High Throughput Hardware Implementation of Secure Hash Algorithm (SHA-3) Finalist: BLAKE

Kashif Latif
National University of Sciences and Technology, Sector H-12
Islamabad, Pakistan
kashif@pnec.edu.pk

Athar Mahboob
National University of Sciences and Technology, Sector H-12
Islamabad, Pakistan
athar@pnec.edu.pk

Arshad Aziz
National University of Sciences and Technology, Sector H-12
Islamabad, Pakistan
arshad@nust.edu.pk

Abstract— **Cryptographic hash functions are at the heart of many information security applications like digital signatures, message authentication codes (MACs), and other forms of authentication. As a consequence of recent innovations in cryptanalysis of commonly used hash algorithms, NIST USA announced a publicly open competition for selection of new standard Secure Hash Algorithm called SHA-3. An essential part of this contest is hardware performance evaluation of the candidates. In this work we present a high throughput efficient hardware implementation of one of the final round candidate of SHA-3: BLAKE. We implemented and investigated the performance of BLAKE on latest Xilinx FPGAs. We show our results in form of chip area consumption, throughput and throughput per area. We compare and contrast these results with most recently reported implementations of BLAKE. Our design ranks highest in terms of speed, achieving throughputs of 2.47Gbps on Virtex 7, 2.41Gbps on Virtex 6 and 2.28Gbps on Virtex 5.**

Keywords- Authentication, SHA-3, BLAKE, Cryptographic Hash Functions, High Throughput Hardware, FPGA.

I. INTRODUCTION

A cryptographic hash function is a one way procedure whose input is an arbitrary block of data and output is a fixed-size bit string, which is known as the hash value. Hash functions are widely used in many information security applications like digital signatures, message authentication codes (MACs), and other forms of authentication.

Presently, there are many well known cryptographic hash functions in use, but with recent improvements in cryptanalysis, most have been found vulnerable. In last few years, cryptanalysis of several hash algorithms has found serious weaknesses. In 2004, Xiaoyun Wang et al. presented the collisions for MD4, MD5, HAVAL-128 and RIPEMD [1]. A breakthrough in cryptanalysis of SHA-1 hash algorithm was achieved in August 2005. Professor M. Szydlo found a collision in SHA-1 in 2^{63} operations [2]. Previously, it was thought that 2^{80} operations are required to find a collision in SHA-1 for a 160-bit block length. This attack is expected to find a hash collision, i.e., two messages with the same hash value in 2^{63} operations. No attacks have yet been reported on the SHA-2 variants; however, they are algorithmically similar to SHA-1. Furthermore, M. Stevens reported a collision attack on MD5 in 2006 [3].

Keeping in view the above mentioned facts, National Institute of Standards and Technology (NIST) USA announced a public competition in the Federal Register Notice published on November 2, 2007 [4] to develop a new cryptographic Hash algorithm called SHA-3.

In response to NIST's announcement 64 submissions were reported, out of which 51 entries fulfilled the minimum submission requirements and were selected as the first round candidates. These candidates were reduced to 14 in Round 2 and further to 5 in the final round of the competition. Five short listed candidates that advanced to the final round are BLAKE, Grøstl, JH, Keccak and Skein. The tentative time-frame for the end of this competition and selection of finalist for SHA-3 is in 4th quarter of 2012 [5].

Among the 5 finalists of SHA-3, we chose BLAKE for our work due to its competitive performance figures in all respective areas, i.e security, computational efficiency, simplicity and flexibility in terms of both software and hardware [6]. This paper describes high throughput efficient hardware implementation of BLAKE. Implementation results are shown for the latest Xilinx FPGAs; Virtex-5, Virtex-6 and Virtex-7. The remainder of this paper is organized as follows. We briefly give an overview about cryptographic hash functions in section II. Section III gives brief description of BLAKE. In section IV we present the efficient hardware implementation of BLAKE. In section V, we give the results of our work and compare it with other reported efficient implementations of BLAKE in section VI. Finally, we provide some conclusions and directions for future work in Section VII.

