Fig. 1. Hardware architectures used for the compression function of CubeHash-256

FPGA implementation result of Cubehash-256 is summarized in the first row of Table I. In this table, C.F. Clk Frequency represents the maximum frequency that the compression function could be clocked at. C.F. Clk Cycles represents the required number of clock cycles for the compression function to create a 256 bit message digest. Combinational ALUTs represents the number of LUTs required for implementation of the combinational circuits. Dedicated logic Registers shows the number of ALUTs use as registers by In_reg and Out_reg registers. Also in the table, I/O Clk cycles and frequency are the result of using PISO and SIPO modules which are explained in the comparison section.

IV. GRØSTEL HASH FUNCTION

The Grøstel algorithm is due to L. R. Knudsen et al. [9]. The compression function is based on two underlying permutation modules P and Q running in parallel. Design of the permutation modules was inspired by the Rijndael block cipher, each consisting of a number of rounds (ten rounds each for the 256-bit version). Each round contains four separate transformations: Addconstant, SuByte, ShiftByte, and MixByte.

The AddConstant adds a round-dependent constant to the state variable. The SubBytes substitutes each byte in the state by another byte using S-box. The ShiftByte cyclically shifts the bytes inside the state variable. In MixBye each byte of the state is seen as an element in $GF(2^8)$ and is being multiplied by another constant element inside the field. SubByte and MixByte transformation are defined the same way as in Rijndael algorithm. The final hash is created by exclusive-oring of the previous hash with the outputs of the P and Q modules. The general block diagram used for the hardware implementation of Grøstel-256 is shown in Fig. 2. FPGA implementation results of Grøstel-256 are summarized in the second row of Table I.

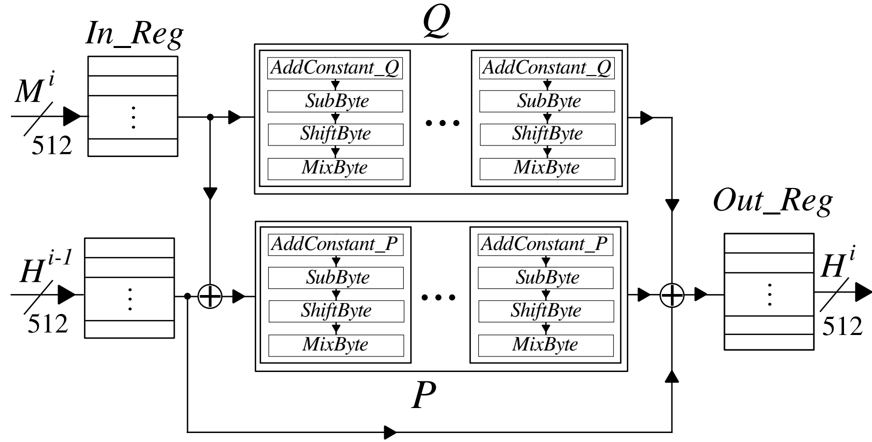

Fig. 2. Hardware architectures used for the compression function of Grøstel-256

V. JH HASH FUNCTION

Hongjun Wu has proposed the JH algorithm [10]. The compression process starts by exclusive-oring the input message to the first part of the previous hash. After regrouping the bits, there exist 35 rounds of the round function. The round function uses the generalized AES design methodology and is made of three separate layers: S-Box, Linear transformation (L), and Permutation (P). There exist two different S-boxes inside the S-box layer. The parameter round constant determines which of the S-boxes is used for each round. The linear transformation layer uses multiplication over the finite field $GF(2^4)$ using the irreducible polynomial $x^4 + x + 1$. The permutation layer is similar to the row rotation in the AES. The next steps in compression are one layer of S-Box and a de-grouping module. At the end, the msb part of the state variable is exclusive-ored with the input to create the output hash. The general block diagram used for the hardware implementation of JH-256 is shown in Fig. 3. FPGA implementation results of JH-256 are summarized in the third row of Table I.

