Efficient Hardware Implementations and Hardware Performance Evaluation of SHA-3 Finalists

Kashif Latif, M Muzaffar Rao, Arshad Aziz and Athar Mahboob

National University of Sciences and Technology (NUST), H-12 Islamabad, Pakistan
kashif@pnec.edu.pk, mrao@pnec.edu.pk, arshad@nust.edu.pk, athar@pnec.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. In 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 efficient hardware implementations and hardware performance evaluations of SHA-3 finalists. We implemented and investigated the performance of SHA-3 finalists on latest Xilinx FPGAs. We show our results in the form of chip area consumption, throughput and throughput per area on most recently released devices from Xilinx on which implementations have not been reported yet. We have achieved substantial improvements in implementation results from all of the previously reported work. This work serves as performance investigation of SHA-3 finalists on most up-to-date FPGAs.

Keywords: SHA-3, Performance Evaluation, Cryptographic Hash Functions, High Speed Encryption Hardware, FPGA.

1 Introduction

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. Cryptographic hash functions are widely used in many information security applications like digital signatures, message authentication codes (MACs), and other forms of authentication. There is a long list of cryptographic hash functions but with recent advances, many have been found vulnerable and should not be used. Vulnerabilities found in a number of hash functions in August 2004, including SHA-0, RIPEMD, and MD5, led to the rendering of long-term security of SHA-1, RIPEMD-128, and RIPEMD-160 algorithms suspect.

In 2004, Xiaoyun Wang et al. presented the collisions for MD4, MD5, HAVAL-128 and RIPEMD [1]. There was a breakthrough in cryptanalysis of SHA-1 Hash Algorithm in August 2005. Professor M. Szydlo found that it is possible to find 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. Furthermore, M. Stevens reported a collision attack on MD5 in 2006 [3].

To ensure the long-term robustness of applications that use hash functions National Institute of Standards and Technology (NIST) USA has 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 of the competition. After 2nd SHA-3 conference, 5 out of 14 Round 2 candidates have been selected and promoted to the Final Round on December 9, 2010. Five short listed candidates advancing 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].

This paper describes: efficient hardware implementations, implementation results on latest FPGA technologies from Xilinx and hardware performance evaluation of these algorithms. The remainder of this paper is organized as follows. We briefly give an overview of SHA-3 finalists in section 2. In section 3, we discuss the related work reported to date. In section 4, we describe the methodology we adopted for efficient implementation and respective

performance evaluation of SHA-3 finalists. In section 5, we present the efficient hardware architectures for SHA-3 finalists. In section 6, we give the results of our work and compare them with previously reported work in section 7. Section 8 presents performance evaluation of SHA-3 finalists. Finally, we provide conclusion in Section 9.

2 SHA-3 Finalists Overview

We provide here a quick overview of five SHA-3 finalists. Detailed descriptions of these algorithms may be found in respective submission documents [6-10].

2.1 BLAKE

J. Aumasson et al. designed and proposed the BLAKE Hash family for SHA-3 [6]. BLAKE is based on Bernstein's stream cipher ChaCha and uses iteration mode HAIFA [11]. The internal construction of BLAKE is local wide-pipe, same as of LAKE hash function [12]. BLAKE hash function consists of two basic variants, BLAKE-256 and BLAKE-512. BLAKE-256 operates on 32-bit words while BLAKE-512 operates on 64-bit words. The other required hash digest sizes may be derived from these two variants, i.e. 224 and 384. The inner state of the compression function is represented as a 4×4 matrix of words. The compression function consists of 8 instances of arithmetic function G. The G function consists of addition, XOR and rotation operations. Each G function operates on 4 elements of state matrix. BLAKE compression function consists of 14 rounds for BALKE-256 and 16 rounds for BLAKE-512.

2.2 Grøstl

P. Gauravaram et al. designed and proposed the Grøstl hash function for SHA-3 [7]. Grøstl is an iterated hash function built from two fixed, large and distinct but similar permutations P and Q. Grøstl is a byte-oriented SP-network which is based on components of AES [13]. These components are named as AddRoundConstant, SubByte, ShiftBytes and MixBytes. The Grøstl has two variants Grøstl-512 and Grøstl-1024. The internal state consists of two 8x8-byte matrices for Grøstl-512 and two 8x16-byte matrices for Grøstl-1024. The permutation Q operates on a message block and permutation P operates on XOR of message and chaining hash value or initial value. Each permutation P and Q is iterated 10 times for Grøstl-512 and 14 times for Grøstl-1024. The next chaining value of hash is calculated by XORing the outputs of P, Q and previous chaining hash value. After processing of all message blocks final hash is transformed applying permutation P on chaining hash and XORing it with permuted value. The resulting hash digest may be truncated to any desired length.

2.3 JH

Hongjun Wu designed and proposed the JH hash function for SHA-3 [8]. JH algorithm is based on the idea that large block ciphers can be constructed through small components and a constant key. JH algorithm generalizes the AES design methodology to high dimensions. JH-512 is the basic variant of JH. JH uses the same design for all variants, i.e. JH-224, JH-256 and JH-384. These variants only differ in initial values (IV) and output hash length. JH compression function is constructed from bijective function (a block cipher with constant key) [8]. JH compression function compresses a previous 1024-bit hash value H^{i-1} and 512-bit message block M^i into new 1024-bit hash value H^i. The bijective function E, consists of 42 rounds. Each round consists of 4-bit S-box substitution, a linear transformation and a series of three permutations. The 1024-bit state of JH is grouped into 256 4-bit pairs before start of round operations and de-grouped after it. Grouping and de-grouping of bits is defined in [8]. Two types of S-boxes are used and selection of S-box for a given 4-bit substitution is controlled by respective bit value of round constant.

2.4 Keccak

G. Bertoni et al. designed and proposed the Keccak Hash Function for SHA-3 [9]. Keccak is a family of sponge functions with members Keccak [r, c] characterized by two parameters, bitrate r and capacity c. The sum $r + c$ determines the width of the Keccak-f permutation used in the sponge construction and is restricted to values in {25, 50, 100, 200, 400, 800, 1600}. For SHA-3 proposal Keccak team proposed the Keccak [1600] with different r and c values for each desired length of hash output [9]. For 256-bit hash output r = 1088 and c = 512. For 512-bit hash output r = 576 and c = 1024. The 1600-bit state of Keccak [1600] consists of 5x5 matrix of 64-bit words. The compression function of Keccak consists of five steps: theta (θ), rho (ρ), pi (π), chi (χ) and iota (i). These steps consists of simple XOR, AND, NOT and permutation operations. Each compression step of Keccak consists of 24 rounds. Keccak hash function operation consists of three phases, initialization, absorbing phase and squeezing phase. Initialization is simply initializing the state matrix with all zeros. In absorbing phase each r-bit wide block of message is XORed with current matrix state and 24 rounds of Keccak permutation are performed. After absorbing all blocks of input message in that fashion there comes the squeezing phase. In squeezing phase the state matrix is simply truncated to desired length of output hash. If more than r-bit hash value is required then more Keccak permutations are performed and their results concatenated until hash width reaches the desired length.

2.5 Skein

N. Ferguson et al. designed and proposed the Skein family of cryptographic hash functions for SHA-3 [10]. Skein has three different internal state sizes: 256, 512, and 1024 bits. Each of these state sizes can support any output size. Skein-512 is the primary proposal for SHA-3. Skein is built from three components, Threefish tweakable block cipher, Unique Block Iteration (UBI) and Optional argument system. The tweakable block cipher makes every instance of compression unique by hashing configuration data along with input message. The compression function of Skein consists of a layer of non-linear MIX operations and permutation. MIX operation consists of addition modulo 2^{64}, rotation and XOR operation on a pair of 64-bit words. The Threefish compression function is used in UBI chaining mode to compress arbitrary length of input data to fixed size hash digest.

