Look-Up Table Based Implementations of SHA-3 Finalists: JH, Keccak and Skein

Kashif Latif[1], Arshad Aziz[1] and Athar Mahboob[2]
[1]National University of Sciences and Technology (NUST) H-12 Islamabad, Pakistan
[2]DHA Suffa University, Karachi, Pakistan
[email: kashif@pnec.edu.pk]
*Corresponding author: Kashif Latif

Received May 22, 2012; revised August 6, 2012; accepted August 23, 2012; published September 26, 2012

Abstract

Cryptographic hash functions are widely used in many information security applications like digital signatures, message authentication codes (MACs), and other forms of authentication. In response to recent advances in cryptanalysis of commonly used hash algorithms, National Institute of Standards and Technology (NIST) announced a publicly open competition for selection of new standard Secure Hash Algorithm called SHA-3. One important aspect of this competition is evaluation of hardware performances of the candidates. In this work we present efficient hardware implementations of SHA-3 finalists: JH, Keccak and Skein. We propose high speed architectures using Look-Up Table (LUT) resources on FPGAs, to minimize chip area and to reduce critical path lengths. This approach allows us to design data paths of SHA-3 finalists with minimum resources and higher clock frequencies. We implemented and investigated the performance of these candidates on modern and latest FPGA devices from Xilinx. This work serves as performance investigation of leading SHA-3 finalists on most up-to-date FPGAs.

Keywords: Authentication, SHA-3, JH, Keccak, Skein, Cryptographic Hash Functions, High Speed Encryption Hardware, FPGA.

http://dx.doi.org/10.3837/tiis.2012.09.024

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 in cryptanalysis, many have been found vulnerable and should not be used. A successful attack against a weakened variant of an algorithm weakens the experts' confidence, even though the hash function has never been broken, that leads to its rejection.

In previous few years, cryptanalysis of several hash algorithms has found serious vulnerabilities. In 2004, X. 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. 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. 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].

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. After reviewing and analyzing, the number of candidates reduced to 14 in Round 2 of the competition. A whole year was allocated for the public review, implementation and analysis of these algorithms and the Second SHA-3 Candidate Conference was held on August 23-24, 2010 in University of California, Santa Barbara. As a result of 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, advanced in 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 of three SHA-3 finalists, on latest FPGA technologies from Xilinx. The remainder of this paper is organized as follows. We briefly give an overview about cryptographic hash functions in section 2. Section 3 gives a brief description of selected SHA-3 finalists. Section 4 discusses some architecture related details of Xilinx's FPGA, concerned to our work. In section 5 we present the efficient hardware implementations of selected SHA-3 finalists. In section 6 we give the results of our work and present the analysis of achieved results. Section 7 provides the performance evaluation of SHA-3 finalists. Section 8 provides the comparison of our work with previously reported implementations. Finally, we provide some conclusions and direction for future work in section 9.

2. 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 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

$$H = h(M)$$

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

- 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'

Hash functions and encryption functions are different: encryption converts plain text into cipher text 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.

3. Brief Description of Selected SHA-3 Finalists

3.1 JH Hash Function

H. Wu designed and proposed the JH hash function for SHA-3 [7]. JH algorithm is based on the idea that large block ciphers can be constructed through small components and constant key. JH algorithm generalizes the AES design methodology to high dimensions. JH has four variants JH-224, JH-256, JH-384 and JH-512. Each variant is named as the output length of the hash. JH uses the same design for all variants. These variants only differ in initial values (IV) and output hash length. The compression function of JH is shown in **Fig. 1**. JH compression function is constructed from bijective function (a block cipher with constant key) [7]. 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 [7]. 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. In this sense, it can be viewed as 5-bit to 4-bit substitution. S-box substitutions are given at Table 1. The linear transformation L of JH implements a (4,2,3) Maximum Distance Separable (MDS) code over $\mathrm{GF}(2^4)$. The bit-wise computation of L is given as follows. Let A, B, C and D denote 4-bit words, the function $(\mathrm{C}, \mathrm{D}) = \mathrm{L}(\mathrm{A}, \mathrm{B})$ is computed as:

$$D^0 = B^0 \oplus A^1, \quad D^1 = B^1 \oplus A^2, \quad D^2 = B^2 \oplus A^3 \oplus A^0, \quad D^3 = B^3 \oplus A^0$$
$$C^0 = A^0 \oplus D^1, \quad C^1 = A^1 \oplus D^2, \quad C^2 = A^2 \oplus D^3 \oplus D^0, \quad C^3 = A^3 \oplus D^0$$

Where A^0 is most significant and A^3 is least significant bit of A. Same convention is used for B, C and D. The permutation P of JH is a combination of three small permutations (ø, P′ and π). In terms of hardware, permutations are achieved through simple rewiring of nets. Description of these permutations can be found in [7]. A sample two rounds of 64-bit state is shown in **Fig. 2**. Round constant bits are not shown in **Fig. 2**. Each JH round uses a 256-bit round constant. For 256 4-bit pairs of JH state, each round constant bit selects either S-box S_0 or S_1 at respective positions. Round constant for each round may be generated on the fly by using identical bijective compression function of JH in parallel to JH data path and setting initial constant value to zero. Alternately, round constants may be pre-computed and stored in memory to be used later. The initial value of JH state H^0 is computed by setting first two bytes of H^{-1} equal to the desired hash digest size and remaining bytes to zero, setting message M^0 to zero and then applying JH compression to H^{-1} and M^0 to obtain H^0. For an N block message final hash H^N is calculated by applying JH compression function to message blocks M^1, M^2, M^3,….., M^N iteratively as follows:

```
for $i = 1$ to $N$
    $H^i \leftarrow compress(H^{i-1}, M^i)$
return $H^N$
```