II. CRYPTOGRAPHIC HASH FUNCTIONS

A cryptographic hash function is a deterministic procedure whose input is an arbitrary block of data and output is a fixed-size bit string, which is known as the (cryptographic) hash value. Hash functions have the most fascinating property of input sensitivity. It is in this way that an intended or unintended change to the message will change the hash value drastically. This property makes hash

978-0-7695-4625-4/11 $26.00 © 2011 IEEE
DOI 10.1109/FIT.2011.42

functions an ultimate choice for applications requiring authenticity and integrity. More often, the data to be hashed is called the "message", and the hash value is sometimes called the **message digest** or simply the **digest**. A hash value H of plaintext M is generated by a hash function h of the form

$$H = h(M) \quad (1)$$

The ideal cryptographic hash function with inputs M, M' and outputs H, H' must have the following properties:

- **Simplicity:** It should be easy to compute the hash value for any given message M.
- **Preimage Resistance:** It should be very hard to find a message that has a given hash, i.e. to find any preimage M such that $h(M) = H$ when given any H.
- **Second Preimage Resistance:** It should be difficult to find another input message such that both messages have the same Hash value, i.e. given M, to find a second preimage $M' \neq M$ such that $h(M') = h(M)$. This property is sometimes referred to as 'weak collision resistance'.
- **Collision Resistance:** It should be difficult to find two different messages that have the same hash value, i.e to find any M and M' which have the same hash i.e. that $h(M) = h(M')$. Such pair of messages is called a cryptographic hash collision. This property is sometimes referred to as 'strong collision resistance'.
- **Input Sensitivity:** It should be infeasible to modify a message without changing its hash.

Hashes and encryption are different: Encryption converts plaintext into ciphertext and by using the appropriate key it converts it back. The two texts roughly correspond to each other with respect to size. Encryption is a two-way operation. On the other hand, hashes convert a stream of data into a fixed size hash value. No matter how long the message may be but its hash value will be of fixed size and it is strictly a one way operation resisting inversion.

Cryptographic hash functions have many information security applications, particularly in digital signatures and message authentication codes (MACs). Moreover, they can be used for fingerprinting, for data indexing in hash tables, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption during communication.

III. BRIEF DESCRIPTION OF BLAKE

BLAKE is based on Bernstein's stream cipher ChaCha and uses iteration mode HAIFA [7]. Figure 1 shows the construction of BLAKE's compression function.

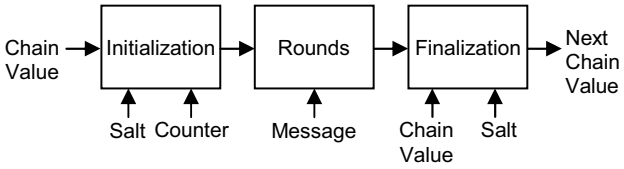

FIGURE 1. The local wide-pipe construction of BLAKE's compression function

BLAKE-224 and BLAKE-256 operate on 32-bit words while BALKE-384 and BLAKE-512 operate on 64-bit words. The inner state of the compression function is represented as a 4×4 matrix of words. We briefly explain here the functionality of BLAKE-256 compression function.

The compression function takes four input values: a chaining hash value $h = h_0, \dots, h_7$, a message block $m = m_0, \dots, m_{15}$, a salt $s = s_0, \dots, s_3$, and a counter $t = t_0, t_1$. Each word is 32-bit long. Additionally, compression function utilizes 32-bit constants $c_0, \dots, c_{15}$ and permutation $\sigma_r\{0, \dots, 15\}$ [7]. BLAKE Permutation constants are shown in Table I.