3 Related Work

There are two main streams of hardware implementations of algorithms on FPGA and ASIC platforms: *high speed implementations* and *compact implementations*. Various groups around the world are working on hardware performance evaluation of SHA-3 candidates using these two types of implementations. The SHA-3 Zoo website [14] reports the comprehensive results of reported work. Most of the reported work is focused on high speed architectures as it provides a direct snapshot of the basic operations' cost for a given algorithm. The relevant category for our work is high speed implementations on FPGAs. In Table 1, we provide a snapshot of high speed implementations' results, for FPGAs, from different groups. The comprehensive studies for all 14 round 2 candidates are reported by Baldwin et al. [15], Matsuo et al. [16], Gaj et al. [17] and Homsirikamol et al. [18]. For round 3, the only comprehensive results for all five finalists are reported by Homsirikamol et al. [19]. Homsirikamol et al. [19] discussed and reported their results for various architectures using pipelining, folding and loop unrolling approaches. For performance comparison, we quote here the results of architecture based on basic iterative approach. The specifications for BLAKE, Grøstl and JH have been tweaked for round 3. Hence, the results listed in Table 1 have been calculated again for round 3 specifications based on the reported clock frequencies and number of clock cycles consumed for respective designs [15-17]. Some efficient compact implementations of SHA-3 finalists are reported in [20-22]. The high speed ASIC implementations are reported in [23-25].

4 Implementation Methodology

We have implemented the 256-bit and 512-bit variants of all five SHA-3 finalists. Our designs are fully autonomus with complete I/O interfaces. We targeted for efficient implementations but keeping in mind the fair hardware performance comparison for these candidates. We assure this appraoch by catering for the follwing constraints:

Table 1. SHA-3 Finalists Implementations. F_{max} in MHz, $Area$ in Slices, TP in Gbps and TPA in Mbps/Slice

SHA-3 Finalist	Author(s)	Device	256-bit				512-bit			
			F_{max}	$Area$	TP	TPA	F_{max}	$Area$	TP	TPA
BLAKE	Aumasson et al. [6]	Virtex 5	100.00	1217	1.76	1.45	50.00	2389	1.55	0.65
	N. Sklavos [26]	Virtex	50.00	3101	0.91	0.29	27.00	11800	0.864	0.07
	Baldwin et al. [15]	Virtex 5	91.35	1653	0.83	0.50	71.05	2888	1.14	0.39
	Matsuo et al. [16]	Virtex 5	115.00	1660	0.64	0.38	-	-	-	-
	Kris Gaj et al. [17]	Virtex 5	117.06	1871	2.07	1.10	106.01	3276	3.29	1.00
	E. Hom. et al. [19]	Virtex 6	-	1247	1.96	1.57	-	2628	3.19	1.21
	E. Hom. et al. [19]	Virtex 5	-	1691	2.25	1.33	-	3337	3.16	0.95
Grøstl	Baldwin et al.[15]	Virtex 5	78.06	2579	3.24	1.26	113.12	4525	3.62	0.80
	Matsuo et al. [16]	Virtex 5	154	2616	1.97	0.75	-	-	-	-
	Kris Gaj et al. [17]	Virtex 5	355.87	1884	8.676	4.61	180.15	3466	6.36	1.84
	E. Hom. et al. [19]	Virtex 6	-	2630	9.34	3.55	-	5106	11.57	2.27
	E. Hom. et al. [19]	Virtex 5	-	2591	8.081	3.12	-	5254	10.12	1.93
JH	Baldwin et al.[15]	Virtex 5	144.11	1763	1.64	0.93	144.11	1763	1.64	0.93
	Matsuo et al. [16]	Virtex 5	201.00	2661	0.733	0.27	-	-	-	-
	Kris Gaj et al. [17]	Virtex 5	278.09	1108	3.39	3.06	275.48	1165	3.36	2.88
	E. Hom. et al. [19]	Virtex 6	-	847	5.70	6.73	-	896	5.34	5.95
	E. Hom. et al. [19]	Virtex 5	-	909	4.62	5.09	-	1020	4.73	4.64
Keccak	Keccak Team [9]	Virtex 5	122.00	1330	5.20	3.91	-	-	-	-
	Strömbergson [27]	Spartan3A	85.00	3393	4.80	1.41	-	-	-	-
	Strömbergson [27]	Virtex 5	118.00	1483	6.70	4.52	-	-	-	
	Baldwin et al.[15]	Virtex 5	195.73	1971	6.26	3.17	195.73	1971	8.52	4.32
	Matsuo et al. [16]	Virtex 5	205.00	1433	4.20	2.93	-	-	-	-
	Akin et al. [28]	Spartan 3	81.40	2024	3.46	1.71	-	-	-	-
	Akin et al. [28]	Virtex-II	136.60	2024	5.81	2.87	-	-	-	-
	Akin et al. [28]	Virtex 4	142.90	2024	6.07	3.00		-	-	-
	Kris Gaj et al. [17]	Virtex 5	238.38	1229	10.81	8.79	276.86	1236	6.64	5.37
	E. Hom. et al. [19]	Virtex 6	-	1165	11.84	10.17	-	1231	7.23	5.87
	E. Hom. et al. [19]	Virtex 5	-	1395	12.77	9.16	-	1220	6.56	5.37
Skein	Baldwin et al. [15]	Virtex 5	-	-	-	-	83.58	2756	0.97	0.35
	Matsuo et al. [16]	Virtex 5	115.00	854	0.283	0.33	-	-	-	-
	Kris Gaj et al. [17]	Virtex 5	116.35	843	1.568	1.86	104.34	1520	2.812	1.85
	M. Long. [29]	Virtex 5	114.94	931	0.407	0.44	114.94	1758	0.82	0.46
	S. Tillich. [30]	Virtex 5	68.40	937	1.751	1.87	69.04	1632	3.535	2.17
	S. Tillich. [30]	Spartan 3	26.14	2421	0.669	0.28	26.66	4273	1.365	0.32
	E. Hom. et al. [19]	Virtex 6	-	1510	3.27	2.17	-	1591	3.11	1.96
	E. Hom. et al. [19]	Virtex 5	-	1728	2.93	1.70	-	1658	2.81	1.7

- Common Environment: It effects the implementations in terms of the level of expertise, language, coding techniques, design methodoly, and development tools. We assured it by keeping: common implementer for all candidates, using Verilog as the common language, use of a common design methodology (discussed in next point) and using Xilinx's ISE 13.1 as the common development tool.

- Design Methodology: For a fair comparison it is necessary to utilize same set of harware recources for all candidates. We assured it by forcing our designs to map on LUT based logic and not to use dedicated hardware resources like BRAMs, Multipliers and DSPSlices. We further assured this approach by using LUT primitives from Xilinx HDL libraray, wherever possible. We will discuss this approach in detail later in this section. Memories are also implemented using distributed RAMs/ROMs because they utilize the LUT resources and memory requirement of a candidate is reflected in terms of utilized area.
- Common I/O Interface: Using common Input/Output interface assures the identical flow of data for all candidates in investigation. It also assures modular approach by reusing the same module wherever possible.
- Overhead suppression: We did not implement the optional parameters of the candidates like salt input, Hash Tree functionality and HMAC etc. Furthermore, we assume that input message blocks are already padded outside. The padding techniques are almost similar for every algorithm and mainly result in same area overhead.