VI. SHAVITE-3 HASH FUNCTION

The SHAvite-3 algorithm has been proposed by Eli Biham and Orr Dunkelman [11]. It is based on the Hash Iterative Frame Work (HAIFA) construction and it uses AES building blocks with block size of 128 bits [11],[12]. Each AES round uses four operations: 8-bit S-box, cyclic shift, multiplication over finite field $GF(2^8)$, and XOR. The compression function is made of a block cipher (E256) in the Davis-Meyer mode. E256 is a twelve round Feistal block cipher where each round is composed of three full rounds of AES. Each AES round requires a key which is obtained from message expansion of the message blocks by the key generator module. The key generator module uses 16 rounds of AES and 128 XOR gates to create 36 key of size 128-bits (, using the message block, and the counter as the inputs. The general block diagram used for the hardware implementation of SHAvite-3-256 is shown in Fig. 4. FPGA implementation result of the JH-256 is summarized in the fourth row of Table I.

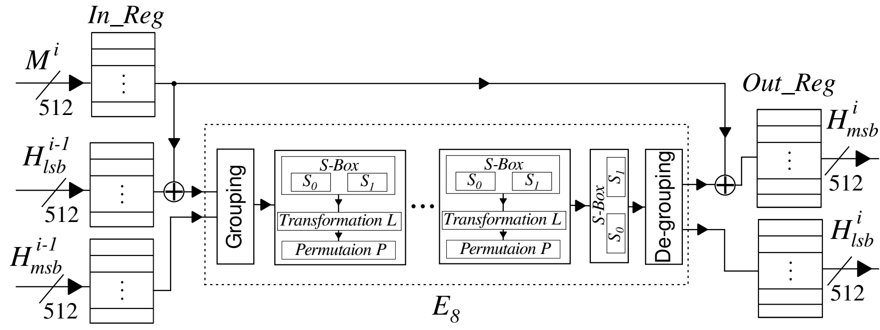

Fig. 3. Hardware architectures used for the compression function of JH-256

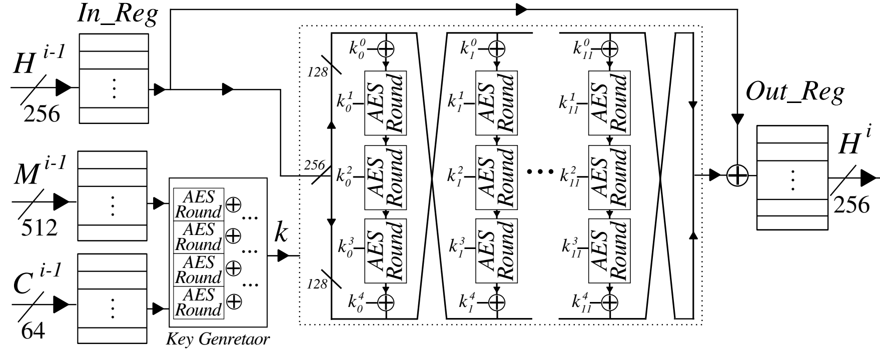

Fig. 4. Hardware architectures used for the compression function of SHAvite-3-256

VII. SHA-2 Hash Function

Considering the discovery of the weakness in SHA-1, NIST has requested the rapid transition to the stronger SHA-2 family of hash functions. The SHA-2 hash functions are structurally similar to SHA-1 and potentially carry the same weakness but since they are much stronger, practical attacks are unlikely in near future.

The basic data block used in SHA-2 (SHA-256) is a word which is 32 bits long. SHA-256 makes use of bit-wise logical word XOR and AND operations along with word addition (modulo 232), rotate right and shift right operations. The compression function is made of 64 consecutive round functions. Each round function received three inputs: 128-bit state variable (in 32-bit groups as input words a-h), K_t which is a 32-bit constants and w_t which is a variable created through the message schedule process from input message blocks. the output of the round function is the next state value (output words a-h). The hardware architecture used for implementation of SHA-256 is shown in Fig. 5. In this figure blocks $\sum_0$, $\sum_1$, Ch, and Maj are logical functions made of rotate, shift and logical operations [15]. FPGA implementation result of the SHA-256 is summarized in the last row of Table I.