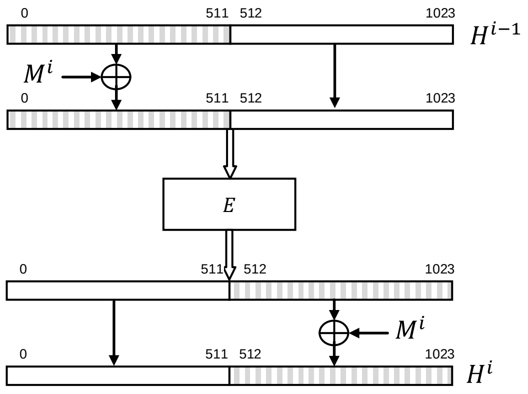


Fig. 1. JH compression function

Table 1. S-box substitutions of JH

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_0(x)$	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
$S_1(x)$	3	12	6	13	5	7	1	9	15	2	0	4	11	10	14	8

3.2 Keccak

G. Bertoni et al. designed and proposed the Keccak Hash Function for SHA-3 [8]. 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* determine 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 [8]. For 256-bit hash output *r* = 1088 and *c* = 512. The 1600-bit state of Keccak [1600] consists of 5x5 matrix of 64-bit words. Each compression step of Keccak consists of 24 rounds. Let us denote the state matrix with *A*. Each round then consists of following five steps:

Theta (θ):

$$C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4] \quad 0 \leq x \leq 4$$

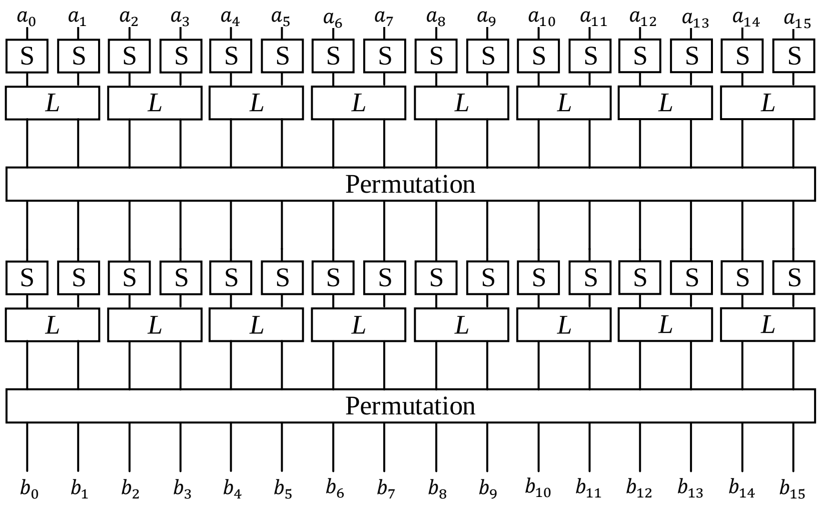

a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
S
L
Permutation
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8 b_9 b_{10} b_{11} b_{12} b_{13} b_{14} b_{15}

Fig. 2. Two sample rounds of JH using 64-bit state size

$$D[x] = C[x-1] \oplus ROT(C[x+1], 1) \quad 0 \le x \le 4$$
$$A[x,y] = A[x,y] \oplus D\,[x] \quad 0 \le x,y \le 4$$

Rho(ρ) and Pi (π):

$$B[y, 2x+3y] = ROT\,(A[x,y], r[x,y]) \quad 0 \le x,y \le 4$$

Chi (χ):

$$A[x,y] = B[x,y] \oplus \big((NOT\, B[x+1,y])\; AND\; B[x+2,y]\big) \;\; 0 \le x,y \le 4$$

Iota (i):

$$A[0,0] = A[0,0] \oplus RC$$

In above listed equations all operations within indices are done modulo 5. A denotes the complete permutation state array and $A[x,y]$ denotes a particular 64-bit word in that state. $B[x,y]$, $C[x]$ and $D[x]$ are intermediate variables. The symbol $\oplus$ denotes the bitwise XOR, NOT the bitwise complement and AND the bitwise AND operation. Finally, $ROT(W,r)$ denotes the bitwise cyclic shift operation, moving the bit at position i into position $i + r$ (modulo the lane size i.e. 64). The constants $r[x,y]$ and RC are cyclic shift offset and round constant respectively, and are defined in [8].

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 simply the state matrix is truncated to desired length of output hash. If more than r-bit (bitrate) hash value is required then more Keccak permutations are performed and their results concatenated until hash width reaches the desired length.

3.3 Skein

N. Ferguson et al. designed and proposed the Skein family of cryptographic hash functions for SHA-3 [9]. Skein has three different internal state sizes: 256, 512, and 1024 bits. Each of these state sizes can support any output size. Skein is built from these three components:

- **Threefish:** Threefish is the tweakable block cipher at the core of Skein, defined with a 256, 512 and 1024 bit block sizes.

- **Unique Block Iteration (UBI):** UBI is a chaining mode that uses Threefish to build a compression function that maps an arbitrary sized input to a fixed sized output.
- **Optional Argument System:** This allows Skein to support a variety of optional features without imposing any overhead on implementations and applications that do not use these features.

Threefish: Skein's compression function is based on Threefish, which is a large tweakable block cipher [9]. Tweakable block ciphers are ciphers that take three inputs: a key, a tweak and a block of message, instead of the usual block ciphers that take two inputs, a key and a block of message. A unique tweak value is used to encrypt every block of message. Different tweaks create different permutations for each encryption process. This technique eliminates the need for altering keys if we want to have a different block cipher every time.