4.1 I/O Interface

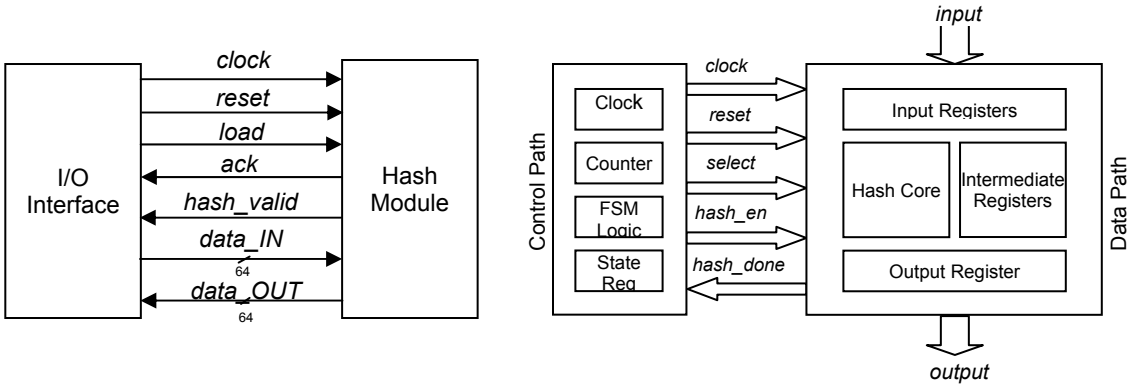
The developed input/output interface is shown in Fig. 1(a). All I/O transactions are synchronized. Each I/O is sampled at the rising edge of clock pulse. The input cycle is initiated by I/O interface by setting *load* signal to high. Hash Module acknowledges the request if it is able to receive data by setting *ack* signal to high. After receiving acknowledgment, I/O interface make 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.

4.2 Control and Data Paths

Hash module of each candidate consists of two major parts, the control path and the data path. Block diagram of hash module separated in control path and data path is depicted in Fig. 1(b). 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 key in case of Skein. Hash Core is the main arithmetic logic unit of the 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.

4.3 FPGA Specific Issues and Their Implication on SHA-3 Algorithm Architectures

The architectures of latest FPGA families from Xilinx (Virtex 5, Virtex 6, Spartan 6 and Virtex 7) are based on 6-input LUTs, named LUT6 [31]. A CLB Slice of Xilinx FPGA consists of 4 such LUTs. Each LUT6 has six independent inputs and two independent outputs. These LUTs may be configured and used in many different ways. A LUT6 may be used as independent 5-input LUT using LUT5 primitive from Xilinx HDL library, shown in Fig. 2(a). On the other hand, it is possible to implement any two 5-input logic functions with shared inputs using LUT6_2 primitive, shown in Fig. 2(b). In this case, LUT input $i5$ selects between two 5-input logic functions to connect at output $O6$. Same LUT6_2 primitive may be used to draw two independent outputs from a LUT6, with shared 5-inputs. In this case, input $i5$ should be tied to logic high (i.e. 1). The INIT value in hexadecimal, shown under attributes in Fig. 2(a) and 2(b) configures the LUT to perform desired operation at its inputs. The INIT value is derived by laying down the truth table for all possible combinations of LUT inputs and outputs. We have used these primitives excessively in architectural designs of SHA-3 finalists. We also exploit the techniques presented in [32] for efficient utilization of modern FPGA resources.



(a) (b)

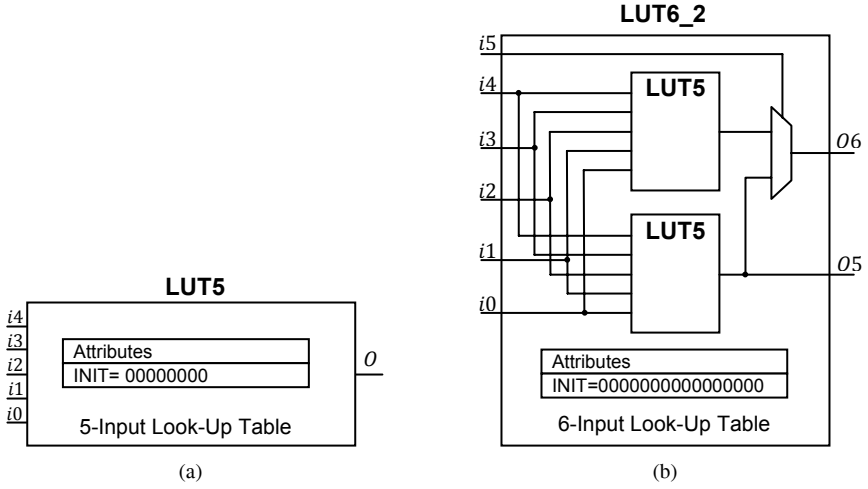
Fig. 1. (a) Common I/O Interface (b) Hash Module separated in Control and Data paths

5 Datapath Architectures for SHA-3 Finalists

This section provides architectural details of datapaths designed for SHA-3 finalists.

5.1 BLAKE

The datapath implemented for BLAKE is shown in Fig. 3. 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. BLAKE is one of the algorithms where use of specific library resources not turns into advantageous outcomes. The addition operation on Xilinx FPGAs is efficiently implemented by synthesis tools itself by using dedicated carry logic resources. The XOR and rotation operation are not expensive operations in terms of resource utilization. Hence, efficient direct coding of the equations of BLAKE returns good synthesis results. The 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.



(a) (b)

