Hardware Performance Evaluation of SHA-3 Finalists - Blake, Keccak and Skein

K. Latif*, A. Aziz and A. Mahboob
National University of Sciences and Technology (NUST), Islamabad, Pakistan

RECIEVED 27.12.2011
REVISED 26.03.2012
ACCEPTED 26.03.2012

*Corresponding Author: K. Latif
E-mail: kashif@pnec.edu.pk

KEYWORDS

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

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, NIST USA 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 implementations of candidates. In this work we present efficient hardware implementations and corresponding performance evaluations of three final round candidates of SHA-3: Blake, Keccak and Skein. We implemented and investigated the performance of these candidates on modern and latest FPGA devices from Xilinx. We show our results for most recently released devices on which implementations have not been reported yet. This work serves as performance investigation of leading SHA-3 finalists on most up-to-date FPGAs.

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, Wang et al presented the collisions for MD4, MD5, HAVAL-128 and RIPEMD (Wang, et al., 2004). There was a breakthrough in cryptanalysis of SHA-1 Hash Algorithm in August 2005. Szydlo found that it is possible to find a collision in SHA-1 in 263 operations (Szydlo, 2004). Previously, it was thought that 280 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 263 operations. No attacks have yet been reported on the SHA-2 variants; however they are algorithmically similar to SHA-1. Furthermore, Stevens reported a collision attack on MD5 in 2006 (Stevens, 2006).

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 (FR, 2007) 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 review and analysis these candidates were 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 (NIST, 2007).

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. Section 2 gives brief description of selected SHA-3 finalists. In section 3 we present the efficient hardware implementations of these algorithms. In section 4 we give the results of our work and compare it with available implementations in section 5. Section 6 presents performance evaluation of selected SHA-3 finalists. Finally, we provide some conclusions and directions for future work in Section 7.

Brief Description of Selected SHA-3 Finalists

1-BLAKE

Aumasson et al. designed and proposed the BLAKE Hash family for SHA-3 (Aumasson, et al., 2007). BLAKE is

based on Bernstein's stream cipher ChaCha and uses iteration mode HAIFA. Figure 1 shows the construction of BLAKE's compression function. BLAKE-224 and BLAKE-256 operate on 32-bit words while BLAKE-384 and BLAKE-512 operate on 64-bit words. The inner state of the compression function is represented as a 4×4 matrix of words. We briefly explain here the functionality of BLAKE-256 compression function.

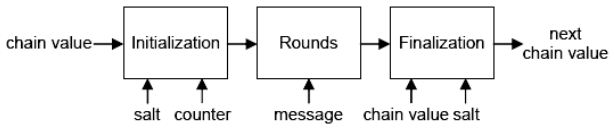

Figure (1) The local wide-pipe construction of BLAKE's compression function

The compression function takes four input values: a chaining hash value $h = h_0, \ldots, h_7$, a message block, a salt $s = s_0, \ldots, s_3$, and a counter $t = t_0, t_1$. Each word is 32-bit long. Additionally, compression function utilizes 32-bit constants $c_0, \ldots, c_{15}$ and permutation $\sigma_r\{0, \ldots, 15\}$ (Aumasson, *et al.*, 2007). The output of compression function is a new chain value $h' = h'_0, \ldots, h'_7$ We can write compression function as $h' = compress(h, m, s, t)$

Initialization

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

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

Round Function

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

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

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

$$a = a + b + \left(m_{\sigma_r(i)} \oplus c_{\sigma_r(i+1)}\right)$$
$$c = c + d$$
$$a = a + b + \left(m_{\sigma_r(i+1)} \oplus c_{\sigma_r(i)}\right)$$
$$c = c + d$$

Where $\oplus$ is the bit-wise XOR operation, $\gg$ is the right rotate operator and $\oplus$ is a permutation, indexed by round number r and type of G_i function $(i = 0,2,4, \ldots 14)$. Here we specified the alternate G functions as described in (Aumasson, *et al.*, 2007). For actual specifications please refer to (Aumasson, *et al.*, 2007). The first four G functions G_0, G_2, G_4 and G_6 can be computed in parallel, because each of them updates a distinct column of the matrix. This is referred to as a *column step*. Likewise, the last four G functions G_8, G_{10}, G_{12} and G_{14} update distinct diagonals of the matrix and thus can also be computed in parallel. This is referred to as a *diagonal step*. At round , the permutation used is $\sigma_{r \bmod 10}$, for example, in the last round and the permutation $\sigma_{13 \bmod 10} = \sigma_3$ is used.