The block and key sizes of Threefish are equal and can be set to 256, 512 or 1024 bits, and they are designated as: Threefish-256, Threefish-512, and Threefish-1024, respectively. The tweak value is 128 bits for all block sizes. Threefish structural design consists of round operations. Threefish-256 and Threefish-512 compression functions are made of 72 consecutive round operations while the Threefish-1024 requires 80 rounds. Each round of the Threefish-256 block cipher is made of two instances of a MIX function along with a permutation module, while a round key is added to the data before the first round and after each 4 consecutive rounds as shown in **Fig. 3**.

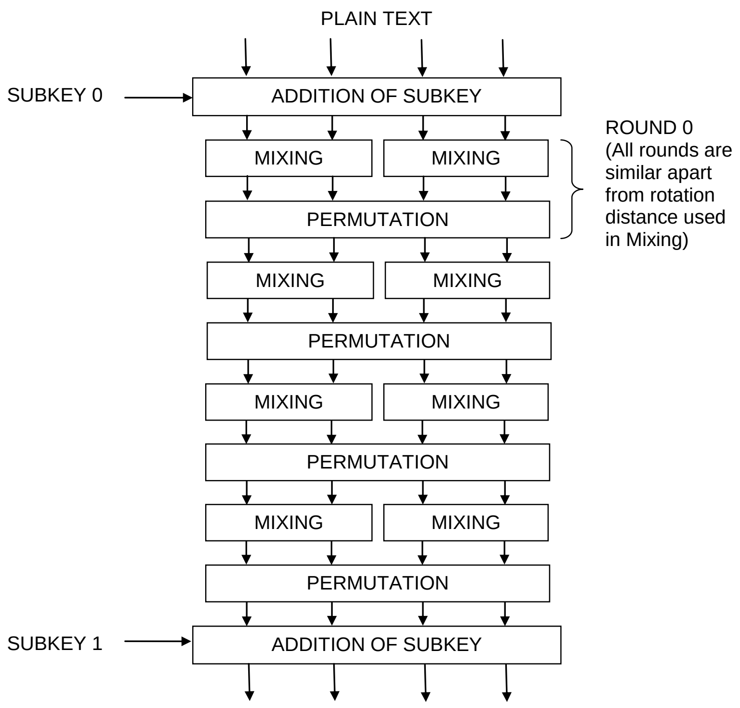


Fig. 3. First Four Round Operations of the Threefish-256 Cipher

Subkeys or round keys consist of three contributions: an input key word, tweak words, and a counter value. The key schedule turns the key and tweak words into a sequence of subkeys, each of which is equal to the size of the block. Tweak depends upon number of factors including position and the bit length of the message block.

The mix operation consists of addition modulo 2^{64}, XORs and left-rotates. These operations are defined on the intermediate state organized in 64-bit words. The MIX operation transforms two of these 64-bit words and is common to all Threefish variants. MIX function has two input words ($X0$ and $X1$) and produces two output words ($Y0$ and $Y1$) using the following relations:

$$Y0 = (X0 + X1)\ mod\ 2^{64}$$
$$Y1 = (X1 \lll R) \oplus Y0$$

Where $\oplus$ is the bit-wise XOR operation and $\lll$ is the left rotate operator and R (Rotation Distance) is a constant value which depends on the Threefish block size, the round index and the position of the two 64-bit words in the Threefish block [9]. All Threefish rounds are similar apart from rotation constant in mixing operation. These rotation constants are defined in [9]. The subsequent permutation operation reorders 64-bit words constructed from a Threefish block. This permutation is fixed for a specific Threefish variant, defined in [9].

Unique Block Iteration (UBI) Construction:

The UBI construction is a variant of the Cascade or (Merkle-Damgård) construction. It uses a tweakable block cipher in Matyas-Meyer-Oseas mode to form a compression function, and uses the bit offset of the block being hashed as the tweak [9]. An example of UBI mode is shown in **Fig. 4**.

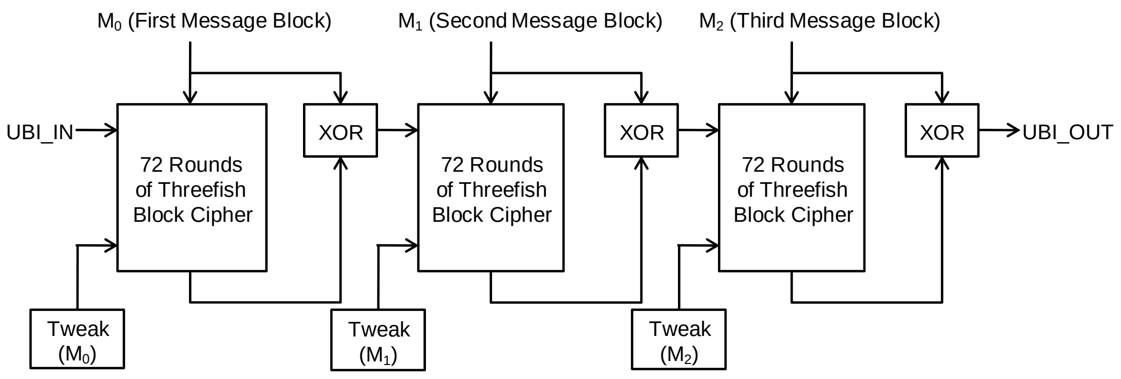

M_0 (First Message Block)
M_1 (Second Message Block)
M_2 (Third Message Block)
UBI_IN
72 Rounds of Threefish Block Cipher
XOR
72 Rounds of Threefish Block Cipher
XOR
72 Rounds of Threefish Block Cipher
XOR
UBI_OUT
Tweak (M_0)
Tweak (M_1)
Tweak (M_2)

Fig. 4. Unique Block Iteration Construction

The message M, shown in **Fig. 4**, comprises of three message blocks M_0, M_1 and M_2. UBI_IN is the first Threefish encryption key which is used along with the tweak value for the encryption of first message block. The output of the Threefish block cipher is XORed with message block itself and its output along with new tweak value is used for the encryption of the next block of message. It means that a new key is used for the encryption of each block. As mentioned earlier, the tweak values depend on the position and bit length of the respective message block. UBI is used in Skein not only for compression and the output transformation, but also for other optional operation modes e.g. tree hashing and keyed hashing.