Fig. 2. LUT5 and LUT6_2 primitives in Xilinx HDL library

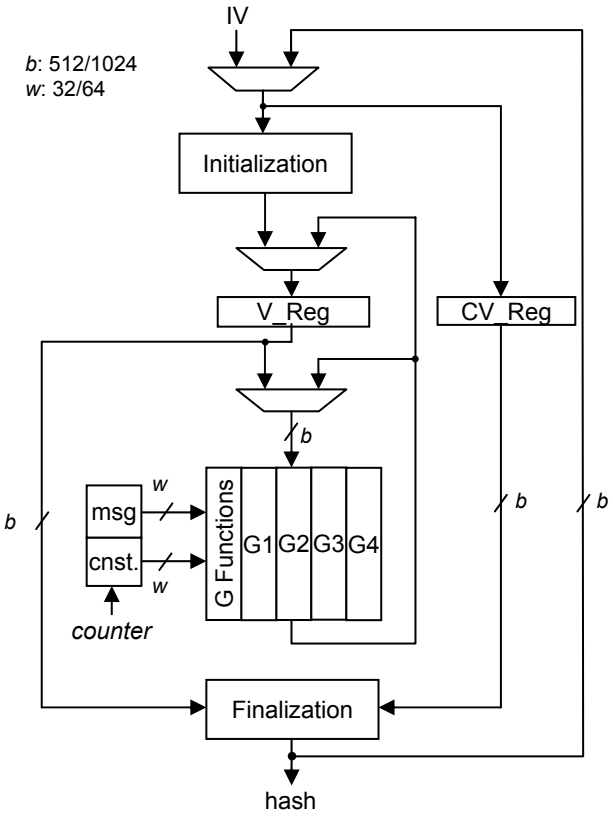


Fig. 3. Data path of BLAKE

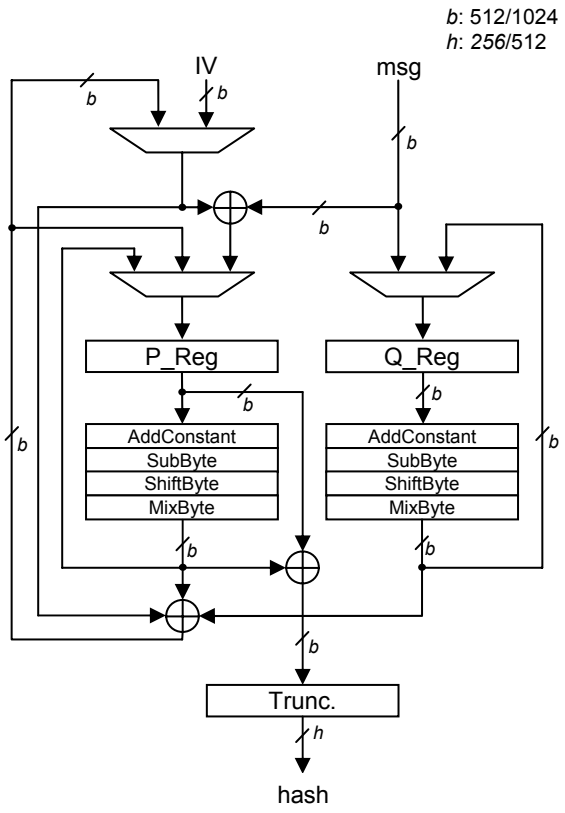


Fig. 4. Data path of Grøstl

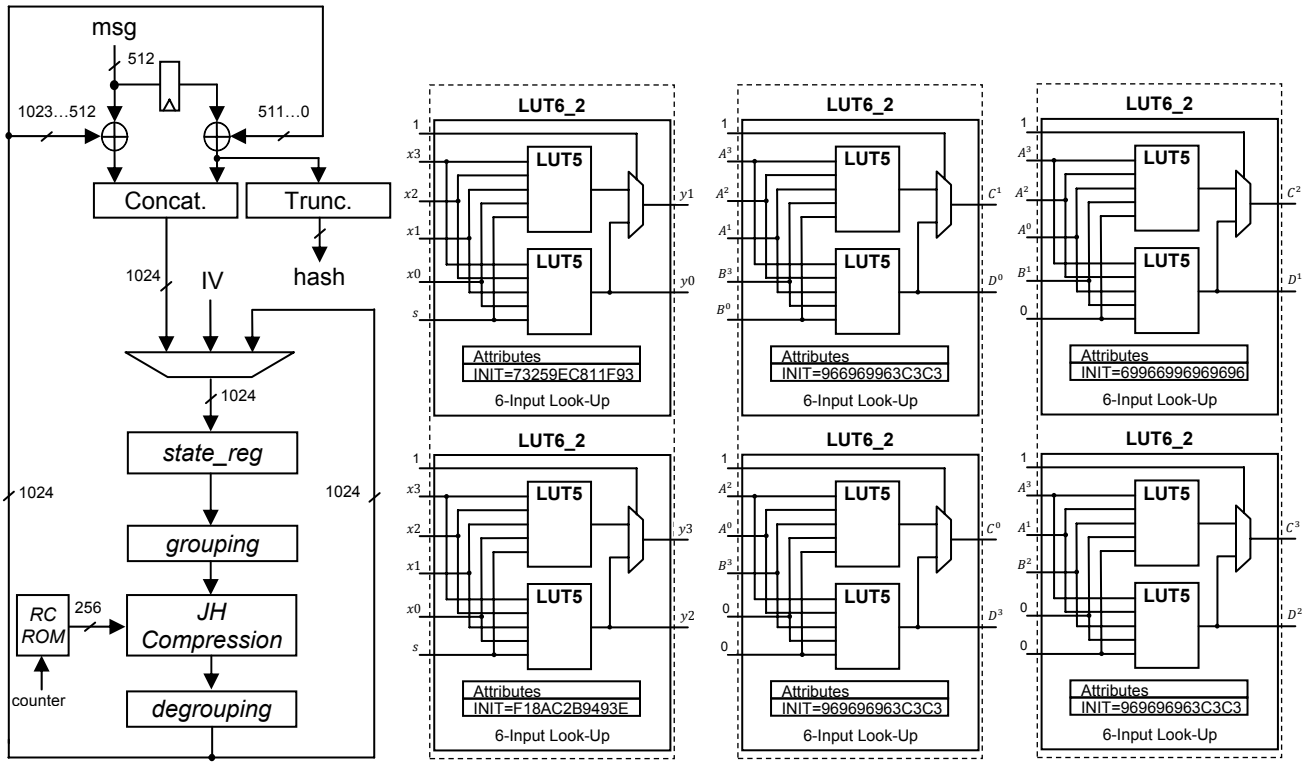
After completion of 14 rounds, finalization module computes final or next chaining hash value by taking contents of V_Reg and CV_Reg as input.

5.2 Grøstl

The datapath implemented for Grøstl is shown in Fig. 4. The compression function of Grøstl consists of two separate permutations *P* and *Q*. These permutations are almost identical and can be implemented in parallel. Two differences between *P* and *Q* are in the step of AddRoundConstant, where different round constants are used, and in ShiftByte step, where different scheme is used to circular shift the bytes left. AddRoundConstant step is a simple eight byte XOR operation of input matrix with round constant. SubByte is AES S-box substitution; it is implemented using combinational logic which utilizes the internal logic structure of inversion in Galois Field GF(2^8) and affine transformation, as described in [8]. ShiftByte is circular shift of bytes in matrix; it is implemented through simple re-labeling of the bytes in matrix. MixByte step multiplies each column of the matrix to a constant 8x8 matrix in Galois Field GF (2^8). This step is implemented as combinational logic described in [8]. In the beginning of every hash process the message block (msg) is directly pass on to *Q* permutation. The *P* permutation operates on XOR of msg and initial value (IV) or chaining hash value. Both permutations *P* and *Q* are iterated 10 times for Grøstl-512 and 14 times for Grøstl-1024. After completion of desired number of rounds next chaining hash value is obtained by XORing outputs of *P*, *Q* and previous chaining value. This process continues till the end of all message blocks. At the end, resulting chaining hash is again permuted through permutation *P* and XORed with permuted value to obtain final hash digest.

5.3 JH

The datapath implemented for JH is shown in Fig. 5(a). The *state_reg* represents the intermediate JH state register, on which processing of JH algorithm takes place. JH hash function uses the same algorithm for all hash digest sizes.