Finalization

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

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

Iterated hash

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

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

Keccak

Bertoni et al. designed and proposed the Keccak Hash Function for SHA-3 (Bertoni, *et al.*, 2007). 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 (Bertoni, *et*

al., 2007). 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. Each round then consists of following five steps:

Theta (θ):

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

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

Rho(ρ) and Pi (π):

Chi (χ):

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 (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 (Bertoni, *et al.*, 2007).

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.

2.3 Skein

Ferguson et al designed and proposed the Skein family of cryptographic hash functions for SHA-3 (Ferguson, *et al.*, 2007). 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 (Ferguson, *et al.*, 2007). 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 function is 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. 2. 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) and produces two output words ($Y0$ and) using the following relations:

$$Y0 = (X0 + X1) \bmod 2^{64}$$

$$Y1 = (X1 \ll R) \bigoplus \square \square 0$$

Where $\oplus$ is the bit-wise XOR operation and $\ll$ $\ll$ 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 (Ferguson, *et al.*, 2007). All Threefish rounds are similar apart from rotation constant in mixing operation. These rotation constants are defined in (Ferguson, *et al.*, 2007). The subsequent permutation operation reorders 64-bit words constructed from a Threefish block. This permutation is fixed for a specific Threefish variant, defined in (Ferguson, *et al.*, 2007).

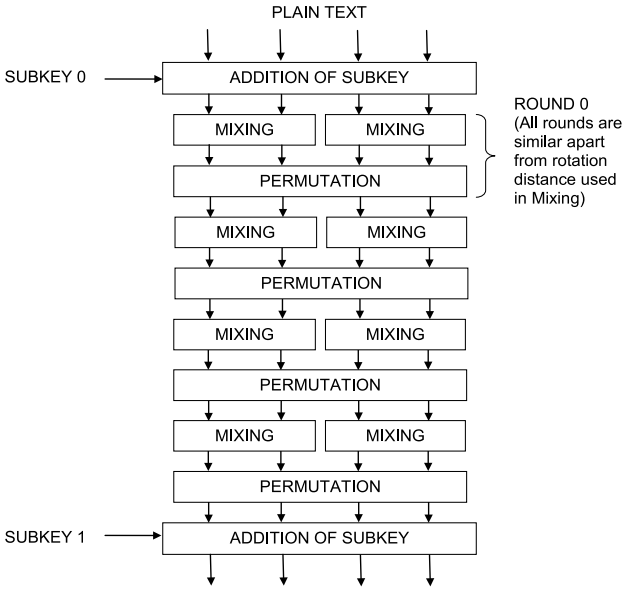

Figure (2) First Four Round Operations of the Threefish-256 Cipher

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 (Ferguson, *et al.*, 2007). An example of UBI mode is shown in Fig. 3. The message M, shown in Fig. 3, 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, keyed hashing).

3. Implementation

We have implemented the core functionality of 256-bit variants of BLAKE, Keccak and Skein. Core funtionality 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 follwing constraints:

- Common Environment: It is concerned with 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 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 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. Memories are also implemented using distributed RAMs/ROMs because they utilize the LUT resources and memory requiremet 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 re-using 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.