4. FPGA Specific Features and Their Implication on SHA-3 Algorithm Architectures

The architectures of latest FPGA families from Xilinx (Virtex 5, Virtex 6, and Spartan 6) are based on 6-input LUTs, named LUT6 [10]. 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. 5(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. 5(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. 5(a)**

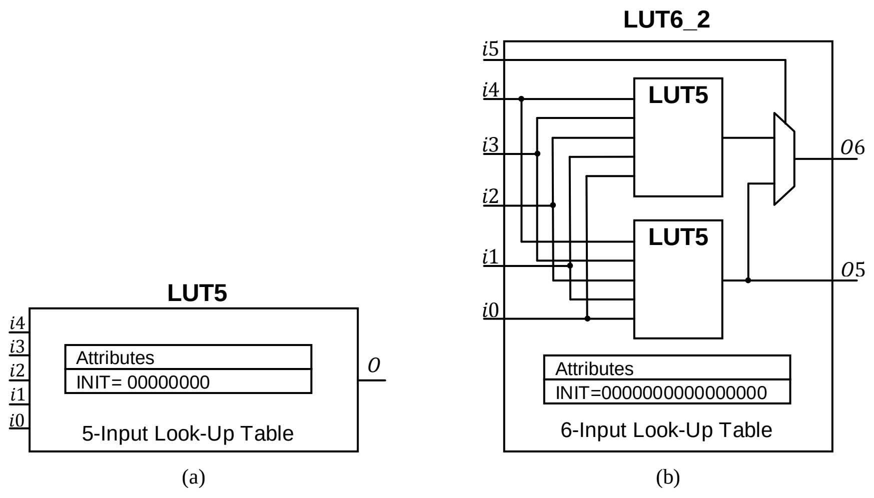


Fig. 5. LUT5 and LUT6_2 primitives in Xilinx HDL library

and **5(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 [11] for efficient utilization of modern FPGA resources.

5. Implementation

We have implemented the core functionality of 256-bit and 512-bit variants of JH, Keccak and Skein. Core functionality does not mean that we have implemented compression function only. 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 approach by cattering for the following constraints:

- Common Environment: It is concerned with the implementations, in terms of the level of expertise, language, coding techniques, design methodology, and development tools. We assured it by keeping: common implementer for all candidates, using Verilog as common language, common design methodology (discussed in next point) and using Xilinx's ISE 13.1 as the common development tool.
- Design Methodology: For fair comparison it is necessary to utilize same set of hardware resources 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. Memories are also implemented using distributed RAMs/ROMs because they utilize the LUT resources and memory requirement of a candidate will be 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 do 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.

5.1 Common I/O Interface

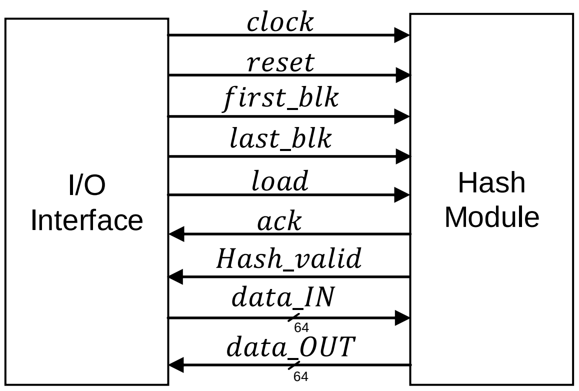
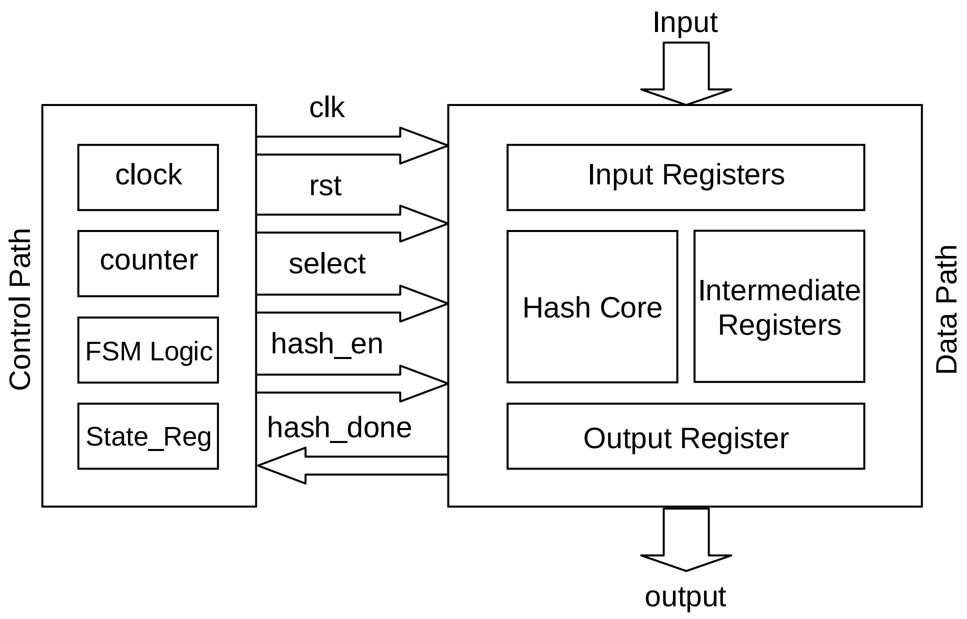


Fig. 6. Common Input/Output Interface

Developed input/output interface is shown in **Fig. 6**. All I/O transactions are synchronized. Each I/O is sampled at the rising edge of clock cycle. The *first_blk* and *last_blk* signals indicate the first and last block of message, respectively. The input cycle is initiated by I/O interface by putting *load* signal to high. Hash Module acknowledges the request if it is able to receive data by putting *ack* signal to high. After receiving acknowledgment, I/O interface makes available 64-bit word of data at each rising edge of clock cycle. During the transaction of data, *ack* signal remains at logic high. After receiving desired amount of input words Hash Module puts 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.