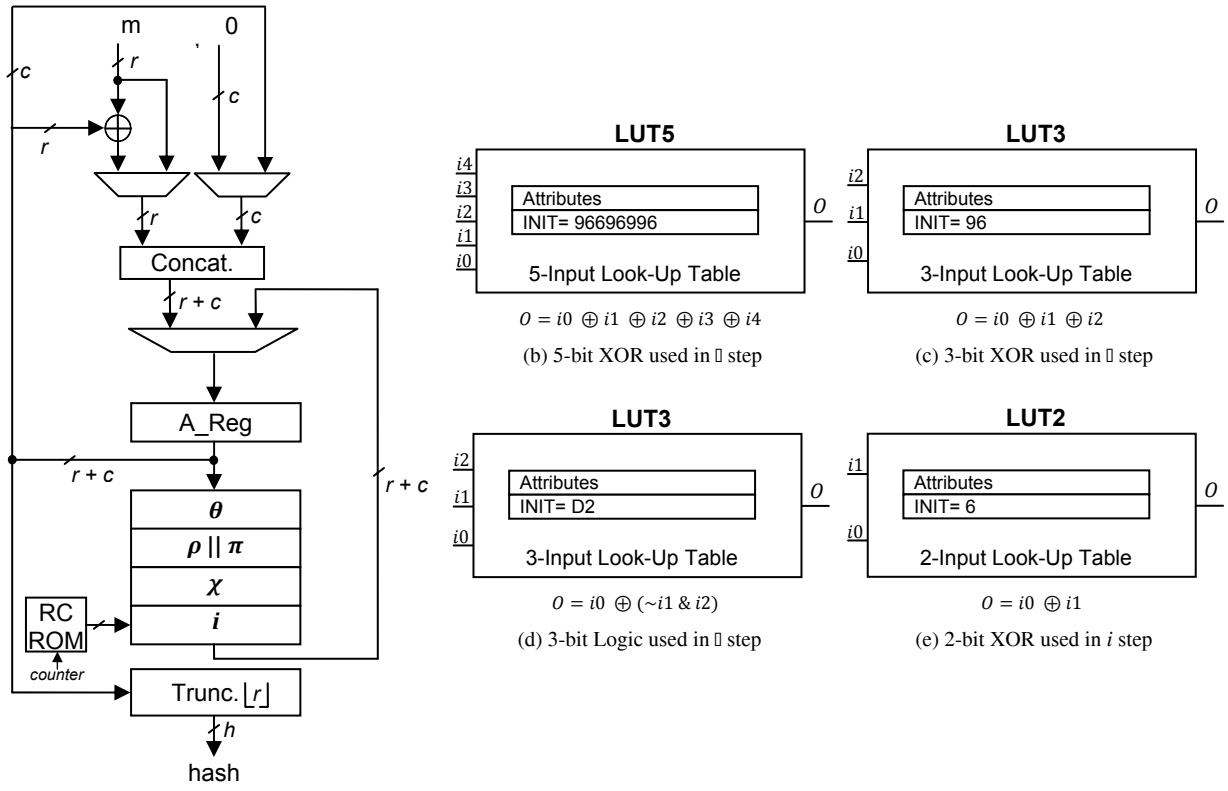


(a) Data path of JH (b) 5-to-4 bit S-box (c) Linear Transformation

Fig. 5. Architectural detail of JH

Hence, same data path is utilized for all hash digest sizes. The only difference between data path for different hash sizes is of initial values (IV) and hash output registers. In the beginning of every hash process *state_reg* is initialized with IV of desired hash digest size. Then a complete JH compression is processed by setting msg and round constant RC to zero. The higher order 512 bits of resulting state of JH compression is then XORed with first message block and stored in *state_reg*. Then contents of *state_reg* are processed through JH compression function with respective round constant. JH compression function consists of 42 rounds of its arithmetic logic unit (ALU). A single round is processed in one clock cycle. Therefore 42 clock cycles are required to complete 42 rounds of JH compression function. After completion of 42 rounds on a message block, resulting lower order 512 bits of JH compression state is XORed with msg, to obtain next chaining hash value. The higher order 512 bits of resulting chaining hash value is then XORed with next message block and stored in *state_reg* and same compression sequence is repeated again. This process continues till the end of all message blocks. At the end, resulting lower 512 bits of chaining hash value is truncated to the desired length of hash output. The Trunc. block in Fig. 5(a) represents the truncation operation. The Concat. block represents the concatenation operation. The *grouping* and *degrouping* blocks are used to perform grouping and de-grouping of JH state bits into 4-bit pairs as specified in JH specification document [8]. In terms of hardware implementation these steps are achieved through simple rewiring of interconnects, at no resource cost. The round constants (RC) are stored in ROM using 43x256 bit single port distributed ROM. Respective round constant is addressed during each round using round number as ROM address.

JH Arithmetic Logic Unit (ALU): JH ALU consists of S-boxes (S) and linear transformation units (L). JH ALU works on 4-bit pairs of JH state register contents. For S-box, we used LUT6_2 primitive (Fig. 2(b)) and used both of its output i.e. $O5$ and $O6$. Using this approach 4 S-boxes are adjusted within a single slice. In this approach S-box logic of JH ALU consists of only 128 slices. Implementation of a single S-box using this approach is depicted in Fig. 5(b). The INIT values (in hexadecimal) shown in figure, are actual configuration values for each LUT to perform S-box operation. Linear transformation is also implemented using same optimized approach. LUT6_2 primitive with both outputs $O5$ and $O6$ is used. Implementation of a single linear transformation unit (L) is depicted in Fig. 5(c). The INIT values (in hexadecimal) shown in figure, are actual configuration values for each LUT to perform L operation. Same variables are shown for inputs and outputs in Fig. 5(c) as denoted in linear transformation equations in specification document [8].