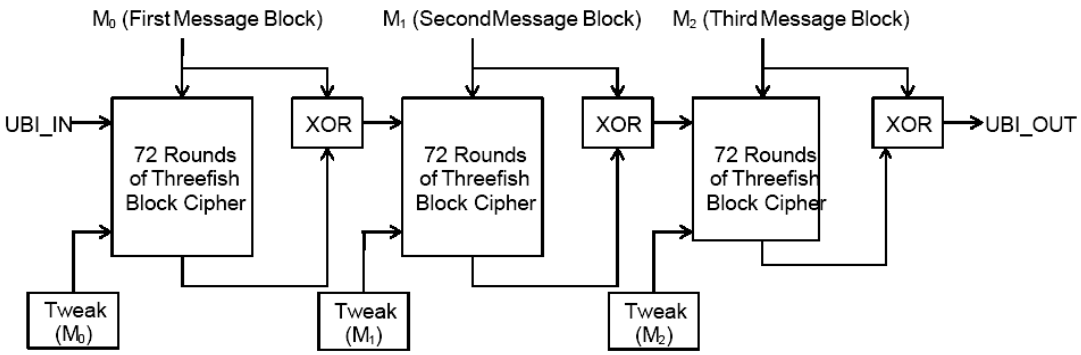

Figure (3) Unique Block Iteration Construction

3.1 Common I/O Interface

Developed input/output interface is shown in Fig. 4. All I/O transactions are synchronized. Each I/O is sampled at the rising edge of clock cycle. 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 make 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 put 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 of hash module. That is, we can make I/O transactions at the same time while hash of a message block is in progress.

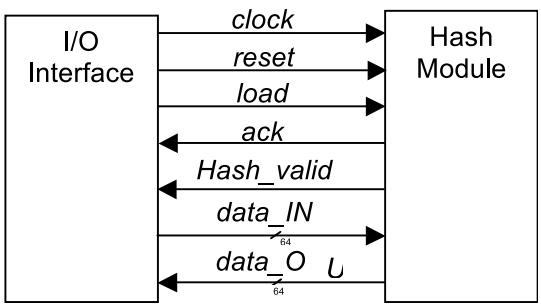

Figure (4) Common Input/Output Interface

3.2 Implementation of BLAKE

Implemented data path for BLAKE is shown in Fig. 5. All nets represent data width of 512-bits. The V_Reg represents the V matrix register, on which processing of BLAKE algorithm takes place. The CV_Reg stores the intermediate chaining hash values. Initialization module initializes the V_Reg by taking IV (Initial Value) or chaining hash value as input. Core functionality of BLAKE algorithm is represented by G Function module. Four instantiations of G function module are utilized to compute 4 G operations in parallel. These instantiations are represented as G1, G2, G3 and G4. Each G function instance computes a different G function on alternate clock cycles. G1 instance computes G0 and G8, G2 computes G2 and G10, G3 computes G4 and G12 and similarly G4 computes G6 and G14 on alternate clock cycles. G Function module is implemented using pure combinational logic. Add and Xor operations are implemented using Verilog operators '+' and '^' respectively. Circular shift operations are performed through rewiring of the nets. Each round takes 2 clock cycles to complete, therefore 28 clock cycles are required to complete 14 rounds of BLAKE algorithm. After completion of 14 rounds, finalization module computes final or next chaining hash value by taking contents of V_Reg and CV_Reg as input.

3.3 Implementation of Keccak

Implemented data path for Keccak is shown in Fig. 6. 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. 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. 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 represents the concatenation operation. Compression function of Keccak consists of five steps. In Fig. 6 each step is denoted by the symbol as specified in Keccak specifications. These steps are θ, ρ, π, χ and *i*. We show here all steps just for clarity of algorithm representation. In fact, we have combined these steps during implementation, wherever possible. We have implemented ρ and as a single step. Similarly we have combined χ and *i* as a single step. 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 state of A_Reg is 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.

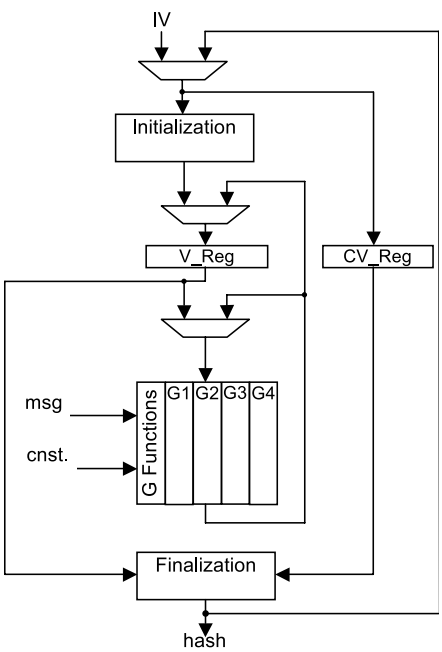

Figure (5) Datapath of BLAKE

3.4 Implementation of Skein

Implemented data path for Skein is shown in Fig. 7. Add_ Subkey module is a 256 bit adder having inputs from key schedule module and Mux_1. Select input S1 of the Mux_1 will be at logic 0 only for the first clock cycle and pass the original message block to Add_Subkey module to add it with subkey0. After first clock S1 remains at high logic and passes the result of the previous round to add it with the next subkey. Output of Add_Subkey is used as input of Demux_1. Demux_1 and Mux_2 have same select input S2.