TABLE I. BLAKE PERMUTATION CONSTANTS

σ_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
σ_1	14	10	4	8	9	15	13	6	1	12	0	2	11	7	5	3
σ_2	11	8	12	0	5	2	15	13	10	14	3	6	7	1	9	4
σ_3	7	9	3	1	13	12	11	14	2	6	5	10	4	0	15	8
σ_4	9	0	5	7	2	4	10	15	14	1	11	12	6	8	3	13
σ_5	2	12	6	10	0	11	8	3	4	13	7	5	15	14	1	9
σ_6	12	5	1	15	14	13	4	10	0	7	6	3	9	2	8	11
σ_7	13	11	7	14	12	1	3	9	5	0	15	4	8	6	2	10
σ_8	6	15	14	9	11	3	0	8	12	2	13	7	1	4	10	5
σ_9	10	2	8	4	7	6	1	5	15	11	9	14	3	12	13	10

The output of compression function is a new chain value $h' = h'_0, \dots, h'_7$. We can write compression function as

$$h' = compress(h, m, s, t) \quad (2)$$

Initialization: The initialization component of BLAKE consists of initialization of 4x4 matrix state of 16 words $v_0, \dots, v_{15}$, as shown below:

$$\begin{pmatrix} v_0 & v_1 & v_2 & v_3 \\ v_4 & v_5 & v_6 & v_7 \\ v_8 & v_9 & v_{10} & v_{11} \\ v_{12} & v_{13} & v_{14} & v_{15} \end{pmatrix} \leftarrow \begin{pmatrix} h_0 & h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 & h_7 \\ s_0 \oplus c_0 & s_1 \oplus c_1 & s_2 \oplus c_2 & s_3 \oplus c_3 \\ t_0 \oplus c_4 & t_0 \oplus c_5 & t_1 \oplus c_6 & t_1 \oplus c_7 \end{pmatrix} \quad (3)$$

Round Function: After initializing state v, compression function iterartes the round function 14 times. A round function consists of simple transformations over state v. Each round computes following 8 G functions:

$$\begin{aligned} &G_0(v_0, v_4, v_8, v_{12}) && G_2(v_1, v_5, v_9, v_{13}) \\ &G_4(v_2, v_6, v_{10}, v_{14}) && G_6(v_3, v_7, v_{11}, v_{15}) \\ &G_8(v_0, v_5, v_{10}, v_{15}) && G_{10}(v_1, v_6, v_{11}, v_{12}) \\ &G_{12}(v_2, v_7, v_8, v_{13}) && G_{14}(v_3, v_4, v_9, v_{14}) \end{aligned}$$

Where $G_i(a, b, c, d)$ is defined as:

$$
\begin{aligned}
a &= a + b + \left(m_{\sigma_r(i)} \oplus c_{\sigma_r(i+1)}\right) \\
d &= (d \oplus a) \gg 16 \\
c &= c + d \\
b &= (b \oplus c) \gg 12 \\
a &= a + b + \left(m_{\sigma_r(i+1)} \oplus c_{\sigma_r(i)}\right) \\
d &= (d \oplus a) \gg 8 \\
c &= c + d \\
b &= (b \oplus c) \gg 7
\end{aligned} \tag{4}
$$

Where $\oplus$ is the bit-wise XOR operation, $\gg$ is the right rotate operator and σ is a permutation, indexed by round number r and type of G_i function ($i = 0,2,4,\ldots 14$). Here we specified the alternate G functions as described in [7]. For actual specifications please refer to [7].

The first four G functions G_0, G_2, G_4 and G_6 can be computed in parallel, because each of them updates a distinct column of the matrix. This is referred to as a *column step*. Likewise, the last four G functions G_8, G_{10}, G_{12} and G_{14} update distinct diagonals of the matrix and thus can also be computed in parallel. This is referred to as a *diagonal step*. At round $r > 9$, the permutation used is $\sigma_{r \bmod 10}$, for example, in the last round $r = 13$ and the permutation $\sigma_{13 \bmod 10} = \sigma_3$ is used.