Fig. 7. Hash Module separated in Control and Data paths



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 of hash module. That is, we can make I/O transactions at the same time while hash of a message block is in progress.

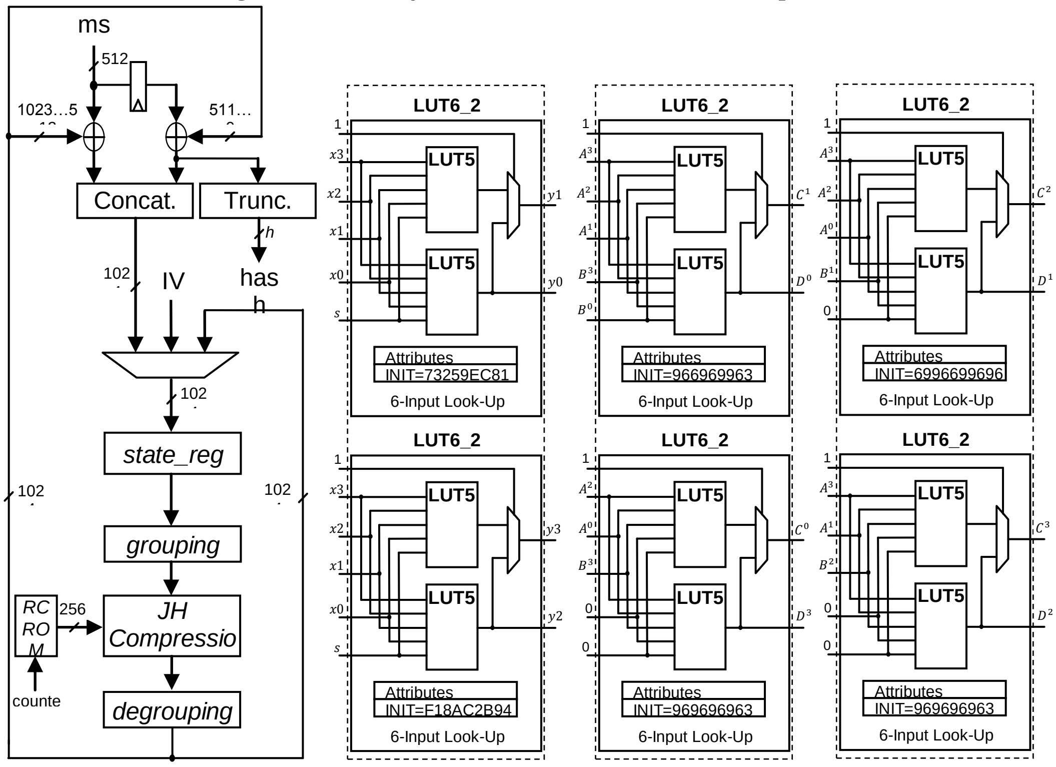
5.2 Control and Data Paths

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. 7**. 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.

5.3 Implementation of JH

The data path implemented for JH is shown in **Fig. 8(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. Hence, same data path is utilized for all hash digest sizes. Only the difference between data path for different hash -



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

Fig. 8. Architectural detail of JH

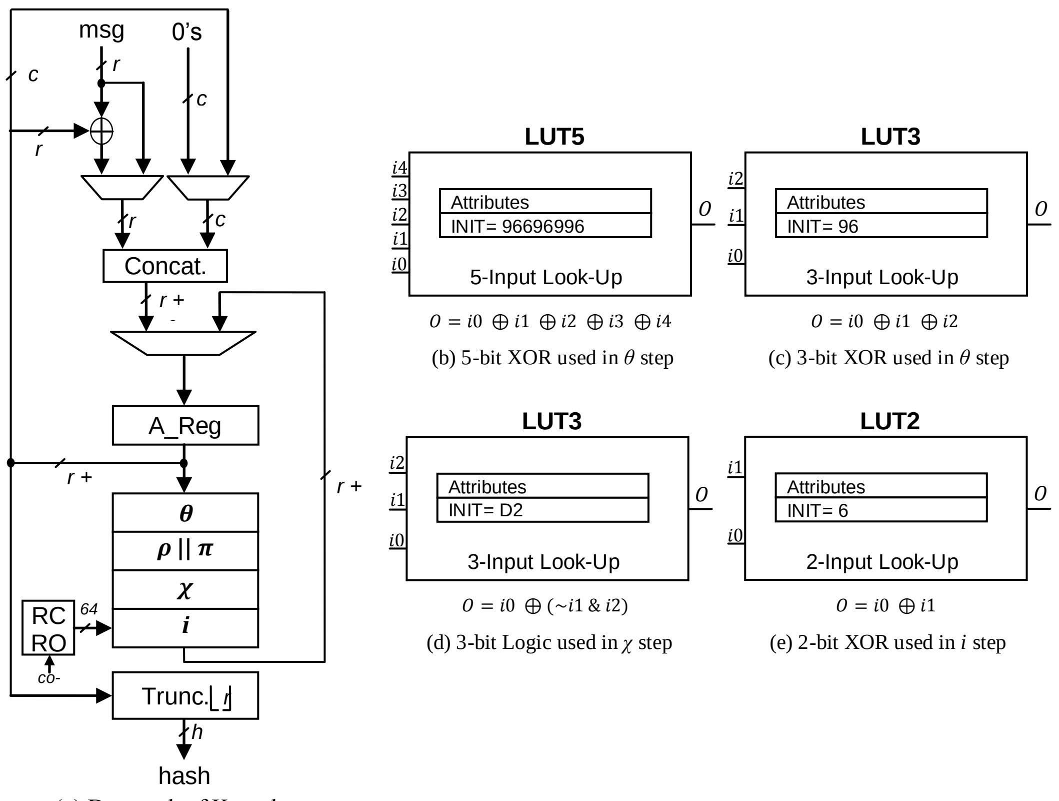
sizes is of initial values (IV) and hash output registers. In the beginning of every hash process *state_reg* is initialized with IV of desired hash digest size. Then a complete JH compression is processed by setting msg and round constant RC to zero. The higher order 512 bits of resulting state of JH 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. 8(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 [7]. 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. 5(b)**) and used both of its outputs 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. 8(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. 8(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. 8(c)** as denoted in linear transformation equations in specification document [7].