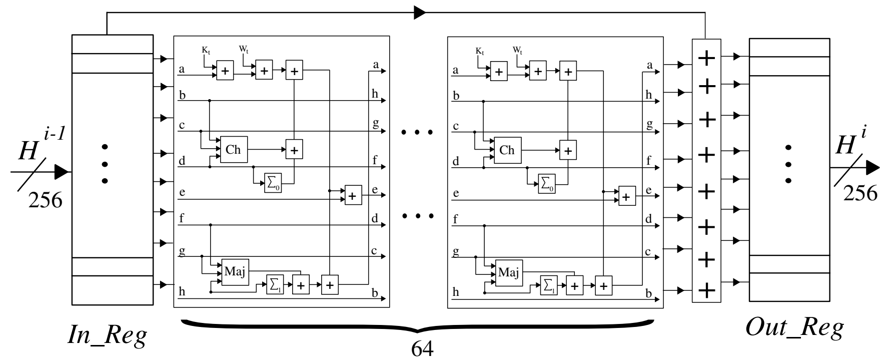

Fig. 5. Hardware architectures used for the compression function of SHA-256

Hash	C.F. Clk Frequency	C.F. Clk Cycles	I/O Clk Frequency	I/O Clk Cycles	Total Delay	Combinational ALUTs	Dedicated Logic Registers	Area × Delay Cost Function	I/O Pins
CubeHash	$20.88\,MHz$	1	$400\,MHz$	40	$147\,ns$	19514	800	1767318	73
Grøstel	$19.59\,MHz$	1	$400\,MHz$	32	$131\,ns$	98298	2592	13204800	110
JH	$15.79\,MHz$	1	$400\,MHz$	64	$223\,ns$	73978	4128	17417638	113
SHAvite-3	$7.36\,MHz$	1	$400\,MHz$	24	$195\,ns$	53324	1888	10766340	173
BMW[13]	$9.55\,MHz$	1	$400\,MHz$	32	$184\,ns$	12917	2607	2856416	111
Luffa[13]	$47.04\,MHz$	1	$400\,MHz$	16	$61\,ns$	16552	3247	1207739	283
Skein-1c[13]	$161.42\,MHz$	72	$400\,MHz$	18	$491\,ns$	1385	1858	1592313	146
Shabal[13]	$195.35\,MHz$	48	$400\,MHz$	32	$325\,ns$	1440	4000	1768000	289
Blake[13]	$46.97\,MHz$	11	$400\,MHz$	24	$294\,ns$	5435	2453	2319072	144
SHA-256	$2.28\,MHz$	1	$400\,MHz$	24	$498\,ns$	24317	1824	13018218	109

TABLE I

FPGA IMPLEMENTATION SUMMARY OF THE DIFFERENT COMPRESSION FUNCTIONS

VIII. FPGA Implementation Comparison

We have used Stratix III FPGA family from Altera for our Implementations. Quartus II software package from Altera under UNIX environment was used to implement the designs. All implementations contain SIPO and PISO modules to transfer the data into or out of the FPGA. Using these modules, the limited number of I/O pins and the I/O speed limits of the FPGA are taken into account. The SIPO module receives the input data as 32-bit words and then stores them for the proper output selected by the input select lines. Using this method input bits can be loaded into the SIPO module 32-bits at a time. Then the data can be transfered to the compression function in parallel. The PISO module uses a similar idea; it loads the data from the compression function in parallel and then transfers them to the output 32-bits at a time.

FPGA implementation results of the four candidates namely CubeHash, Grøstel, JH, and SHAvite-3 are listed in the first four rows of the Table I. For the purpose of comparison, implementation results for our previous work [13] on five other candidates namely Blue Midnight Wish, Luffa, Skein, Shabal, and Blake using the same FPGA technology are listed in the next five rows of the table [13]. The last row of the table, represents the implementation results of SHA-2 from Section VII.

In Table I, the I/O Clk Frequency represents the maximum speed at which the SIPO and PISO modules could be clocked at. The I/O speed is limited by the FPGA itself which in our case is $2.5\,ns$ [14]. Also in the same table I/O Clk cycles, represents the required number of clock cycles by SIPO and PISO modules to load the data into/out of the FPGA. The details of SIPO and PISO modules are shown in Fig. 6.