(a) Data path of Keccak

(b) 5-bit XOR used in θ step

(c) 3-bit XOR used in θ step

(d) 3-bit Logic used in ρ step

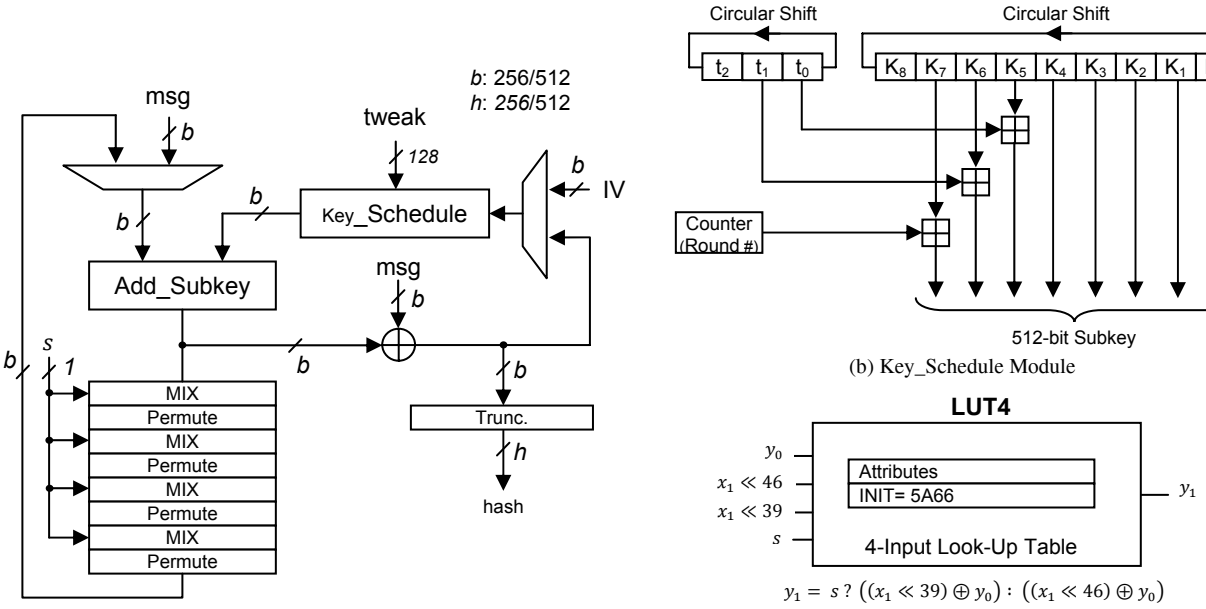
(e) 2-bit XOR used in *i* step

Fig. 6. Architectural detail of Keccak

5.4 Keccak

The datapath implemented for Keccak is shown in Fig. 6(a). The A_Reg represents the A matrix register, on which processing of Keccak algorithm takes place. Keccak data path is fully parameterized, such that the design may be synthesized for any value of r (bitrate) and c (capacity). For that reason, the width of each net is highlighted as r, c or $r + c$ in Fig. 6(a). The length of A_Reg also varies according to r and c and it is defined as $r + c$ (bits). For Keccak-256, r is specified as 1088-bits and c as 512-bits. For Keccak-512, r is specified as 576-bits and c as 1024-bits. Accordingly A_Reg will be of 1600-bits. In beginning of every hash process A_Reg is initialized with all zeros. First message block is directly copied to A_Reg after concatenating it with c wide stream of 0's. The Concat block in Fig. 6(a) represents the concatenation operation. Compression function of Keccak consists of five steps. In Fig. 6(a) each step is denoted by the symbol as specified in Keccak specifications. These steps are θ, ρ, π, χ and i. We have combined these steps during implementation, wherever possible. We have implemented ρ and π as a single step. Keccak algorithm's compression function consists of very simple arithmetic operations. It involves simple XOR, AND and NOT operations. These operations are implemented using LUT primitives from Xilinx specific libraries. Following are details of implementation of each step:

Theta (θ) Step: There are three equations in θ step. First equation (calculation of C) is implemented using LUT5 primitive for XOR logic as shown in Fig. 6(b). The INIT value in hexadecimal, shown under attributes in figure, configures the LUT to perform XOR operation at its inputs. The INIT value is derived by laying down the truth table for all possible combinations of LUT inputs. To XOR 5 64-bit operands of the equation, LUT5 primitive is instantiated 64 times. For complete implementation of equation, 5x64 LUT5 are required. We can combine remaining two equations of theta step. For its implementation, LUT3 primitive is used for XOR logic as shown in Fig. 6(c). The one bit rotation in last operand is implemented through rewiring. To implement the complete logic, 25x64 instantiations of LUT3 primitive are required.



(a) Data path of Skein

(b) Key_Schedule Module

(c) Selection between two rotation constants in MIX operation

Fig. 7. Architectural detail of Skein

Rho (ρ) and Pi (π) Steps: The ρ and π are permutations, which may be achieved through simple rewiring in hardware, at no resource cost. The cyclic shift constant $r[x, y]$ is fixed and known for each position of matrix A. It is also implemented by means of fixed rewiring.

Chi (χ) Step: In χ step three logical operations XOR, NOT and AND are used. These are implemented using LUT3 primitive as shown in Fig. 6(d). In order to accomplish the χ step, LUT3 with χ logic is instantiated 25x64 times.

Iota (i): The i step involves simple XOR of round constant with least significant 64 bits of A_Reg, i.e. $A[0,0]$. It is implemented using LUT2 primitive as shown in Fig. 4(d). LUT2 is instantiated 64 times for i step.

The round constants (RC) are stored in ROM using 24x64 bit single port distributed ROM. Respective round constant is addressed during each round using round number as ROM address. These five steps or a single round of Keccak algorithm are accomplished in one clock cycle. Therefore 24 clock cycles are required to complete 24 rounds of Keccak algorithm. After completion of 24 rounds on a message block, resulting r-bits of state of A_Reg are XORed with next message block and same round sequence is repeated again. This process continues till the end of all message blocks. At the end, state of A_Reg is truncated to the desired length of hash output.

5.5 Skein

The datapath implemented for Skein is shown in Fig. 7(a). Add_Subkey module consists of 8 64-bit adders, implemented using fast carry chain logic available in Xilinx FPGAs. The Threefish compression function of Skein is partially implemented using 4 unrolled rounds. These 4 rounds are then iteratively used to complete 72 rounds of compression function. The novel idea in implementation of these 4 unrolled rounds is that, we do not need separate MIX modules and multiplexers to select between different rotation constants in second step of MIX operation. We have efficiently implemented second step in MIX module using a LUT4 primitive depicted in Fig. 7(c). The select bit s, selects between two rotated instances of x_1, according to round number, to XOR with y_0. For first four rounds s is zero and upper half rows of rotation constants' table are used for respective MIX modules. For next four rounds s will be 1 and lower half rows of rotation constants' table are used for respective MIX modules. For example, $x_1 \ll 46$ will be selected and XORed with y_0 in first round and $x_1 \ll 39$ will be selected and XORed with y_0 in fifth round. Hardware architecture of key schedule module is shown in Fig. 7(b). The extended key K_8 is obtained by XORing the input 64-bit key words ($K_{0.....}K_7$) and constant C_{240}. The extended teak t_2 is obtained by XORing the two input 64-bit tweak word (t_0 and t_1). The extended key and tweak words are then loaded into the circular shift registers K (576 bit) and t (192 bit). These two registers are clocked and rotated once for each subkey. Key Schedule module generates subkeys on every falling edge of clock pulse. Add_Subkey module gives output on the rising edge

of each clock pulse. Next subkey is available on falling edge of the same clock pulse. In this way one clock cycle is required to complete four rounds, subkey addition and subkey generation. Therefore to complete 72 rounds and 19 subkey addition of Skein, 19 clock cycles will be required. The next chaining hash value will be available after 19 clock cycles.

6 Implementation Results

As mentioned earlier, for implementations and hardware performance evaluation of SHA-3 candidates, we aimed to target the latest and up-to-date FPGA technology from Xilinx. The latest 7 series release from Xilinx was of main interest. From this series we chose Virtex 7 for our implementations. We also implemented our designs on Virtex 6, latest before the release of 7 series in June 2010, and Virtex 5. Detailed device specifications are: Virtex 5 LX30T, speed grade 3, package FF323 (5vlx30tff323-3), Virtex 6 LX75T, speed grade 3, package FF784 (6vlx75tff784-3) and Virtex 7 285T, speed grade 3, package FG1157 (7v285tffg1157-3). The resulting clock frequencies and area utilization after place and route are reported. Table 2 shows achieved area consumption ($Area$), clock frequency (F_{max}), throughput (TP) and throughput per area (TPA) for implemented designs. The $Block\ Size$ is the block size of message in bits and N_{clk} is the number of clock cycles required for hash of a single message block.

Table 2. Implementation Results for 256-bit and 512-bit variants for SHA-3 Finalists

SHA-3 Finalist	Device	256-bit						512-bit					
		$Block\ Size$ [bits]	N_{clk} [cycles]	F_{max} [MHz]	$Area$ [Slices]	TP [Gb/s]	TPA [Mbps/slice]	$Block\ Size$ [bits]	N_{clk} [cycles]	F_{max} [MHz]	$Area$ [Slices]	TP [Gb/s]	TPA [Mbps/slice]
BLAKE	Virtex 5	512	28	125.55	1382	2.29	1.66	1024	32	100.02	2582	3.21	1.24
	Virtex 6	512	28	125.82	1104	2.30	2.08	1024	32	104.30	2246	3.34	1.46
	Virtex 7	512	28	137.14	1322	2.51	1.90	1024	32	115.01	2441	3.68	1.51
Grøstl	Virtex 5	512	10	121.03	1419	6.20	4.37	1024	14	101.22	2523	7.40	2.94
	Virtex 6	512	10	146.87	1467	9.62	5.12	1024	14	125.44	2359	9.17	3.89
	Virtex 7	512	10	175.65	1421	8.99	6.33	1024	14	155.02	3524	11.3	3.23
JH	Virtex 5	512	42	287.44	865	3.50	4.05	512	42	292.48	888	3.57	4.02
	Virtex 6	512	42	303.65	562	3.70	6.59	512	42	306.37	661	3.74	5.65
	Virtex 7	512	42	329.49	587	4.02	6.84	512	42	338.41	679	4.13	6.08
Keccak	Virtex 5	1088	24	275.56	1333	12.49	9.37	576	24	263.16	1197	6.32	5.28
	Virtex 6	1088	24	301.57	915	13.67	14.94	576	24	291.21	1015	6.99	6.89
	Virtex 7	1088	24	292.74	1161	13.27	11.43	576	24	254.91	1039	6.12	5.88
Skein	Virtex 5	512	19	113.78	1492	3.07	2.05	512	19	113.60	1544	3.06	1.98
	Virtex 6	512	19	114.30	1163	3.08	2.65	512	19	112.36	1203	3.03	2.52
	Virtex 7	512	19	127.73	1170	3.44	2.94	512	19	128.24	1244	3.46	2.78