Finalization: After the 14 round sequence, new chain value $h' = h'_0, \ldots, h'_7$ is calculated from state $v_0, \ldots, v_{15}$ with combination of the initial chain value $h_0, \ldots, h_7$ and the salt $s_0, \ldots, s_3$, as follows:

$$
\begin{aligned}
h'_0 &= h_0 \oplus s_0 \oplus v_0 \oplus v_8 \\
h'_1 &= h_1 \oplus s_1 \oplus v_1 \oplus v_9 \\
h'_2 &= h_2 \oplus s_2 \oplus v_2 \oplus v_{10} \\
h'_3 &= h_3 \oplus s_3 \oplus v_3 \oplus v_{11} \\
h'_4 &= h_4 \oplus s_0 \oplus v_4 \oplus v_{12} \\
h'_5 &= h_5 \oplus s_1 \oplus v_5 \oplus v_{13} \\
h'_6 &= h_6 \oplus s_2 \oplus v_6 \oplus v_{14} \\
h'_7 &= h_7 \oplus s_3 \oplus v_7 \oplus v_{15}
\end{aligned} \tag{5}
$$

Iterated hash: To calculate hash of a message greater than 512-bits (16 32-bit words), BLAKE-256 compression function is used iteratively as follows:

$$
\begin{aligned}
&h^0 \leftarrow IV \\
&\text{for } i = 0, \ldots, N-1 \\
&\quad h^{i+1} \leftarrow compress(h^i, m^i, s, l^i) \\
&\text{return } h^N
\end{aligned}
$$

Here we split the message into 16-word blocks $m^0, \ldots, m^N$. s is the salt value and l^i denotes the number of message bits in current message block excluding padding bits. h^0 is initialized with initial value IV. The new chaining value h^{i+1} becomes the input of next iteration and eventually after processing of N blocks, final hash value h^N is returned.

IV. IMPLEMENTATION

We have implemented the core functionality of 256-bit variant of BLAKE. Our design is fully autonomus with a complete I/O interface. We targeted for efficient implementation but keeping in mind the hardware performance comparison with previously reported implementations. We assure this appraoch by catering to common development environment, same design methodology and use of same set of chip resources. We further assured it by forcing our designs to map on LUT based logic and not to use dedicated hardware resources like BRAMs, Multipliers and DSPSlices. Memories are also implemented using distributed RAMs/ROMs because they utilize the LUT resources and memory requiremet of an algorithm is reflected in terms of utilized area.

A. I/O Interface

The input/output interface we developed is shown in Figure 2. All I/O transactions are synchronized. Each I/O is sampled at the rising edge of clock pulse. The input cycle is initiated by I/O interface by setting *load* signal to high. Hash Module acknowledges the request if it is able to receive data by setting *ack* signal to high. After receiving acknowledgment, I/O interface makes available 64-bit word of data at each rising edge of clock pulse. During the transaction of data, *ack* signal remains at logic high. After receiving desired amount of input words Hash Module sets the *ack* signal to low. Accordingly, I/O interface pulls the *load* signal to low if no more transactions are required. If message blocks are still present, *load* signal will remain high but Hash Module acknowledges it after one clock cycle from the previous transaction. In the same way when Hash Module is ready with a valid hash value it signals the I/O interface by putting *hash_valid* signal to high. After putting *hash_valid* signal hash module outputs 64-bit words on each rising edge of clock cycle until the desired hash length is achieved. I/O interface is designed in a way that it does not affect the ongoing processing within hash module. That is, we can make I/O transactions at the same time while hash of a message block is in progress.

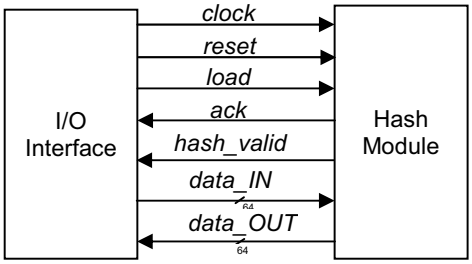

FIGURE 2. Input/Output Interface

B. Control and Data Paths

Hash module consists of two major parts, the control path and the data path. Control path consists of Finite State Machine, State register, clock and counter. Data path consists of Input registers, Hash Core, Intermediate registers and Output registers. Input registers of data path consist of a Serial In Parallel Out (SIPO) register and other registers to store message and other input parameters like salt value. Hash Core is the main arithmetic logic unit of the BLAKE hash algorithm. Intermediate registers are utilized to store intermediate results of the hash algorithm. Output register contains the resulting hash and it is a Parallel In Serial Out (PISO) register to serially output the result.