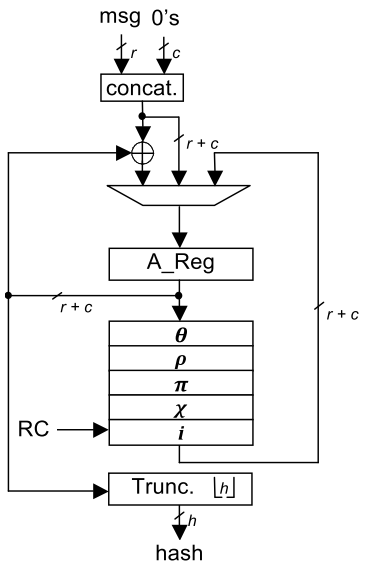

Figure (6) Data path of Keccak

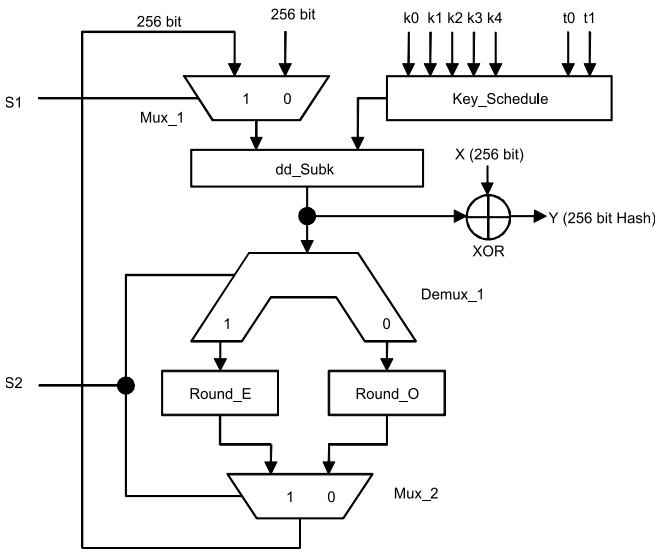

Figure (7) Data path of Skein-256

If S2 is at logic 1, the data path through module Round_E will be selected otherwise data path through module Round_O is selected. Each round consists of MIX and Permutation operations as described in Fig. 2. Round_E and Round_O modules are same, except the values of left rotate constant R involved in MIX operations. Internal designs of modules Round_E and Round_O are shown in Fig. 8. The key scheduling can be implemented in several ways. The simplest one is to store the extended key and extended tweak in two shift registers and clock and rotate the shift registers once for each subkey (Ferguson, *et al.*, 2007).

Hardware architecture of key schedule module is shown in Figure 9. A preprocessing circuit generates and provides extended key k_4 and extended tweak t_2 to key schedule module. We had supposed that the two parameters $k_0,\ldots,k_4$ and $t_0,\ldots,t_2$ are available at the start of the circuit operation and are loaded into the circular shift registers k (320 bit) and t (192 bit). Key Schedule module generates subkeys on every falling edge of clock on the basis of initial key (k0, k1, k2, k3) and tweak value (t0, t1). Add_Subkey, Round_O, and Round_E modules give 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-256, 19 clock cycles will be required. Final hash value will be available after 19 clock cycles at the output of the XOR gate, as shown in Fig. 7.