5.4 Implementation of Keccak

The data path implemented for Keccak is shown in **Fig. 9(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. 9(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. 9(a)** represents the concatenation operation. Compression function of Keccak consists of five steps. In **Fig. 9(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 ($\boldsymbol{\theta}$) 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. 9(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. 9(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 Keccak

(b) 5-bit XOR used in θ step

(c) 3-bit XOR used in θ step

(d) 3-bit Logic used in χ step

(e) 2-bit XOR used in i step

Fig. 9. Architectural detail of Keccak

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.

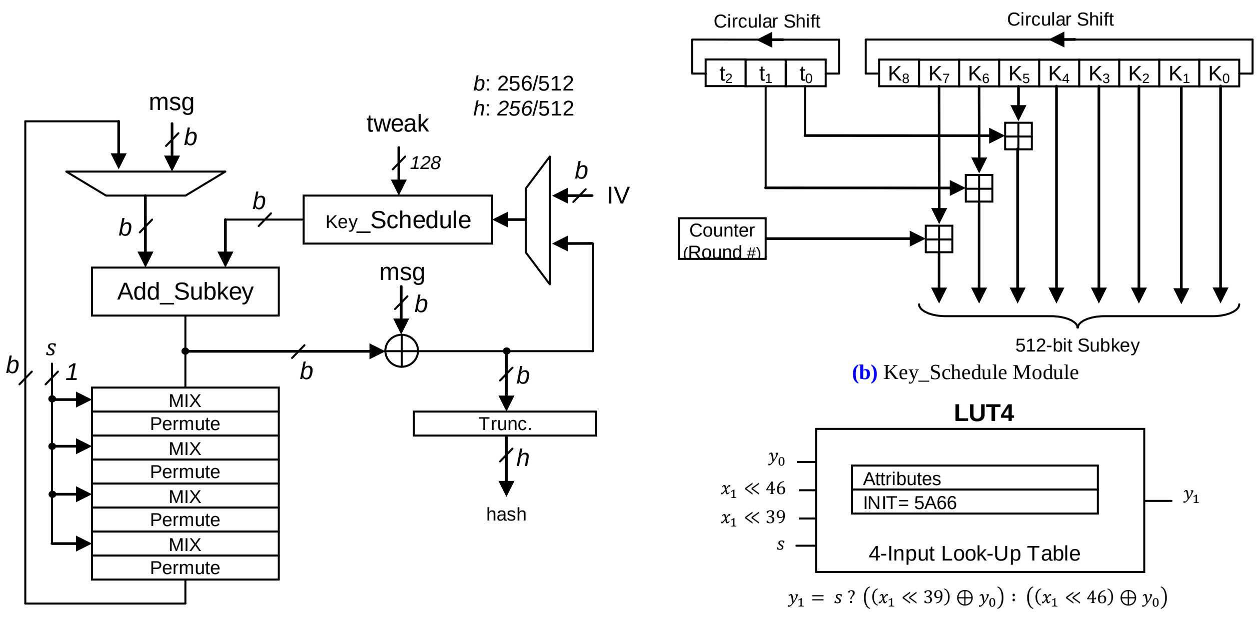
Chi (χ) Step: In χ step three logical operations XOR, NOT and AND are used. These are implemented using LUT3 primitive as shown in **Fig. 9(d)**. In order to accomplish the χ step, LUT3 with χ logic is instantiated 25x64 times.

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. 9(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 Implementation of Skein

The data path implemented for Skein is shown in **Fig. 10(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



(b) Key_Schedule Module

(a) Data path of Skein

(c) Selection between two rotation constants in MIX operation

Fig. 10. Architectural detail of Skein

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. 10(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. 10(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 tweak 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

The designs have been implemented on Xilinx Virtex 5 and Virtex 6. Detailed device specifications are: Virtex 5 LX30T, speed grade 3, package FF323 (5vlx30tff323-3) and Virtex 6 LX75T, speed grade 3, package FF784 (6vlx75tff784-3). 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. In order to complete the profile of SHA-3 candidates, results of BLAKE and Grøstl are included from [15] and [16], respectively. However, we would like to point out that BLAKE and Grøstl implementations did not benefit from the use of Lookup Table based design. This can be attributed to the fact that JH, Keccak and Skein utilize boolean functions extensively in the compression transformation whereas BLAKE utilizes primitives which make use of arithmetic operations. BLAKE and Grøstl are algorithms where use of specific library resources not turns into advantageous outcomes. For example, 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.