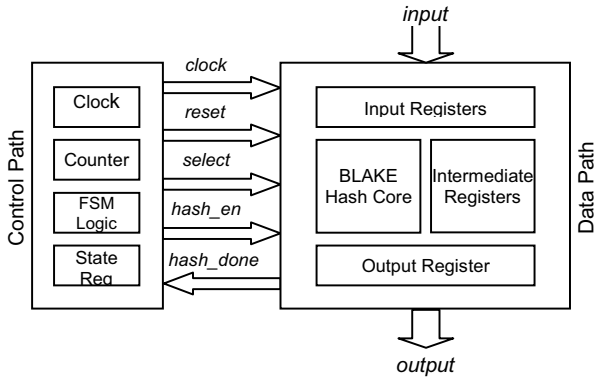

FIGURE 3. Hash module separated in control and data paths

C. Data Path of BLAKE

The data path implemented for BLAKE is shown in Figure 4. All nets represent data width of 512-bits. The V_Reg represents the v matrix register, on which processing of BLAKE algorithm takes place. The CV_Reg stores the intermediate chaining hash values. Initialization module initializes the V_Reg by taking IV (Initial Value) or chaining hash value as input. Core functionality of BLAKE algorithm is represented by G function module. Four instantiations of G function module are utilized to compute 4 G operations in parallel. These instantiations are represented as G1, G2, G3 and G4. Each G function instance computes a different G function on alternate clock cycles. G1 instance computes G_0 and G_8, G2 computes G_2 and G_{10}, G3 computes G_4 and G_{12} and similarly G4 computes G_6 and G_{14}, on alternate clock cycles. G function module is implemented using pure combinational logic. ADD and XOR operations are implemented using Verilog operators '+' and '^' respectively. Circular shift operations are performed through rewiring of the nets. Each round takes 2 clock cycles to complete, therefore 28 clock cycles are required to complete 14 rounds of BLAKE algorithm. After completion of 14 rounds, finalization module computes final or next chaining hash value by taking contents of V_Reg and CV_Reg as input.

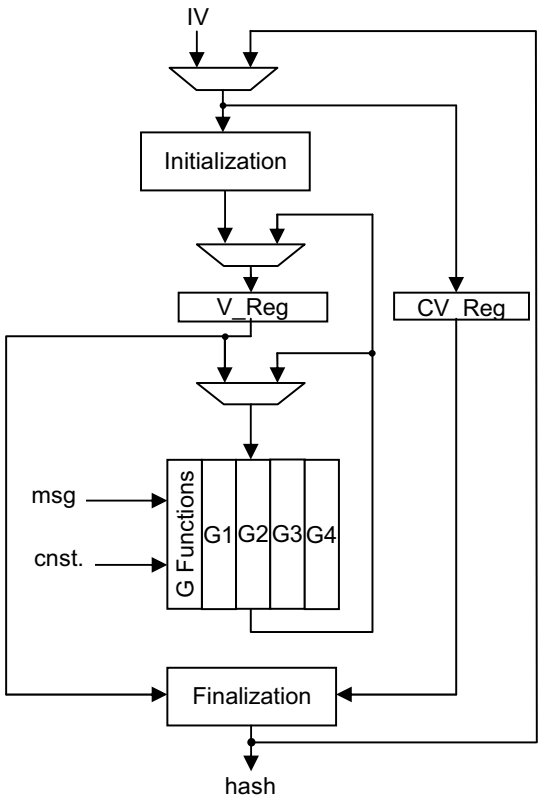

FIGURE 4. Datapath of BLAKE

V. IMPLEMENTATION RESULTS

The design has been implemented on Xilinx Virtex 5, Virtex 6 and Virtex 7 FPGAs. Detailed device specifications are: Virtex 5 155T, speed grade 3, package FF1136 (5vlx155tff1136-3), Virtex 6 LX365T, speed grade 3, package FF1156 (6vlx365tff1156-3) and Virtex 7 585T, speed grade 3, package FG1157 (7v585tffg1157-3), respectively. The resulting clock frequencies and area utilization after place and route are reported. Table II shows achieved area consumption ($Area$), clock frequency (F_{max}) and time delay (T) for implemented design.