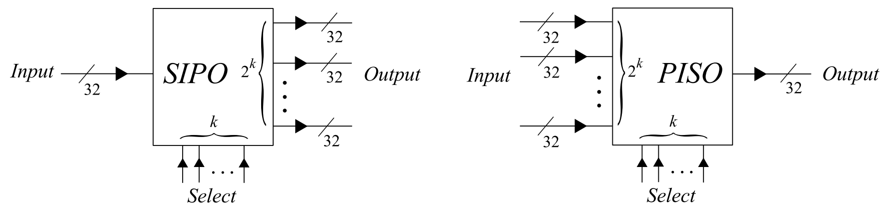

Fig. 6. SIPO and PISO modules used in FPGA implementations

Regarding the compression function speed, as can be seen from the table, Luffa presents the fastest architecture followed by CubeHash, Grøstel and JH. Taking into account the I/O delay, the fastest candidate is still Luffa followed by Grøstel, CubeHash, and Blue Midnight Wish. Regarding area usage, Skein, Shabal, and Blake present the smallest area utilization.

Comparing the candidates to SHA-256, all considered candidates perform faster than SHA-256 (with or without taking into account the I/O delay) except Skein which is almost presenting the same performance. Regarding the area usage Grøstel, JH, and SHAvite-3 require more area compared to SHA-256. Note that usually minimizing area and delay are both important goals in any hardware implementation. To this end, we have defined area times delay as a measure of performance in our comparison table. We can see that Luffa outperforms all other designs following by Skein, CubeHash, and Shabal as it is shown in the Area × Delay Cost Function column in Table I.

IX. Conclusions

We have presented FPGA implementation of the compression function for four candidates of the SHA-3 competition. Implementations have been carried out using the Stratix III FPGA family from Altera. Results have also been compared with the implementation results of some other candidates using the same platform.

X. Acknowledgments

The authors would like to thank CMC for providing the CAD tools and support. This work was supported in part by NSERC strategic project grant awarded to Dr. Hasan.

References

[1] Menezes, A. J., Van Oorschot, P. C., Vanstone, S. A.: Handbook of Applied Cryptography. The CRC Press series on discrete mathematics and its applications, pp. 321-383, 1997.

[2] National Institute of Standards and Technology: Secure Hash Standard, Federal Information Processing Standards (FIPS) Publication 180-3, October, 2008.

[3] Wang, X., Yao, A., Yao, F.: New Collision search for SHA-1. Crypto 2005 rump sesson.

[4] National Institute of Standard and Technology (NIST): Cryptographic Hash Algorithm Competition Website: http://csrc.nist.gov/groups/ST/hash/sha-3/.

[5] The SHA-3 Zoo - The ECRYPT hash function website. Website: http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo.

[6] European Network of Excellence for Cryptology II (ECRYPT II): ECRYPT Benchmarking of All Submitted Hashes (EBASH): http://bench.cr.yp.to/ebash.html.

[7] Fleischmann, E., Forler, C., Gorski, M.: Classification of the SHA-3 Candidates. International Association for Cryptologic Research (IACR) ePrint archive

[8] Bernstein, D. J., CubeHash specification, Website: http://cubehash.cr.yp.to/

[9] Gauravaram, p.,Knudsen, L. R., Matusiewicz, K., Mendel, F., Rechberger, C., Schlaffer, M., Thomsen, S. S., Grøstel - a SHA-3 candidate, Website: http://www.groestl.info.

[10] Wu, H., SHA-3 proposal JH, Website: http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/.

[11] Biham, E., Dunkelman, O., The SHAvite-3 Hash Function, Website: http://www.cs.technion.ac.il/ orrd/SHAvite-3/.

[12] Biham, E., and Dunkelman, O.: A Framework for Iterative Hash Functions HAIFA, NIST 2nd hash function workshop, Santa Barbara, August 2006.

[13] Namin, A. H., Hasan, M. A.: Implementation of the Compression Function for Selected SHA-3 Candidates on FPGA, 17th Reconfigurable Architectures Workshop, accepted.

[14] ALTERA: Stratix III Device Handbook, Volume 1, May 2009.

[15] Federal Information Processing Standards Publication (FIPS PUB) 180-2: Secure Hash Signature Standard (SHS), pp. 1-79, Aug. 2002.