Table 2. Implementation Results for 256-bit and 512-bit variants of 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	124.55	1739	2.28	1.31	1024	32	100.02	2582	3.21	1.24
	Virtex 6	512	28	131.96	1602	2.41	1.51	1024	32	104.30	2246	3.34	1.46
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
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
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
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

For both Virtex 5 and Virtex 6, designs of JH and Keccak result in higher frequencies, due to their simple compression functions. However, more number of rounds make JH algorithm's throughput restricted to the lower end. The larger message block size and average number of rounds of Keccak make it the top performer in terms of both throughput and throughput per area. In terms of area consumption, JH is the most compact algorithm. BLAKE and Skein are computationally rigorous algorithms as compared to other candidates, hence, results in low throughput and throughput per area designs.

7. Evaluation of SHA-3 Finalists

It is clear from results that Keccak is far ahead of other four candidates, on both Virtex 5 and Virtex 6, 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. For 256-bit variants JH and Grøstl give almost similar throughput per area performance. 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 both devices. For throughput and throughput per area, BLAKE and Skein are well behind the performances 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.

8. Comparison with previous work

We have achieved significant improvements in implementation results from all of the previously reported work. For JH, Keccak and Skein we take advantage of Look Up Table (LUT) resources, available on Xilinx FPGAs, to reduce chip area consumption and critical paths. The use of resource primitives from Xilinx specific libraries allowed us to design high frequency designs with minimum use of resources. Table 3 shows the comparison of results with previously reported implementations in terms of F_{max}, area and throughput per area. In Table 3, we show our exceeding results in bold font. Most of our results for Virtex 5 and Virtex 6 are exceeding from all previously reported work in terms of throughput per area. Only the JH is the case where our throughput per area results are slightly behind of [17]. However, our result figures are very close to throughput per area results of [17] with exceptional use of smaller area for JH. In case of Keccak and Skein algorithms our throughput per area results are ahead of previously reported work. Comparison of BLAKE and Grøstl has been presented in [15] and [16] respectively; therefore, their comparison is not included in Table 3.

9. Conclusion and Future Work

In this work we have presented efficient hardware implementations of SHA-3 finalists: JH, Keccak and Skein. We have reported the implementation results of 256-bit and 512-bit variants on Xilinx FPGAs Virtex 5 and Virtex 6 in terms of area, throughput and throughput per area; and compared it with previous results. Utilization of Look-Up Table (LUT) resources on FPGAs proves beneficial to enhance the hardware performance of the JH, Keccak and Skein SHA-3 candidates in terms of both speed and area. Use of LUT primitives is, therefore, demonstrated to be a justified design approach. However, for BLAKE and Grøstl SHA-3 candidates, further effort is required to be able to extract performance gains from LUT based implementation. This shows that SHA-3 finalists offer implementors different tradeoffs in FPGA based implementations. We intend to further explore the possibilities of LUT based implementations for BLAKE and Grøstl which may prove competitive relative to non LUT designs.
We have achieved significant improvements in implementation results compared to the previously reported work. Results achieved in this work are exceeding the various implementations reported so far. This work serves as performance investigation of SHA-3 finalists on modern FPGAs.

Table 3. Comparison with previous work. F_{max} in MHz, $Area$ in Slices and TPA in Mbps/Slice

SHA-3 Finalists	Author(s)	Device	256-bit			512-bit		
			F_{max}	$Area$	TPA	F_{max}	$Area$	TPA
JH	**Our work**	Virtex 5	287.44	**865**	4.05	292.48	**888**	4.02
	Our work	Virtex 6	303.65	**562**	6.59	306.37	**661**	5.65
	Baldwin et al.[12]	Virtex 5	144.11	1763	0.93	144.11	1763	0.93
	Matsuo et al. [13]	Virtex 5	201.00	2661	0.84	-	-	-
	Gaj et al. [14]	Virtex 5	278.09	1108	3.06	275.48	1165	2.88
	Homsirikamol et al. [17]	Virtex 6	-	847	6.73	-	896	5.95
	Homsirikamol et al. [17]	Virtex 5	-	909	5.09	-	1020	4.64
Keccak	**Our work**	Virtex 5	275.56	1333	**9.37**	263.16	**1197**	5.28
	Our work	Virtex 6	301.57	**915**	**14.94**	291.21	**1015**	**6.89**
	Keccak Team [8]	Virtex 5	122.00	1330	3.91	-	-	-
	Strömbergson [18]	Spartan3A	85.00	3393	1.41	-	-	-
	Strömbergson [18]	Virtex 5	118.00	1483	4.52	-	-	
	Baldwin et al.[12]	Virtex 5	195.73	1971	3.17	195.73	1971	4.32
	Matsuo et al. [13]	Virtex 5	205.00	1433	5.86	-	-	-
	Akin et al. [19]	Spartan 3	81.40	2024	1.71	-	-	-
	Akin et al. [19]	Virtex-II	136.60	2024	2.87	-	-	-
	Akin et al. [19]	Virtex 4	142.90	2024	3.00		-	-
	Gaj et al. [14]	Virtex 5	238.38	1229	8.79	276.86	1236	5.37

	Homsirikamol et al. [17]	Virtex 6	-	1165	10.17	-	1231	5.87
	Homsirikamol et al. [17]	Virtex 5	-	1395	9.16	-	1220	5.37
Skein	**Our work**	Virtex 5	113.78	1492	**2.05**	113.60	1544	1.98
	Our work	Virtex 6	114.30	**1163**	**2.65**	112.36	**1203**	**2.52**
	Baldwin et al. [12]	Virtex 5	-	-	-	83.58	2756	0.35
	Matsuo et al. [13]	Virtex 5	115.00	854	1.64	-	-	-
	Gaj et al. [14]	Virtex 5	116.35	843	1.86	104.34	1520	1.85
	Long [20]	Virtex 5	114.94	931	0.44	114.94	1758	0.46
	Tillich [21]	Virtex 5	68.40	937	1.87	69.04	1632	2.17
	Tillich [21]	Spartan 3	26.14	2421	0.28	26.66	4273	0.32
	Homsirikamol et al. [17]	Virtex 6	-	1510	2.17	-	1591	1.96
	Homsirikamol et al. [17]	Virtex 5	-	1728	1.70	-	1658	1.7

References

[1] X. L. X. 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. Article (CrossRef Link).