TABLE II. RESULTS FOR BLAKE-256

Device	$Area$ [Slices]	F_{max} [MHz]	T [ns]
Xilinx Virtex 7	1566	135.355	7.388
Xilinx Virtex 6	1602	131.961	7.578
Xilinx Virtex 5	1739	124.55	8.029

A. Throughput

From these results we can calculate throughput of our design. The throughput TP of a given design can be calculated by:

$$TP = \frac{Block\ Size}{T_{hash}} \quad (6)$$

Where $Block\ Size$ is the block size of message in bits. T_{hash} is the total time required to calculate hash value which is given by:

$$T_{hash} = T \cdot N_{clk} \quad (7)$$

Where T is the time period of the system clock and N_{clk} is the number of clock cycles required for a valid hash output. Table III shows these parameters and throughput results.

TABLE III. THROUGHPUT RESULTS FOR BLAKE-256

Device	$Block\ Size$ [bits]	N_{clk} [cycles]	T [ns]	T_{hash} [ns]	TP [Gb/s]
Xilinx Virtex 7	512	28	7.388	206.86	2.47
Xilinx Virtex 6	512	28	7.578	212.18	2.41
Xilinx Virtex 5	512	28	8.029	224.81	2.28

B. Throughput Per Area

Throughput per area is a significant performance measure that combines the performance effect of both area and speed in a single value. It measures the contribution of each unit area to throughput and hence the efficiency of the implementation.

From the results given above, we can calculate throughput per area of our designs. The throughput per area TPA of a given design can be calculated by:

$$TPA = \frac{TP}{Area} \quad (8)$$

Where TP is the throughput of the design. $Area$ is the occupied area of the design on the chip. In our case we have chosen the number of slices as a unit of area consumption. Table IV shows throughput per area results.

TABLE IV. THROUGHPUT PER AREA RESULTS FOR BLAKE-256

Device	TP [Gb/s]	$Area$ [Slices]	TPA [Mb/slice]
Xilinx Virtex 7	2.47	1566	1.58
Xilinx Virtex 6	2.41	1602	1.51
Xilinx Virtex 5	2.28	1739	1.31

VI. COMPARISON WITH PREVIOUS WORK

We have taken this opportunity to report a SHA-3 candidate's hardware implementation results, for the very first time, on latest Xilinx FPGA i.e. Virtex 7. Before this no results have been reported on Virtex 7. All reported work to date utilized Virtex 5 and Virtex 6 at most. Table V shows the comparison of our results with previously reported implementations in terms of throughput, area and throughput per area.

TABLE V. COMPARISON OF BLAKE-256 IMPLEMENTATIONS

Author(s)	Device	F_{max} [MHz]	$Area$ [slice]	TP [Gbps]	TPA [Mbps/slice]
This work	Virtex 7	135.35	1566	2.47	1.58
This work	Virtex 6	131.96	1602	2.41	1.51
This work	Virtex 5	124.55	1739	2.28	1.31
Aumasson et al. [7]	Virtex 5	100.00	1217	1.76	1.45
Baldwin et al. [8]	Virtex 5	91.35	1653	0.83	0.50
Matsuo et al. [9]	Virtex 5	115.00	1660	0.64	0.38
Kris Gaj et al. [10]	Virtex 5	117.06	1871	2.07	1.10
E. Hom. et al. [11]	Virtex 6	-	1247	1.96	1.57
E. Hom. et al. [11]	Virtex 5	-	1691	2.25	1.33

Most of previously reported results are for 10 rounds of the algorithm. However, in final round specifications, BLAKE-256 has been tweaked from 10 to 14 rounds. The results listed here have been calculated again for 14 rounds based on the reported clock frequencies and number of clock cycles consumed for respective designs [7-10]. E. Homsirikamol et al. reported their results for 14 rounds [11]. E. Homsirikamol et al. discussed and reported their results for various architectures of BLAKE-256. For performance comparison, we considered the results of architecture most similar to our work.