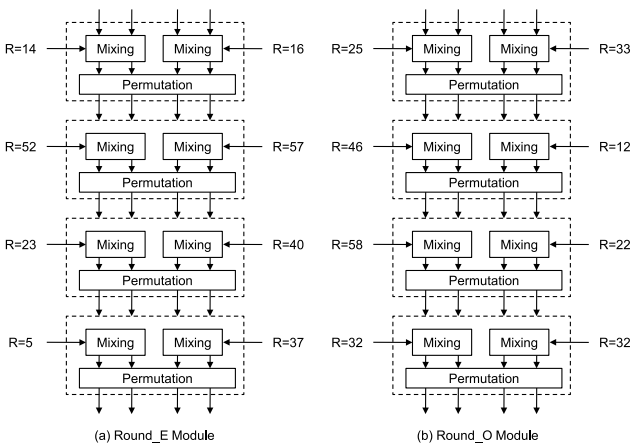

Figure (8) Details of Round_E and Round_O Modules

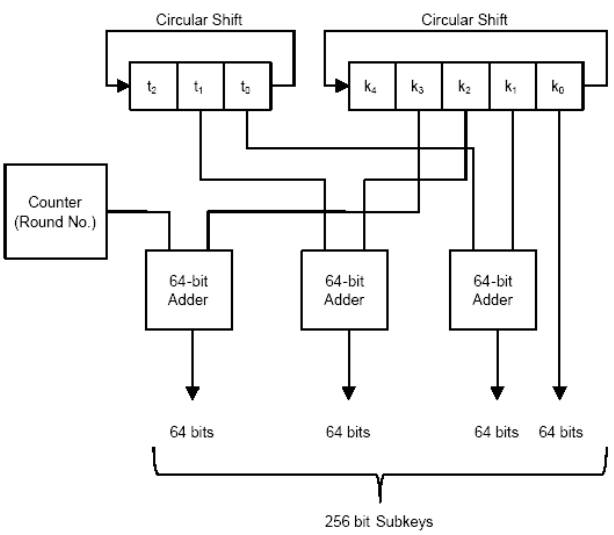

Figure (9) Hardware Architecture of Key Schedule Module

4. 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. We coded our designs according to the optimized techniques specific to these devices and mentioned in relevant XST documents of these devices. Detailed device specifications are: Virtex 7 585T, speed grade 3, package FG1157 (7v585tffg1157-3) and Virtex 6 LX365T, speed grade 3, package FF1156 (6vlx365tff1156-3). The resulting clock frequency and area utilization after place and route are reported. Table 2 shows achieved area consumption ($Area$), clock frequency (F_{max}) and time delay (T) for implemented designs.

Table 2. Results for SHA-3 Candidates (256-bit variants)

Candidate	Xilinx Virtex 7			Xilinx Virtex 6		
	$Area$ [Slices]	F_{max} [MHz]	T [ns]	$Area$ [Slices]	F_{max} [MHz]	T [ns]
BLAKE	1566	135.355	7.388	1602	131.961	7.578
Keccak	1043	255.885	3.908	1043	231.481	4.320
Skein	782	135.501	7.380	786	142.126	7.036

4.1 Througput

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

$$TP = \frac{Block\ Size}{T_{hash}}$$

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

$$T_{hash} = T \cdot N_{clk}$$

Where T is the time period of the system clock and N_{clk} is the number of clock cycles required for a valid hash output. Table 3 shows these parameters for each candidate.

4.2 Througput Per Area

Throughput per area is a significant performance measure, such that it combines the performance effect of both area and speed in a single value. It measures the contribution of each unit area to throughput and hence the efficiency of the implementation. In evaluation of hardware performance of SHA-3 candidates in second round, NIST has considered throughput-to-area ratio as a major deciding factor (NIST, 2010).

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

$$TPA = \frac{TP}{Area}$$

Where TP is the throughput of the design. $Area$ is the occupied area of the design on the chip. In our case we have chosen the number of slices as a unit of area consumption. Table 4 shows throughput per area results.

Table 3. Throughput Results for SHA-3 Candidates (256-bit variants)

Candidate			Xilinx Virtex 7			Xilinx Virtex 6		
	[bits]	[cycles]	[ns]	[ns]	[Gb/s]	[ns]	[ns]	[Gb/s]
BLAKE	512	28	7.388	206.86	2.47	7.578	212.18	2.41
Keccak	1088	24	3.908	93.79	11.60	4.320	103.68	10.49
Skein	256	19	7.380	140.22	1.83	7.036	133.68	1.91

Table 4. Throughput per Area Results of SHA-3 Candidates (256-bit variants)

Candidate	Xilinx Virtex 7			Xilinx Virtex 6		
	[Mb/s]	[Slices]	[Mb/slice]	[Mb/s]	[Slices]	[Mb/slice]
BLAKE	2475.063	1566	1.580	2413.001	1602	1.506
Keccak	11600.12	1043	11.122	10493.805	1043	10.061
Skein	1825.698	782	2.335	1914.961	786	2.436

5. 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 results have been reported on Virtex 6 and Virtex 7. All reported work to date utilized Virtex 5 at most. In order to compare efficiency of our designs we have processed and compiled results for Virtex 5. Table 5 summarizes and portrays the complete picture of all reported works up till now.