One important observation from the results presented in Table 2 is that the achieved clock frequencies (F_{max}) on Virtex 7 for all designs are at higher end as compared with Virtex 5 and Virtex 6 results. However, most of the time designs on Virtex 7 consume more area which leads to a reduction in the overall TPA. This is somewhat unusual and unexpected. We can suggest possible reasons for this behavior. Firstly, Xilinx has introduced Virtex 7 for the very first time in ISE Design Suite 13.1, which we used for our implementations. There is a possibility that ISE 13.1 synthesis and implementation algorithms are not optimized for the Virtex 7 architecture. Secondly, the LUT primitives we used from Xilinx library are specific to Virtex 5 and Virtex 6 and the tool did not synthesize them correctly for Virtex 7. Xilinx has just released the documentation for Virtex 7 series FPGAs. A closer look at the Virtex 7 internal architecture and modification of all designs accordingly may lead to better results. We suggest this idea as a future work.

7 Comparison with Previous work

We have taken this opportunity to report SHA-3 candidates' hardware implementation results, for very first time, on latest Xilinx FPGAs. Before this no implementation results have been reported on Virtex 7 family. All reported work to date utilized the Virtex 6 family at most. We have achieved substantial improvements in implementation results from all of the previously reported work. Our results for Virtex 5, Virtex 6 and Virtex 7 are far ahead from all previously reported work in terms of throughput per area. But comparison on different devices is not justified due to various technological differences between these devices. Based on the comparison of Table 1 and Table 2, it is evident that our results for Virtex 6 and Virtex 5 are also exceeding most of the reported designs in terms of throughput per area.

8 Performance Comparison of SHA-3 Finalists

In evaluation of hardware performance of SHA-3 candidates in second round, NIST has considered throughput per area as a major deciding factor [33]. Keeping this criterion in mind we consider our results for each candidate, against each device, from Table 2. Fig. 8 and Fig. 9 represent the performance comparison of 256-bit and 512-bit variants, respectively, in a graphical view based on our results. It is clear from graphs that Keccak is far ahead of other four candidates, on all devices, in terms of throughput per area for both 256-bit and 512-bit variants. The difference is large for 256-bit variant; however, in case of 512-bit variants JH is very close to the performance of Keccak. In terms of area consumption JH leads all of the other candidates by consuming lesser area, for both variants. The area consumption difference from JH to other candidates is even more significant for 512-bit variants. In terms of throughput, again Keccak is far ahead for 256-bit digest sizes but Grøstl beats the Keccak with significant differences for 512-bit digest sizes on all devices. For all of the three comparison units, i.e. area, throughput and throughput per area, BLAKE and Skein are well behind the performance of Keccak, JH and Grøstl. BLAKE and Skein are computationally intensive designs as compared to other candidates. If we consider throughput per area as the major deciding factor for performance comparison, we can easily rank Keccak first and JH and Grøstl as second and third, respectively.

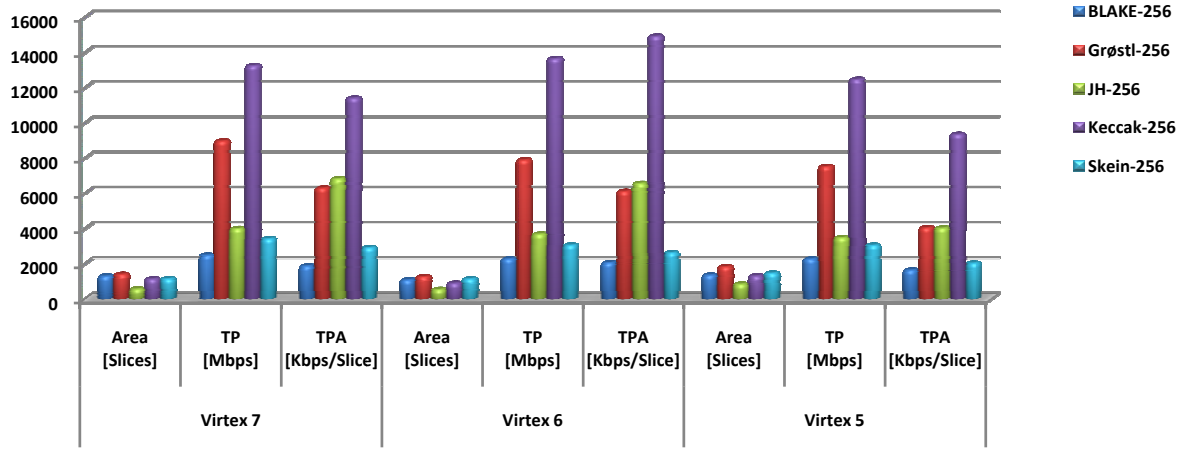


Fig. 8. Performance Comparison of 256-bit Variants of SHA-3 Finalists

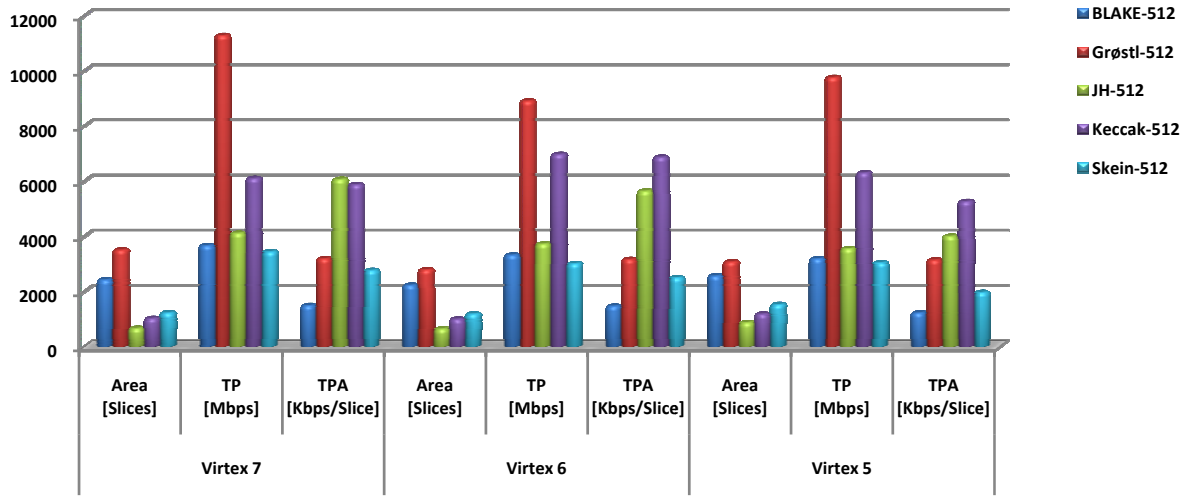


Fig. 9. Performance Comparison of 512-bit Variants of SHA-3 Finalists

9 Conclusion

In this work we have presented efficient hardware implementations of all 5 SHA-3 finalists. We reported the implementation results of 256-bit and 512-bit variants on most up-to-date Xilinx FPGAs i.e Virtex 6 and Virtex 7. We reported the performance figures of our implementations in terms of area, throughput and throughput per area and compared it with available results. We have achieved substantial improvements in implementation results from all of the previously reported work. Results achieved in this work are exceeding the various implementations reported so far. We compared and contrasted the performance figures of subject candidates on Virtex 5, Virtex 6 and Virtex 7. This work serves as performance investigation of SHA-3 finalists on most up-to-date FPGAs.