[2] M. Szydlo, "SHA-1 collisions can be found in 2^{63} operations," *CryptoBytes Technical Newsletter*, 2005. Article (CrossRef Link).

[3] M. Stevens, "Fast collision attack on MD5," *Cryptology ePrint Archive*, Report 2006/104, pp. 1-13, 2006. Article (CrossRef Link).

[4] Federal Register / Vol. 72, No. 212 / Friday, November 2, 2007 / Notices, pp. 1-9, 2007. Article (CrossRef Link).

[5] National Institute of Standards and Technology (NIST), "cryptographic hash algorithm competition," 2007. Article (CrossRef Link).

[6] NIST Interagency Report 7764, "Status report on the second round of the SHA-3 cryptographic hash algorithm competition," pp.1-38, 2010. Article (CrossRef Link).

[7] H. Wu., "The hash function JH," pp.1-54, 2011. Article (CrossRef Link).

[8] G .Bertoni, J. Daemen, M. Peeters, G. V. Assche, "The KECCAK SHA-3 Submission version 3," pp.1-14, 2011. Article (CrossRef Link).

[9] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas and J. Walker, "The Skein hash function family version 1.3," pp.1-100, 2011. Article (CrossRef Link)

[10] Xilinx Virtex Family Documentation, available online at Article (CrossRef Link).

[11] K. Latif, A. Aziz and A. Mahboob, "Optimal utilization of available reconfigurable hardware resources," *Elsevier Computer & Electrical Engineering*, vol.37, pp.1043-1057, 2011. Article (CrossRef Link).

[12] 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*, pp.1-18, Aug.2010. Article (CrossRef Link).

[13] 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*, pp.1-15, Aug.2010. Article (CrossRefLink).

[14] K. Gaj, E. Homsirikamol, and M. Rogawski, "Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates using FPGAs," *Proc. Cryptographic Hardware and Embedded Systems workshop*, CHES 2010, Santa Barbara, 2010. Article (CrossRef Link).

[15] K. Latif, A. Mahboob and A. Aziz, "High Throughput Hardware Implementation of Secure Hash Algorithm (SHA-3) Finalist – BLAKE," in *Proc. of 9th International Conference on Frontiers of Information Technology, IEEE Computer Society*, pp.189-194. 2011. Article (CrossRef Link).

[16] M. M. Rao, K. Latif, A. Aziz, and A. Mahboob, "Efficient FPGA Implementation of Secure Hash Algorithm Grøstl - SHA-3 Finalist," In *Emerging Trends and Applications in Information Communication Technologies*, vol.281, pp.361-372, 2012, Article (CrossRef Link).

[17] 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*, pp.1-15, 2011. Article (CrossRef Link).

[18] J. Strömbergson, "Implementation of the Keccak Hash Function in FPGA Devices," 2010. Article (CrossRef Link).

[19] A. Akin, A. Aysu, O. C. Ulusel and E. 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*, pp.1-12, Aug.2010. Article (CrossRef Link).

[20] M. Long, "Implementing Skein Hash function on Xilinx Virtex-5 FPGA platform," pp.1-15, 2009. Article (CrossRef Link).

[21] S. Tillich, "Hardware implementation of the SHA-3 candidate skein," *Cryptology ePrint Archive*, Report 2009/159, pp. 1-7, 2009. Article (CrossRef Link).

Kashif Latif obtained B.E. in Industrial Electronics from N.E.D. University of Engineering and Technology, Karachi, Pakistan and M.S. degree in Electrical Engineering from National University of Sciences and Technology, Pakistan in 2002 and 2008, respectively. He has been involved in various R&D assignments since 2002. Presently he is Ph.D. candidate at National University of Sciences and Technology, Pakistan. His research interests include Information Security and Cryptography, FPGA based Systems Designs, Hardware Solutions of Cryptographic Applications and Digital Systems Design.

Arshad Aziz obtained B.E. and M.E. degrees in Computer Engineering from Sir Syed University of Engineering and Technology, Karachi, Pakistan in 1998 and 2002, respectively. He obtained his Ph.D. in Electrical Engineering from National University of Sciences and Technology, Pakistan in 2007. He is currently an Associate Professor in Electrical Engineering at the National University of Sciences and Technology, Pakistan. His research interests include Computer and Network Security, Cryptography, Computer Networks and Internetworking, TCP/IP Protocol suite, FPGA Based Systems Design, Computer Architectures and the Operating Systems.

Athar Mahboob obtained B.S. and M.S. degrees in Electrical Engineering from Florida State University at Tallahassee, Florida, USA in 1992 and 1995, respectively. He obtained his Ph.D. in Electrical Engineering from National University of Sciences and Technology, Pakistan in 2005. He is currently a Professor and Head of Department of Electrical Engineering at the DHA Suffa University, Pakistan. His research interests include implementing Enterprise Information Services using Linux, Information Security and Cryptology, Computer Networks and Internetworking using TCP/IP Protocols, Digital Systems Design and Computer Architectures.