In case of BLAKE all previously reported results are for 10 rounds of the algorithm. However, in final round specifications, BLAKE-256 has been tweaked from 10 to 14 rounds. The results listed here have been calculated again for 14 rounds based on the reported clock frequencies and number of cycles consumed for respective designs [6, 10-13]. Baldwin et al (Baldwin, et al., 2010) reported their work only for the Skein-512 variant. We estimate their results for Skein-256.

Our results for 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. Virtex 5 results show that our designs for BLAKE and Keccak are top performers in terms of throughput and placed third in case of Skein. In terms of throughput per area ratio, all of our Virtex 5 designs are placed second with very marginal differences in all cases.

Table 5. All reported results to date (256-bit variants)

SHA-3 Candidate	Implementer	Device	F_{max} [MHz]	*Area* [slice]	*TP* [Mbps]	*TPA* [Mbps/slice]
BLAKE	**This work**	Virtex 7	135.35	1566	2475.06	1.58
	This work	Virtex 6	131.96	1602	2413.00	1.51
	This work	Virtex 5	124.55	1739	2277.47	1.31
	(Sklavos, *et al.*, 2009)*	Virtex	50.00	3101	914.29	0.29
	(Aumasson, *et al.*, 2007)*	Virtex 5	100.00	1217	1765.52	1.45
	(Baldwin, *et al.*, 2007)*	Virtex 5	91.35	1653	835.19	0.50
	(Matsuo, *et al.*, 2010)*	Virtex 5	115.00	1660	637.13	0.38
	(Gaj, *et al.,* 2010)*	Virtex 5	117.06	1871	2066.71	1.10
Keccak	**This work**	Virtex 7	255.88	1043	11600.12	11.12
	This work	Virtex 6	231.48	1043	10493.81	10.06
	This work	Virtex 5	254.06	1419	11517.61	8.12
	(Bertoni, *et al.*, 2007)	Virtex 5	122.00	1330	5200.00	3.91
	(Strömbergson, *et al.,* 2009)	Spartan3A	85.00	3393	4800.00	1.41
	(Baldwin, *et al.*, 2010)	Virtex 5	195.73	1971	6263.00	3.17
	(Matsuo, *et al.*, 2010)	Virtex 5	205.00	1433	4196.72	2.93
	(Akin, *et al.*, 2010)	Spartan 3	81.40	2024	3460.00	1.71
	(Akin, *et al.*, 2010)	Virtex-II	136.60	2024	5810.00	2.87
	(Akin, *et al.*, 2010)	Virtex 4	142.90	2024	6070.00	3.00
	(Gaj, *et al.*, 2010)	Virtex 5	238.38	1229	10806.51	8.79
Skein	**This work**	Virtex 7	135.50	782	1825.70	2.33
	This work	Virtex 6	142.13	786	1914.96	2.44
	This work	Virtex 5	109.67	912	1477.70	1.62
	(Baldwin, *et al.*, 2010)	Virtex 5	83.58	893	972.00	1.08
	(Matsuo, *et al.*, 2010)	Virtex 5	115.00	854	283.18	0.33
	(Gaj, *et al.*, 2010)	Virtex 5	116.35	843	1567.62	1.86
	(Long, *et al.*, 2009)	Virtex 5	114.94	931	408.68	0.44
	(Tillich, *et al.*, 2009)	Vritex 5	68.40	937	1751.00	1.87
	(Tillich, *et al.*, 2009)	Spartan 3	26.14	2421	669.00	0.28

* Figures are extrapolated or estimated from published results as explained in text

6. Performance Comparison

In evaluation of hardware performance of SHA-3 candidates in second round, NIST has considered throughput per area ratio as a major deciding factor (NIST, 2010). Keeping this criterion in mind we chose best results for each candidate, against each device, from Table 6. Fig. 10 represents the performance comparison in a graphical view based on these results. It is clear from graph that Keccak is far ahead of other two candidates in terms of throughput per area ratio. Skein stands second in terms of throughput per area on all devices. But still the difference is large, almost 5:1 ratio between Keccak and Skein on each device. BLAKE and Skein are computationally intensive designs as compared to Keccak. Keccak's computational over head involves simple XOR, AND and rotate operations, which leads to high frequency designs. Moreover, Keccak single compression operation operates on large block of message, which yields high throughput per area results.