Our results for Virtex 5, Virtex 6 and Virtex 7 are far ahead from all previously reported work in terms of throughput. Device wise comparison shows that our design for BLAKE-256 is top performer in terms of throughput. In terms of throughput per area ratio, our Virtex 6 design is placed second with very marginal difference with [11] and our Virtex 5 design is placed third with marginal differences with [7] and [11].

VII. CONCLUSION AND FUTURE WORK

In this work we have presented efficient hardware implementation of SHA-3 finalist: BLAKE. We reported the implementation results of BLAKE-256 on Xilinx Virtex 5, Virtex 6 and Virtex 7 FPGAs. We reported the performance figures of our implementation in terms of area, throughput and throughput per area and compared it with previously reported implementation results. Results achieved in this work are exceeding the performance for implementations reported so far. We compared and contrasted the performance figures of BLAKE-256 on

Virtex 5, Virtex 6 and Virtex 7. This work serves as performance investigation of BLAKE-256 on most up-to-date FPGAs.

We used the 256-bit variant of BLAKE for our implementation. Other variants are 224, 384 and 512, as specified by NIST for SHA-3. Present work may easily be modified for all these variants. Future work may consist of performance investigation of all these variants. Furthermore, pipelining the design at appropriate points may result in higher throughput rates. Existing design may be enhanced by using pipelining techniques.

REFERENCES

[1] X. L. Xiaoyun Wang, D. Feng and H. Yu, "Collisions for hash functions MD4, MD5, HAVAL-128 and RIPEMD", *Cryptology ePrint Archive*, Report 2004/199, http://eprint.iacr.org/2004/199, pp. 1-4.

[2] M. Szydlo, "SHA-1 collisions can be found in 263 operations", CryptoBytes Technical Newsletter, August 19, 2005.

[3] M. Stevens, "Fast collision attack on MD5", ePrint-2006-104, March 2006 http://eprint.iacr.org/2006/104.pdf, pp. 1-13.

[4] Federal Register / Vol. 72, No. 212 / Friday, November 2, 2007 / Notices, http://csrc.nist.gov/groups/ ST/hash/documents/ FR_Notice_Nov07.pdf, pp. 1-9.

[5] National Institute of Standards and Technology (NIST), "Cryptographic Hash Algorithm Competition". http://www.nist.gov/itl/csd/ct/.

[6] NIST Interagency Report 7764, "Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition", February 2011, pp. 1-38.

[7] J. Aumasson, L. Henzen, W. Meier, R. W. Phan, "SHA-3 Proposal BLAKE version 1.3", http://131002.net/blake/blake.pdf, December 2010, pp. 1-79.

[8] B. Baldwin, N. Hanley, M. Hamilton, L. Lu, A. Byrne, M. Neill and W. P. Marnane, "FPGA Implementations of the Round Two SHA-3 Candidates", 2nd SHA-3 Candidate Conference, Santa Barbara, August 23-24, 2010, pp. 1-18.

[9] 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?", 2nd SHA-3 Candidate Conference, Santa Barbara, August 23-24, 2010, pp. 1-15.

[10] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates using FPGAs," in Proceedings of Cryptographic Hardware and Embedded Systems workshop, CHES 2010, Santa Barbara, Aug. 2010.

[11] E. Homsirikamol, M. Rogawski and K. Gaj, "Comparing Hardware Performance of Round 3 SHA-3 Candidates using Multiple Hardware Architectures in Xilinx and Altera FPGAs", ECRYPT II Hash Workshop 2011, Tallinn, Estonia, May 19-20, 2011, pp. 1-15.