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, pp. 1-4, (2004), http://eprint.iacr.org/2004/199,
2. M. Szydlo: SHA-1 collisions can be found in 2^{63} operations. CryptoBytes Technical Newsletter, (2005)
3. M. Stevens: Fast collision attack on MD5. ePrint-2006-104, pp. 1-13, (2006), http://eprint.iacr.org/2006/104.pdf
4. Federal Register / Vol. 72, No. 212 / Friday, November 2, 2007 / Notices http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf
5. National Institute of Standards and Technology (NIST): Cryptographic Hash Algorithm Competition. http://www.nist.gov/itl/csd/ct/
6. J. Aumasson, L. Henzen, W. Meier, R. W. Phan, SHA-3 Proposal BLAKE version 1.3, pp. 1-79, (2010),.http://131002.net/blake/blake.pdf.
7. P. Gauravaram, L. R. Knudsen, K. Matusiewics, F. Mendel, C. Rechberger, M. Schlffer and S. S. Thomsen. SHA-3 Proposal Grøstl Version 2.0.1, (2011), http://www.groestl..info/
8. Hongjun Wu, The Hash Function JH, pp. 1-54, (2011), http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf
9. G .Bertoni, J. Daemen, M. Peeters, G. V. Assche: The Keccak SHA-3 Submission version 3, pp. 1-14, (2011), http://keccak.noekeon.org/Keccak-submission-3.pdf
10. N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas, J. Walker, The Skein Hash Function Family Version 1.3, pp. 1-100, (2010), http://www.skein-hash.info/sites/default/files/skein1.3.pdf.
11. Daniel J. Bernstein, "ChaCha, a variant of salsa20", Workshop Record of SASC 2008. The State of the Art Stream Ciphers, (2008), http://cr.yp.to/chacha.html#chacha-paper
12. J. Aumasson, L. Henzen, W. Meier, R. W. Phan, The hash function family LAKE, Proceedings of Fast Software Encryption (FSE), pp36-53 (2008)
13. J. Daemen and V. Rijmen, "The Design of Rijndael – AES Advanced Encryption Standard". Springer-Verlag Inc., New York USA (2002)

14. The SHA-3 Zoo Hardware Implementations, http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations
15. 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, pp. 1-18, August 23-24 (2010)
16. 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, pp. 1-15, August 23-24 (2010)
17. 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. 2nd SHA-3 Candidate Conference, Santa Barbara, pp. 1-14, August 23-24 (2010)
18. E. Homsirikamol, M. Rogawski, and K. Gaj, "Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates Using FPGAs," Cryptology ePrint Archive: Report 2010/445, available on line at http://eprint.iacr.org/2010/445.pdf
19. 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, pp. 1-15, May 19-20 (2011)
20. Jean-Luc Beuchat, Eiji Okamoto and Teppei Yamazaki, "Compact Implementations of BLAKE-32 and BLAKE-64 on FPGA", Cryptology ePrint Archive, Report 2010/173, (2010), http://eprint.iacr.org/2010/173.pdf
21. S. Kerckof, F. Durvaux, N. Charvillon, F. Regazzoni, G. Meurice and F. Standaert, "Compact FPGA Implementations of the Five SHA-3 Finalists", ECRYPT II Hash Workshop 2011, Tallinn, Estonia, pp. 1-19, May 19-20, (2011)
22. B. Jungk, "Compact Implementations of Grøstl, JH and Skein for FPGAs", ECRYPT II Hash Workshop 2011, Tallinn, Estonia, May 19-20, pp. 1-12 , (2011)
23. X. Guo, S. Huang, L. Nazhandali, and P. Schaumont. "On The Impact of Target Technology in SHA-3 Hardware Benchmark Rankings," Cryptology ePrint Archive: Report 2010/536, (2010), http://eprint.iacr.org/2010/536.pdf
24. L. Henzen, P. Gendotti, P. Guillet, E. Pargaetzi, M. Zoller and F.K. Gurkaynak, "Developing a hardware evaluation method for SHA-3 candidates," Proc. Cryptographic Hardware and Embedded Systems workshop, CHES 2010, Santa Barbara, pp. 248-263, Aug. (2010)
25. S. Tillich, et al. "High-Speed Hardware Implementations of Blake, Blue Midnight Wish, Cubehash, ECHO, Fugue, Grøstl, Hamsi, JH, Keccak, Luffa, Shabal, Shavite-3, SIMD, and Skein". Cryptology ePrint Archive, Report 2009/510, (2009). http://eprint.iacr.org/2009/510.pdf
26. Nicolas Sklavos and Paris Kitsos, BLAKE HASH Function Family on FPGA: From the Fastest to the Smallest, in *Proceedings of IEEE Computer Society Annual Symposium on VLSI (IEEE ISVLSI'10)*, (Kefalonia, Greece, 2010), pp. 1-4.
27. Joachim Strömbergson, Implementation of the Keccak Hash Function in FPGA Devices, pp. 1-4, (2008) http://www.strombergson.com/files/Keccak_in_FPGAs.pdf
28. Abdulkadir Akin, Aydin Aysu, Onur Can Ulusel, and Erkay Savas, Efficient Hardware Implementations of High Throughput SHA-3 Candidates Keccak, Luffa and Blue Midnight Wish for Single- and Multi-Message Hashing, 2nd SHA-3 Candidate Conference, Santa Barbara, pp. 1-12, August 23-24 (2010)
29. M. Long, Implementing Skein Hash function on Xilinx Virtex-5 FPGA platform, pp. 1-15, (2009), http://www.skein-hash.info/sites/default/files/skein_fpga.pdf
30. S. Tillich, Hardware implementation of the SHA-3 candidate skein, *ePrint-2009-159*, pp. 1-7, (2009), http://www.eprint.iacr.org/2009/159.pdf
31. Xilinx Virtex Family Documentation, available online at http://www.xilinx.com/support/documentation/
32. Kashif Latif, Arshad Aziz and Athar Mahboob, "Optimal Utilization of Available Reconfigurable Hardware Resources", Elsevier Computer & Electrical Engineering, Volume 37 Issue 6, pp. 1043-1057, doi:10.1016/j.compeleceng.2011.07.010, (2011)
33. Kris Gaj and Pawel Chodowiec, "Chapter 10: FPGA and ASIC Implementations of AES – Cryptographic Engineering", Editor: Cetin Kaya Koc, Springer (2009)
34. NIST Interagency Report 7764, Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition, pp. 1-38, (2011), http://csrc.nist.gov/groups/ST/hash/sha3/Round2/documents/Round2_Report_NISTIR_7764.pdf

View publication stats