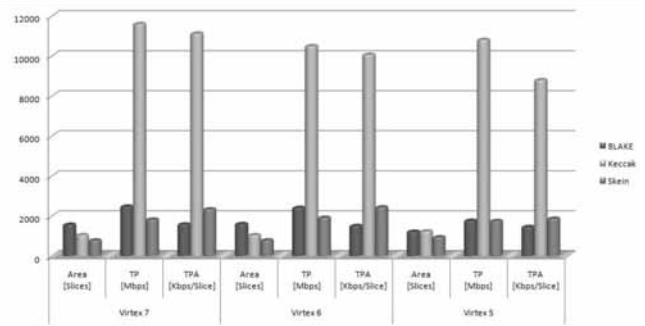

Figure (10) Performance Comparison of SHA-3 Candidates

7. Conclusion

In this work we have presented efficient hardware implementations of SHA-3 finalists: BLAKE, Keccak and Skein. We reported the implementation results of 256-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. 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.

We used the 256-bit variants for our implementations. Other variants are 224, 384 and 512, as specified by NIST for SHA-3. Present work may easily be modified for all these variants.

References

Akin, A., Aysu, A., Onur, C.U., Savas, E. (2010) 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, August 23-24, 2010, pp. 1-12, USA

Aumasson, J., Henzen L., Meier, W. and Phan R., W., (2007) SHA-3 Proposal BLAKE version 1.3, http://131002.net/blake/blake.pdf, pp. 1-79

Baldwin, B., Hanley, Hamilton, M., Lu, L., Byrne, A., Neill and Marnane (2010) FPGA Implementations of the Round Two SHA-3 Candidates, 2nd SHA-3 Candidate Conference, Santa Barbara, August 23-24, 2010, pp. 1-18, USA

Bertoni, G., Daemen, J., Peeters, M. and Assche (2007) The KECCAK SHA-3 Submission version 3, http://keccak.noekeon.org/Keccak-submission-3.pdf, pp. 1-121

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

Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, Kohno, Callas, and Walker, J. (2007) The Skein Hash Function Family Version 1.3, http://www.skein-hash.info/sites/default/files/skein1.3.pdf, Oct 2010, pp. 1-100

Gaj, K., Homsirikamol, E., and Rogawski, M. (2010) Fair and Comprehensive Methodology for Comparing Hardware Performance of Fourteen Round Two SHA-3 Candidates using FPGAs. In: Proceedings of Cryptographic Hardware and Embedded Systems workshop (CHES 2010), Santa Barbara, August 17-20, 2010, pp. 1-15, USA

Long, M. (2009) Implementing Skein Hash function on Xilinx Virtex-5 FPGA platform http://www.skein-hash.info/sites/default/files/skein_fpga.pdf, pp. 1-15

Matsuo , S., Knezevic, M., Schaumont, P., Verbauwhede, Satoh, A., Sakiyama and Ota, K. (2010) How Can We Conduct Fair and Consistent Hardware Evaluation for SHA-3 Candidate?, 2nd SHA-3 Candidate Conference, Santa Barbara, August 23-24, 2010, pp. 264--278, USA

NIST (2007) National Institute of Standards and Technology (NIST) Cryptographic Hash Algorithm Competition. http://www.nist.gov/itl/csd/ct/

NIST (2010) NIST Interagency Report 7764, Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition', http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Round2_Report_NISTIR_7764.pdf, pp. 1-38

Szydlo, M. (2005) SHA-1 collisions can be found in 263 operations, CryptoBytes Technical Newsletter

Stevens, M. (2006) Fast collision attack on MD5, ePrint-2006-104, http://eprint.iacr.org/2006/104.pdf, pp. 1-13

Sklavos, Nicolas and Kitsos, Paris. (2010) 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, July 5-7, 2010, pp. 1-4, Greece.

Strömbergson, Joachim (2009) Implementation of the Keccak Hash Function in FPGA Devices http://www.strombergson.com/files/Keccak_in_FPGAs.pdf, pp. 1-4

Tillich, S. (2009) Hardware implementation of the SHA-3 candidate Skein, ePrint-2009-159, http://www.eprint.iacr.org/2009/159.pdf, pp. 1-7

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